Planning and Scheduling for Large-Scale Distributed Systems

by

Han Yu
M.S. University of Central Florida, 2002
B.E. Shanghai Jiao Tong University, 1996

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall 2005

Major Professors:
Dan. C. Marinescu
Annie S. Wu

Abstract

Many applications require computing resources well beyond those available on any single system. Simulations of atomic and subatomic systems with application to material science, computations related to study of natural sciences, and computer-aided design are examples of applications that can benefit from the resource-rich environment provided by a large collection of autonomous systems interconnected by high-speed networks. To transform such a collection of systems into a user's virtual machine, we have to develop new algorithms for coordination, planning, scheduling, resource discovery, and other functions that can be automated. Then we can develop societal services based upon these algorithms, which hide the complexity of the computing system for users.

In this dissertation, we address the problem of planning and scheduling for large-scale distributed systems. We discuss a model of the system, analyze the need for planning, scheduling, and plan switching to cope with a dynamically changing environment, present algorithms for the three functions, report the simulation results to study the performance of the algorithms, and introduce an architecture for an intelligent large-scale distributed system.

To my wife: Lingxia Song

and my parents: Jinling Yu and Chenglan Wang

Acknowledgments

This dissertation would not have been completed without the help, advice, and encouragement of my advisors, committee members, related research associates, and graduate students. I would like to thank all of these people for their continuous support on my Ph.D. study and research work. In particular, I would like to thank **Dr. Dan C. Marinescu** for leading me to the field of distributed computing, workflow management, and planning, for his countless amount of time spent on sharing precious ideas with me on tough research problems, and for his financial support on my personal life and research work. I would like to thank **Dr. Annie S. Wu** for introducing the field of evolutionary computation to me, and for her invaluable guidance and advice which have shown me a model of what a research scientist truly should be. I would also like to thank other committee members, **Dr. Howard Jay Siegel**, **Dr. Fernando Gomez**, and **Dr. Ladislau Bölöni** for their critical reviews and insightful suggestions on my dissertation. I am very grateful for having all of them in my committee. I would also like to thank my wife, **Lingxia Song**, my parents, **Jinling Yu** and **Chenglan Wang**, and other family members for supporting me to pursue my Ph.D. study in U.S. and for their continuous encouragement through all these years. Finally, I would like to thank National Science Foundation for supporting this work, and thank the University of Central Florida for providing a nice and professional environment for me to conduct my study and research.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

LIST OF PUBLICATIONS

Journal Articles

1. Annie S. Wu, Han Yu, Shiyuan Jin, Guy Schiavone, and Kuo-Chi Lin, "An incremental genetic algorithm approach to multiprocessor scheduling", in IEEE Transactions on Parallel and Distributed Systems, 15(9), pages 824-834, 2004.

2. Xin Bai, Han Yu, Yongchang Ji, and Dan C. Marinescu, "Resource Matching and a Matchmaking Service for an Intelligent Grid", in International Journal of Computational Intelligence, 1(3), pages 197-205, 2004.

3. Han Yu, Xin Bai, and Dan C. Marinescu, "Workflow management and resource discovery for a grid environment", Parallel Computing, 31(7), pages 797-811, 2005.

4. Xin Bai, Han Yu, Guoqiang Wang, Yongchang Ji, Gabriela M. Marinescu, Dan C. Marinescu, and Ladislau Boloni, "Coordination in intelligent grid environments", in Proceedings of the IEEE, 93(3), pages 613-630, 2005.

5. Han Yu, Dan C. Marinescu, Annie S. Wu, and Howard Jay Siegel, "Genetic-based planning with recursive subgoals", to appear in International Journal of Computational Intelligence.

Book Chapter

1. Xin Bai, Han Yu, Guoqiang Wang, Yongchang Ji, Gabriela M. Marinescu, Dan C. Marinescu, and Ladislau Bölöni, "Intelligent grids", submitted as a chapter to book "Grid Computing: Software Environments and Tools".

Conference Papers

1. Kuo-Chi Lin, Han Yu, Lei Zhou, Zheng Xia, Alex F. Sisti, and Steven M. Alexander, "Robust control of a swarm of UCAVs", In Proceedings of SPIE Volume 4716, Enabling Technologies for Simulation Science VI, pages 108-115.

2. Annie S. Wu, Han Yu, Kuo-Chi Lin, and Guy Schiavone, "Length variation in response to a changing environment", In Proceedings of Genetic and Evolutionary Computation Conference (GECCO) 2002 Late Breaking Papers, pages 482-489, July 2002, New York, NY.

3. Han Yu, Dan C. Marinescu, Annie S. Wu, and Howard Jay Siegel, "A genetic approach to planning in heterogeneous computing environments", In Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), IEEE Computer Society Press, April 2003, Nice, France.

4. John C. Sciortino, Jr., Annie S. Wu, Han Yu, Brian N. McQuay, Ayse S. Yilmaz, and Vijayanand C. Kowtha, "ISR team formation for unattended ground sensor networks using evolutionary algorithms", In Proceedings of the 48th Joint Electronic Warfare Conference, May 2003.

5. Han Yu, Annie S. Wu, Kuo-Chi Lin, and Guy Schiavone, "Adaptation of length in a nonstationary environment", In Proceedings of Genetic and

Evolutionary Computation Conference (GECCO) 2003, pages 1541-1553, Springer-Verlag LNCS Series, July 2003, Chicago, IL.

6. Ayse S. Yilmaz, Brian N. McQuay, Han Yu, Annie S. Wu, and John C. Sciortino, Jr., "Evolving sensor suites for enemy radar detection", In Proceedings of Genetic and Evolutionary Computation Conference (GECCO) 2003, pages 2384-2395, Springer-Verlag LNCS Series, July 2003, Chicago, IL.

7. Han Yu, Ning Jiang, and Annie S. Wu, "Simulating GA search in a dynamic grid environment", In Proceedings of Genetic and Evolutionary Computation Conference (GECCO) 2003 Late Breaking Papers, pages 330-337, July 2003, Chicago, IL.

8. Han Yu, Xin Bai, Guoqiang Wang, Yongchang Ji, and Dan C. Marinescu, "Metainformation and workflow management for solving complex problems in grid environments", Heterogeneous Computing Workshop in the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004).

9. Han Yu, Ning Jiang, and Annie S. Wu, "Populating genomes in a dynamic grid", Genetic and Evolutionary Computation Conference (GECCO), pages 418-419, June 2004, Seattle, WA.

10. Han Yu, Dan C. Marinescu, Annie S. Wu, and Howard Jay Siegel, "Planning with recursive subgoals", in Proceedings of the 8th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES), pages 17-27, September 2004, Wellington, New Zealand.

11. Xin Bai, Han Yu, Yongchang Ji, and Dan C. Marinescu, "Resource matching and a matchmaking service for an intelligent grid", in Proceedings of

International Conference on Computational Intelligence (ICCI), pages 262-265, December 2004, Istanbul, Turkey.

12. Han Yu and Annie S. Wu, "An incremental approach to the proportional GA", in Genetic and Evolutionary Computation Conference (GECCO) Late Breaking Papers, June 2005, Washington D.C.

Technical Report

1. Han Yu and Annie S. Wu, "A simulated annealing approach to multi-processor task scheduling", Technical Report CS-TR-04-06, University of Central Florida, 2004.

CHAPTER 1

INTRODUCTION

In this dissertation, we address the problem of planning, scheduling, and plan switching for large-scale distributed systems. A *large-scale distributed system* contains a large collection of interconnected computers and provides transparent access to computing resources for applications requiring substantial CPU cycles, very large memories, and massive secondary storage spaces that cannot be provided by a single machine. Examples of complex computing tasks that can be supported by large-scale distributed systems include simulations of atomic and subatomic systems with application to material science, computations related to study of natural sciences, and computer-aided design.

We define a *system* to be a computer or a collection of computers that can provide the computation for users. Examples of systems include a personal computer, a cluster of homogeneous computing nodes, and a large-scale distributed system. A system supports the computation by providing a group of *computing services*. Examples of computing services that can be supported by a system include data compression, image processing, and word processing. We refer *computing resources* to include hardware resources (e.g., processing nodes), computing services, and data related to a computing task. When submitting a computing task to a system, the user sends a *request* that specifies the set of initial data and the expected results of the computing task. A user may also specify the

preferences or restrictions on the use of computing services in a system. When a request is accepted, the system executes the computing task and sends the results back to the user.

Planning and scheduling are among the essential functions for executing a complex computing task. A complex computing task typically requires a large amount of computing resources and has a long execution time. The request for executing a complex task may not always be satisfied by directly using a single computing service. Instead, we need planning to select a group of computing services in a system and arrange them into an activity graph that specifies the order and data dependencies among the computing services. This activity graph is called a *plan*, and the execution of the plan must achieve the desired goals of the computing task. After such a plan is available, the next step is to execute the computing task. This step, called scheduling, is achieved by assigning the execution of each computing service in a plan to a computing node. If a system contains only one processing node, scheduling becomes a trivial problem by assigning all computing services on one node. The execution of a computing task may fail due to various reasons, e.g., the resources required by the computing task are unavailable. If failure occurs, we need to create a new plan for the computing task and execute the new plan. Alternatively, we can switch the execution of a computing task from one plan to another plan so that the execution may continue without the need to create a new plan. Such a method is called *plan switching*. After a plan switching is performed, we need another round of scheduling to continue the execution of the computing task until a computing task can successfully finish, or a failure is inevitable. Figure 1.1 shows the role of planning, scheduling, and plan switching in executing a complex computing task.

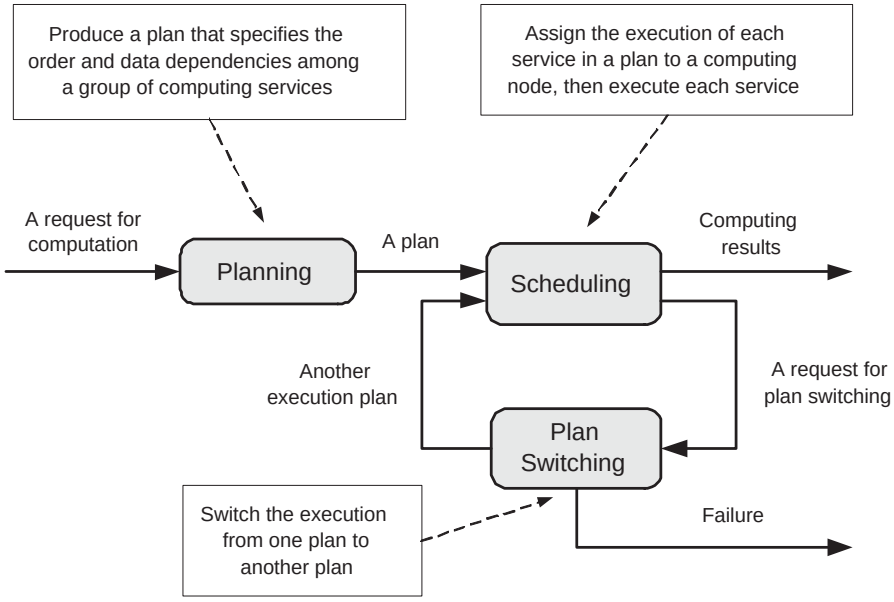


Figure 1.1: The role of planning, scheduling, and plan switching in executing a complex computing task. We may perform scheduling and plan switching multiple times if failure occurs during the execution.

1.1 Motivation

We have seen in the past decade the emergence of the large-scale distributed system as a new paradigm of high performance computing. A computational grid is a typical example of a large-scale distributed system [1, 2]. The seti@home project, set up to detect extraterrestrial intelligence, is a successful application of large-scale distributed computing. The project is designed to take advantage of the unused cycles of PCs and workstations distributed around the world. Once a computer joins the project, the application is activated by a mechanism similar to the one for screen savers. The participating computers form a primitive computational grid structure. Once a system is enabled to accept work, it contacts a load distribution service, receives an assignment for a specific task, and

starts computing. When interrupted by a local user, this task is checkpointed and migrated to the load distribution service for redistribution to another available system.

We list the important features for large-scale distributed systems as follows.

(a) Scale. A large-scale distributed system may contain tens of thousands or more nodes.

(b) Heterogeneity and diversity. A large-scale distributed system contains computers with different processors and system architectures. The communication channels linking these computers differ in terms of latency and bandwidth. Computers may have different hardware architecture and use different operating systems. Diverse application software may run on these computers. Multiple versions of the same application software may be available in a system.

(c) Autonomy of individual nodes. There is no single administration authority in a large-scale distributed system. The computers in a system may belong to different administrative domains with possibly different access, security, and resource management policies [3]. As a result, it is not possible to have a central administrative domain to manage and coordinate the use of all computing resources in a system.

(d) The dynamic and open-ended character. The state of a distributed system typically changes very quickly. The state of a system is given by the status of computing resources, which is further determined by the number of available computing nodes, the availability of computing services and data, and so on. A large-scale distributed system is also open-ended: new resources are constantly added to a system; existing ones are updated or removed. Users are allowed to supply computing resources and share with other users in a system. Keeping track of the state of a large-scale distributed system can be a daunting task.

(e) The dominant service policy in a large-scale distributed system is based upon a "best effort". Enforcing end-to-end quality of service appears to be rarely possible.

(f) There is a large population of users with individual and often conflicting requirements on computing resources. Coordination among different computing tasks or processes is very important for a large-scale distributed system to assure the fairness and quality of service for each user.

(g) Computing tasks submitted by individual users are typically complex and resource-intensive [4]. The complexity of a task is difficult to quantify. It has multiple facets. It may refer to the number and relationship of component activities, the predictability of the amount of resources needed for the completion of individual activities, the security constraints, the presence or absence of deadlines for a computing task, the duration of individual activities, the diversity of resources used, and so on [3].

All above features request us to develop new algorithms for coordination, planning, scheduling, resource discovery, and other functions to transform a large-scale distributed system into a user's virtual machine. Based on these algorithms, we are able to build services that can hide the complexity of a large-scale computing system and provide users coordinated access to computing resources in the system.

In this dissertation, we address the problem of developing planning and scheduling algorithms for such a system. Both planning and scheduling for large-scale distributed systems require a search in a very large solution space and in a changing problem environment due to the complexity of computing tasks and the large scale and dynamics of the computing system. Our goal is to develop planning and

scheduling algorithms that can produce valid plans and schedules for computing tasks and adapt quickly to the changes in the computing environment.

1.2 Addressed Problems and the Approach

1.2.1 Planning and Scheduling

Planning and scheduling are two essential functions for large-scale distributed systems. Briefly speaking, the function of *planning* is to automatically create a plan for executing a computing task. A plan can be represented as a directed activity graph whose vertices are the computing activities to be executed and whose arcs denote data and flow control dependencies among activities. Without planning, such a plan has to be created manually by an individual user who needs considerable knowledge of both the system and the computing task. The function of *scheduling* is to assign the execution of each computing activity in a plan to a computing node in the system and minimize the execution time of the computing task.

Planning and scheduling for a large-scale distributed system are non-trivial problems for the following two reasons. First, a large-scale distributed system provides a dynamic computing environment. The conditions of a system may change very quickly during the course of planning and scheduling. Quick adaptation to changing conditions is essential to the success of the planning and scheduling algorithms. Second, planning and scheduling for a large-scale distributed system typically requires a search in a huge solution space due to the large scale of the

system and the complexity of a computing task. A successful algorithm should be able to balance the efficiency of the search and the quality of solutions.

1.2.2 Our Approach: Evolutionary Computation

We study the evolutionary computation (EC) approaches to planning and scheduling for large-scale distributed systems. The EC approach is one type of parallel search method often used to solve difficult optimization problems. EC is inspired by a fundamental principle of natural selection, the *survival of the fittest*. EC approaches have been applied successfully to numerous optimization problems and have been shown to perform well on problems with non-stationary environments. Typical EC approaches include the genetic algorithm (GA), genetic programming (GP), evolutionary strategies (ES), and evolutionary programming (EP).

A genetic algorithm evolves a population of solutions for multiple generations. Each individual in the population encodes a candidate solution to a given problem. Initially, these solutions are randomly generated. With each new generation, a GA evaluates the performance of every individual with a fitness function that gives a numeric value indicating the quality of the encoded solution. Selection of the individuals is based on their fitness. Good solutions have a higher chance of being selected in the population. Selected individuals are subjected to crossover and mutation to explore new search spaces without completely losing the existing solutions that have been evolved. A newly generated population is found. The GA then repeats these evolutionary steps to generate the next population. The following pseudo code shows the basic steps of a typical genetic algorithm.

```
procedure GA
```

```
{
   initialize population;
   while termination condition not satisfied do
   {
      evaluate current population;
      select parents;
      apply genetic operators to parents to create offspring;
      set current population equal to the new offspring population;
   }
}
```

1.3 Implementation

1.3.1 The Need for an Intelligent Middleware

We believe that the need for an intelligent middleware for large-scale distributed systems is amply justified by the complexity of the computing tasks submitted by users and the heterogeneity and multitude of resources contained in the system. A *middleware* is a software that serves as the interface between the application and the system-level services. A middleware for a large-scale distributed system supports a set of services similar to those of the operating system of a centralized system (e.g., scheduling, event handling, authentication, and data staging). Other functions of the middleware, such as resource discovery and brokerage, do not have a counterpart in the operating system of a centralized system. Such a middleware is expected to make a large-scale distributed system more usable by

allowing a more efficient use of resources and a well-balanced mix of individualistic versus societal objectives.

An "intelligent middleware" means more system automation and less user intervention. The intelligence of a middleware is supported by two elements in the design of the middleware: a multi-agent framework and ontology-based knowledge sharing.

1.3.2 A Multi-agent Framework and Ontology-based Knowledge Sharing

Our middleware consists of a variety of services, the execution of which is supported by a group of autonomously running software agents. A *software agent* is a program capable of taking actions to reach desired goals and reacting to changes in the environment. We assign each agent the role to perform a pre-specified service of a middleware. These agents work coherently with each other to achieve the overall functionality of a middleware. We classify these services into two categories: *core* services and *end-user* services. Core services, or *societal* services, refer to the system-wide services that support coordinated and transparent access to computing resources. End-user services are specialized services offered by autonomous service providers and they carry out the actual computations for end users. A collection of various core services is essential for a middleware. Core services should be persistent and reliable, and typically there are multiple copies of the same core service in a system to ensure efficiency and quality of service. End-user services, on the other hand, can be transient in nature. The providers of

end-user services may suspend their support temporarily or permanently. Therefore, the reliability of end-user services may not be guaranteed.

Knowledge sharing among agents is also of great importance to the intelligence of middleware. Each agent in a middleware maintains its own knowledge base. The knowledge base stores all essential knowledge for an agent to support its desired services and share with other agents in a middleware. Knowledge sharing among agents requires that all agents adopt the same structure for knowledge, also called *ontology*. An ontology defines a common set of terms for entities who need to share information in a system. Ontologies serve as a common language among all agents in the middleware to ensure a seamless inter-operability among the agents.

1.3.3 Architecture of the Middleware

Figure 1.2 shows the basic architecture of the middleware. A non-exhaustive list of core services for the middleware includes: authentication, coordination, planning, matchmaking, brokerage, scheduling, plan switching, information, event handling, monitoring, and simulation services. The *authentication* service ensures the security of the environment. The *coordination* service acts as a proxy for a user. It receives computing tasks delivered from users, monitors the execution of each computing task, and supports coordinated resource sharing among concurrent tasks. After a computation finishes, either with success or failure, the coordination service sends the results back to the user. The *planning* service is responsible for creating plans for a given computing task. The *matchmaking* service supports the function of resource discovery and attempts to find the

computing resources that best match the request from a computing task. The *brokerage* service maintains the up-to-date information related to the available services in a system. The *scheduling* service generates optimized schedules for executing computing tasks. The function of the *plan switching* service is to switch the execution of a computation from one plan to another when failure occurs during the execution of the current plan. The *information* service has similar functions as a DNS in the Internet. It is responsible for locating every registered service (both core and end-user services) in a system. The *event handling* service provides a message passing method for event handling and inter-service communication. The *monitoring* service is responsible for tracking the current status of all computing resources in a system. The *simulation* service provides statistical data for monitoring the performance of the system.

Figure 1.3 shows the basic structure of the ontology for this middleware. The ontology is composed of classes and slots. Each class specifies one entity of knowledge for a large-scale distributed system. A basic set of classes includes "Task", "Process Description", "Case Description", "Data", "Activity", "Transition", "Service", "Resource", "Hardware", and "Software." Classes can be further described with a set of attributes. For instance, the entity "data" can be defined as a class in the ontology with each instance of class "data" corresponding to a data item in the system. We can specify a data with the attributes such as "name", "location", "format", etc. Each attribute is defined as a slot for class "data." An ontology can be extended with the inclusion of additional classes and slots for specific computing domains. Although all above core services are developed exclusively for this middleware, they are open-ended to other systems that share the same ontology as the one used by this middleware.

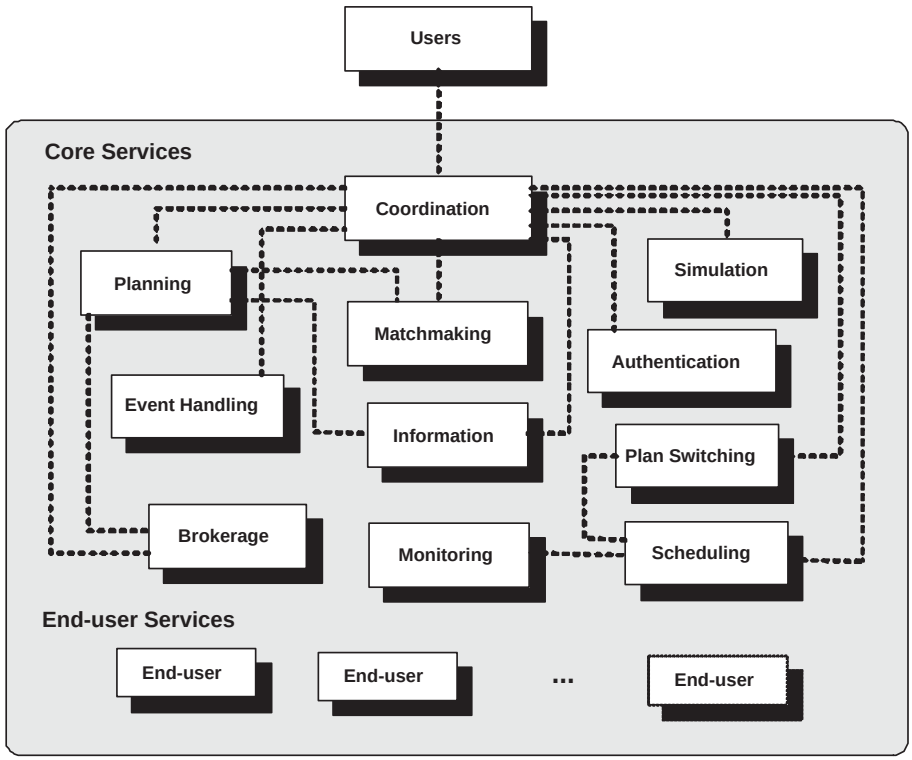


Figure 1.2: The basic architecture of an intelligent middleware for large-scale distributed systems. The middleware consists of a collection of core services and end-user services. A list of core services includes authentication, coordination, planning, matchmaking, brokerage, scheduling, plan switching, information, event handling, monitoring, and simulation services.

1.4 Contributions of this Dissertation

The focus of this dissertation is implementing the functions of planning, scheduling, and plan switching for large-scale distributed systems. This dissertation also extends the work in evolutionary computation and distributed computing. The contributions of the dissertation are:

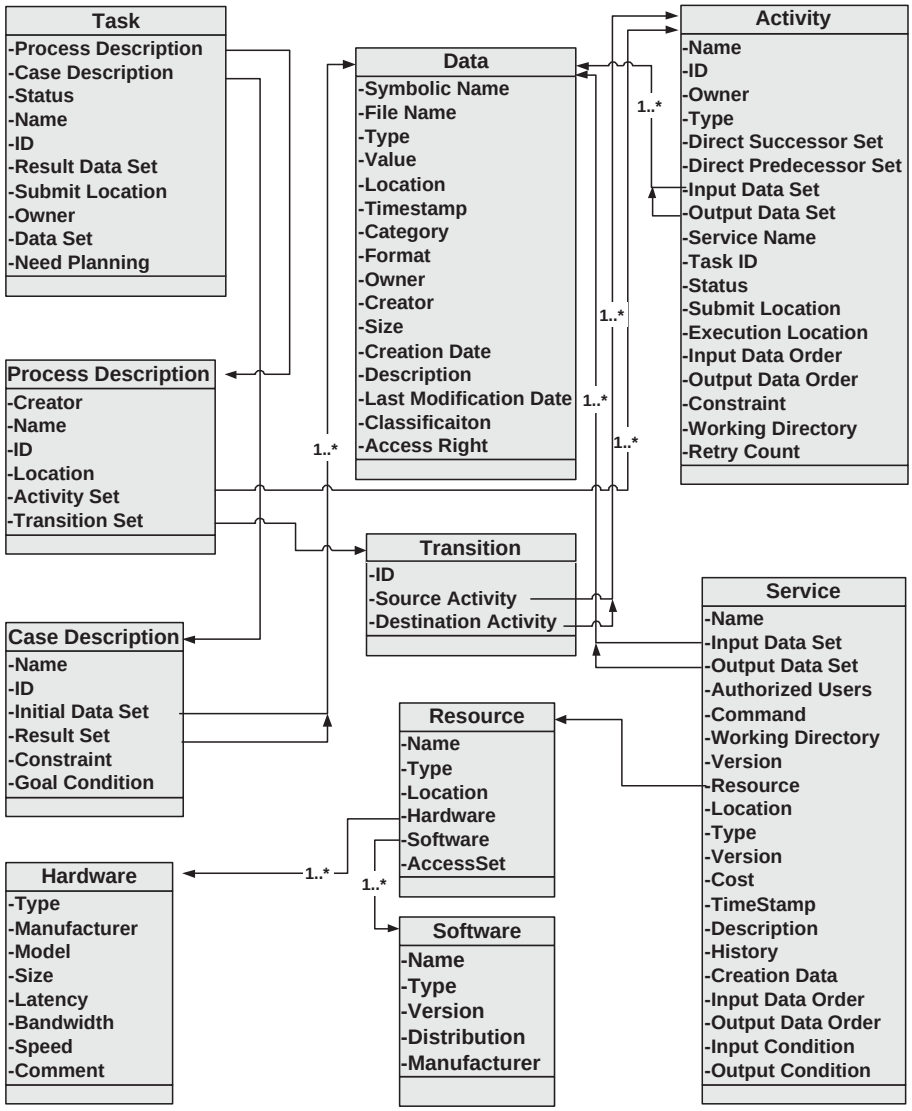


Figure 1.3: A basic set of classes and attributes for the ontology.

(a) *A genetic algorithm approach to AI planning problems and planning for large-scale distributed systems*

The problem of planning has been extensively studied by AI researchers. Numerous planning models and algorithms have been proposed. Many existing planning algorithms take advantage of the heuristics extracted from the domain

knowledge to improve the efficiency of the search for a solution. These heuristics, however, are typically domain specific and are consequently difficult to generalize to other problems. In addition, planning for a large-scale distributed system requires quick adaptation to a changing problem environment. Heuristics that are useful for a current problem condition may not be as effective when the problem condition changes.

We investigate a GA approach to planning. We are one of the few forerunners to apply a GA-based approach to AI planning. Our approach is non-deterministic, requires less domain knowledge than traditional planning approaches, and exhibits consistent performance on a variety of planning domains.

In addition, we study the problem of planning for large-scale distributed systems. In this work, we formulate the problem of planning based on the ontologies defined for the middleware; address the issue of replanning, a process of adapting a plan to dynamic computing environments; and classify the problem of planning into two categories: deterministic and non-deterministic planning. We apply an adapted approach to planning for large-scale distributed systems. We evaluate the performance of the approach both on a real-world scientific computing domain and in a simulation computing environment.

(b) *A genetic algorithm approach to multi-processor scheduling and scheduling for large-scale distributed systems*

Scheduling for large-scale distributed systems is similar to multi-processor scheduling, but it is much more complicated. Many existing multi-processor scheduling algorithms suffer from similar problems as traditional AI planning algorithms. First, they use heuristics that cannot be applied to various domains. Second, they do not work well in a dynamic problem environment.

We apply a GA-based approach to multi-processor scheduling. Our GA has two distinguishing features compared to traditional GAs. First, it uses a flexible representation scheme that allows GA to evolve both the structure and value of the solutions. This feature is effective to give GA a complete exploration in the search space, while other approaches with restricted representation do not. Second, it uses an incremental fitness function that starts out rewarding simpler goals and gradually increases in difficulty until a complete solution is found. Experiments on benchmark task graphs show that this approach produces comparable or better performance as compared to traditional deterministic scheduling approaches.

In addition, we investigate the effectiveness of this approach in dynamic problem environments. This study gives us an indication of how well this approach may perform in a large-scale distributed system, a naturally dynamic computing environment.

Finally, we study the problem of planning for a large-scale distributed system. Our formulation of the problem takes into account both the heterogeneity and dynamics of the system. The modified scheduling approach is able to handle conditional and iterative execution of tasks, which is a major extension from the original approach.

(c) *Plan switching for large-scale distributed systems*

A large-scale distributed system is dynamic in nature. We cannot fully guarantee the success of the execution of a computation. Quick adaptation to a changing computing environment is necessary during the course of a computation. Replanning is an approach to deal with this problem, but it incurs a significant amount of computational cost in search for a new plan.

We introduce a method called plan switching as an alternative solution to deal with the uncertainty in a large-scale distributed system. Given a group of plans available to perform a computing task, the function of plan switching is to switch the execution to a backup plan when the current plan fails. If plan switching is successful, we stop the execution of the computation, switch the computation to the backup plan, reschedule the computation for the backup plan, and resume the computation. Plan switching serves as the "glue" to integrate the functions of planning and scheduling for a large-scale distributed system.

We present an approach to plan switching. The main idea of this approach is to locate execution points from other plans in parallel with the execution of the current plan. When the execution cannot proceed, we continue the execution of a computing task from a selected execution point in another plan. We perform a simulation study and investigate the effect of various parameters on the performance of this approach.

(d) *Study of representation and incremental search strategies in evolutionary computation*

We extend the study of GA on two aspects. First, we study the solution representation in GA and its effects on the search performance. In particular, we study the behavior of variable length representation in both static and dynamic problem environments. Second, we study the effectiveness of using incremental search strategies in a GA. In the GA-based planning approach, we use an incremental method by dividing the search into multiple independent phases. In the GA-based scheduling approach, we use an incremental, dynamic fitness function. We perform experiments to evaluate the effectiveness of using these incremental search strategies in a GA.

(e) *Development of an intelligent middleware for large-scale distributed systems*

Our goal for developing large-scale distributed systems is to improve the usability of the systems, i.e., to minimize the user intervention while ensuring the fairness and quality of service for all users. Our solution is the introduction of an intelligent middleware. The intelligence of the middleware is supported by two essential elements that we bear in mind in our design: a multi-agent framework and ontology-based knowledge sharing among agents.

The middleware consists of a variety of services, the execution of which is supported by a group of autonomously running software agents. Each agent is assigned a role to perform a pre-specified service of a middleware. We classify all services into two categories: core services and end-user services. Core services refer to the system-wide services that support coordinated and transparent access to computing resources, and they are the essential elements of a middleware. End-user services are specialized services offered by autonomous service providers that perform the actual computations for users.

Knowledge sharing among agents is also of great importance to the intelligence of the middleware. Each agent in this middleware maintains its own knowledge base. The knowledge base stores all essential knowledge for an agent to support its desired service and share with other agents in the middleware. Knowledge sharing requires that all agents adopt the same structure for knowledge. The ontology we define for the middleware provides a common language for all agents to ensure seamless collaboration. Our ontology is extensible to allow the inclusion of the knowledge for specific computing domains.

CHAPTER 2

PLANNING ALGORITHMS AND PLANNING SERVICES

In this chapter, we study the problem of planning for large-scale distributed systems. The goal of planning is to produce valid execution plans for a given computation. We first study the traditional AI planning problems that are simpler but similar to planning for large-scale distributed systems. We present a GA-based approach to planning and evaluate the performance on two artificial planning domains. We also introduce an effective heuristic, recursive subgoal strategy, for subgoal division and ordering in planning problems that contain conjunctive goals and the division of recursive subgoals for these planning problems maintain the serializability among subgoals. Finally, we address the problem of planning for large-scale distributed systems. We formulate the problem, present a modified genetic approach to the problem, and evaluate the performance of the approach.

2.1 AI Planning

2.1.1 Introduction to AI Planning

AI planning, or simply planning, has a wide range of real-world applications. A planning problem is associated with a system, the state of which can be changed with a set of actions. Given an initial state, a set of goal specifications, and a set of actions, the objective of planning is to construct a valid sequence of actions, or a plan, to reach a state that satisfies the goal specifications starting from the initial state of a system. Many complex problems require planning. A simple example of a planning problem is the process of solving a puzzle, given a set of pieces with different geometric shapes scattered on the floor.

Much effort has been devoted to building computational models for a variety of planning systems. Our work is based on STRIPS-like domains [5] in which the change of the system state is given by operators and their preconditions and postconditions. In addition, we are interested in the linear planning problems in which solutions are represented by a total order of operators that must be executed sequentially to reach the goal.

Definition 1. We define a planning problem to be a four-tuple

$$\Pi \ = \ (\mathcal{P}, \mathcal{O}, \mathcal{I}, \mathcal{G}).$$

$\mathcal{P}$ is a finite set of ground atomic conditions (i.e., elementary conditions instantiated by constants) used to define the system state. $\mathcal{O} = \{\mathcal{O}_i\}$, where $1 \leq i \leq |O|$ is a finite set of operators that can change the system state. Each operator has three attributes: a set of preconditions $\mathcal{O}_i^{pre}$, a set of postconditions $\mathcal{O}_i^{post}$, and a cost $\mathcal{C}(\mathcal{O}_i)$. $\mathcal{O}_i^{post}$ consists of two disjunctive subsets: $\mathcal{O}_i^{post+}$ and

$\mathcal{O}_i^{post-}$. $\mathcal{O}_i^{post+}$, the add list, is a set of conditions that must be true for a system state after the execution of the operator; $\mathcal{O}_i^{post-}$, the delete list, consists of a set of all conditions that do not hold after the execution of the operator. $\mathcal{I} \subseteq \mathcal{P}$ is the initial state and $\mathcal{G} \subseteq \mathcal{P}$ is the set of goal conditions. A plan Δ contains a finite sequence of operators. An operator may occur more than once in a plan.

An operator is *valid* if and only if its preconditions are a subset of the current system state. A plan Δ solves an instance of Π if and only if every operator in Δ is valid and the result of applying these operators leads a system from state $\mathcal{I}$ to a state that satisfies all the conditions in $\mathcal{G}$.

Planning is generally more difficult than a typical search problem. First, most planning problems involve extremely large search spaces. Second, the existence of a solution is not always guaranteed, i.e., a goal may not be reachable from a given initial state with the execution of a finite number of operators. Third, the size of an optimal solution cannot be easily anticipated. As a result, it is difficult to quantify the time and space complexity of planning algorithms.

2.1.2 Previous Work on Planning Algorithms

Erol et al. [6] provide a comprehensive analysis of the computational complexity of domain-independent planning problems with STRIPS-like operators. The authors investigate the effect of the nature of planning operators on the decidability of a planning problem. Their study shows that, when planning is decidable, the time complexity of a domain-independent planning algorithm depends on numerous factors, such as whether conditional effects and negative preconditions are allowed in a planning problem.

The *Graphplan* approach exploits the fact that the operator space of a planning problem is much smaller than its state space [7]. This approach first generates a compact planning graph consisting of all the possible operators at every time step. Operators that interfere with one another can coexist in a graph. The plan graph guides the plan formation and is extended in every time step of the search. The Graphplan approach is a sound and complete partial-order planner. Experimental results show that Graphplan outperforms other planning algorithms in a variety of problem domains.

Jonsson et al. study the efficiency of universal planning algorithms [8]. They conclude that universal planners that run in polynomial time and polynomial space cannot satisfy even the weakest types of completeness. If, however, one of the polynomial requirements is removed, constructing a plan that satisfies completeness becomes a trivial problem. They also propose *Stocplan*, a randomized approach to universal planning under a restricted set of conditions. They show that this approach can construct plans that run in polynomial time and use polynomial space and also satisfy both soundness and completeness for these problems. Experiments indicate that the performance of Stocplan is competitive with Graphplan.

Another approach to planning is to partially reuse existing plans. This approach consists of two steps, plan matching and plan modification. Nebel and Koehler [9] analyze the relative computational complexity of plan reuse versus planning from scratch. Their study shows that the problem of plan reuse is intractable and the efficiency of this approach is not guaranteed. Generally, reusing an existing plan is harder than planning from scratch. This approach is expected to work better only when the new planning problem is sufficiently close to the

old one. Plan matching, a necessary step in this approach, can be the bottleneck in computation time.

Bonet and Geffner [10] show that a general planning algorithm can be transformed into a heuristic search algorithm by extracting the heuristics from problem representations. They introduce two planners, *HSP*, a hill-climbing planner, and *HSP2*, a best-first planner. Both planners are forward state planners and have competitive performance with Graphplan. Both planners assume that all subgoals are independent; therefore, admissible heuristic functions can be defined and they never overestimate the cost.

A different direction of planning research focuses on domain-specific planning for limited problem domains. Korf and Taylor study useful search heuristics for the Sliding-tile puzzle [11]. They present the work on an accurate admissible heuristic function in the $\mathcal{IDA}^*$ search algorithm. The heuristics used include the linear conflict heuristic, last moves heuristic, and corner-tile heuristic. These heuristics are shown to improve the search performance of the $\mathcal{IDA}^*$ search algorithm.

Korf and Felner [12] use a disjoint pattern database heuristic in planning algorithms. With this heuristic, the subgoals are first split into disjoint subsets such that an operator affects only the subgoals in one subset. The values obtained for each subset are then combined to form the result of the heuristic evaluation function. Korf and Felner use this technique to search for a solution for the Sliding-tile puzzle and for Rubik's cube. Their approach successfully finds optimal solutions to different instances of 5×5 Sliding-tile puzzles. The results indicate that this heuristic improves the search efficiency by decreasing the number of nodes traversed during the search. Nevertheless, the computational cost of this approach still increases very quickly with the increase of puzzle size. Finding

optimal solutions to 6×6 or larger Sliding-tile puzzles is still considered to be a formidable task.

All of the above approaches except Stocplan are deterministic approaches that tend to require large coverage of a search space to generate a good result. Problem specific heuristics can be used to reduce the size of a search space; however, heuristics for one class of problems may not be applicable to other classes of problems. Evolutionary computation (EC) has emerged as a competitive technique in planning research. Because of an element of randomness in their implementation, the search results of EC methods may vary over different runs and these methods are not guaranteed to find an optimal solution. EC methods, however, are robust and can consistently find solutions that are approximate to an optimal solution.

The works of Koza [13] and Spector [14] are among the early attempts to applying genetic programming to planning problems. Both report positive results in the Blocks World domain. Koza's approach uses a set of specifically designed functions in the solution representation. These functions work only on domains that contain only one block stack, which largely restricts the applicability of this approach. Spector uses less specific functions in the plan representation. His approach successfully generates a universal plan for the 3-block domain. This success, however, may be partially attributed to the small search space of the 3-block domain, which is not difficult for deterministic search algorithms. No experiments on larger problem domains are reported.

Muslea [15] presents a GP approach to planning. *Sinergy* is a general linear planning system built on the GP paradigm. Sinergy extends the expressive power of traditional planning algorithms in the encoding of planning problems. Experiments are performed on the Single and 2-Robot Navigation problems and on the Briefcase problem. Results indicate that Sinergy can handle problems that

are one or two orders of magnitude larger than *UCPOP*, a deterministic partial order planner with the equivalent expressive power [16]. Sinergy works only on problems with conjunctive goals.

Another example of a GP based algorithm is *GenPlan* [17]. GenPlan uses a linear structure to encode solutions. Experiments on three domains, the Blocks World domain, the Briefcase domain, and the Logistics domain, show that GenPlan can solve the same problems as Sinergy but with fewer generations. The authors also report a study on five GP seeding strategies and show that these strategies improve the search quality on the Blocks World domain [18]. Seeding partial solutions and keeping some randomness in the initial population appear to benefit GP performance.

2.1.3 A Genetic Algorithm Approach to Planning

Our genetic approach to planning differs from the traditional GA in three aspects. First, we use an indirect encoding method for individuals to eliminate invalid operators in a plan. Second, we introduce two novel crossover schemes in an attempt to reduce the disruption from the genetic operations as a result of this indirect encoding method. Third, the search process is divided into multiple phases, with each phase an independent GA run. This multi-phase process enables the GA to build solutions incrementally.

2.1.3.1 Solution Encoding

The solution to a planning problem is encoded as a sequence of genes, where each gene represents a single operator in the plan. The simplest way to encode such a plan is to use direct encoding: each operator is represented by an integer and the sequence of integers encodes the sequence of operators in a plan. Because an operator may not be valid in every system state, a direct encoding method cannot avoid invalid operators in a plan.

We use an indirect encoding method instead. Each gene is represented as a floating point number x, $0 \leq x < 1$. Every number in the solution is mapped to a valid operator for the corresponding system state. The result of this mapping depends on the value of the floating point number and the set of valid operators in a given system state. This method ensures that all genes in a solution will represent valid operators. For example, assume that in a given state there are four valid operators, $\mathcal{O}_1$, $\mathcal{O}_2$, $\mathcal{O}_3$, and $\mathcal{O}_4$. Then a floating point number x is mapped as follows:

$$
\begin{aligned}
0.00 \leq x < 0.25 &\longmapsto \mathcal{O}_1 \\
0.25 \leq x < 0.50 &\longmapsto \mathcal{O}_2 \\
0.50 \leq x < 0.75 &\longmapsto \mathcal{O}_3 \\
0.75 \leq x < 1.00 &\longmapsto \mathcal{O}_4.
\end{aligned}
$$

As it is generally difficult to determine the size of the optimal solution (i.e., the number of operators in the optimal solution), we use a variable length representation. Variable length representations allow a GA to evolve individuals of different lengths and is especially useful for domains in which the sizes of the optimal solutions cannot be easily determined or estimated. Early attempts of using variable length representation include Smith's LS-1 learning system [19],

Goldberg's messy GA [20], Koza's GP [13], and Harvey's SAGA [21]. Recent studies have focused on the effects of non-coding genes (i.e., the sections of genes that do not contribute to solution encoding) on variable length GA performance. These studies find that the inclusion of non-coding regions affects the probability of disruption caused by crossover and as a result is more likely to preserve the building blocks that have been evolved [22, 23]. A recent study on a variable length GA under non-stationary problem environments reveals that a variable length GA is more likely to recognize and maintain good build blocks after target changes and hence exhibits better adaptability than a fixed length GA [24].

Although we allow a GA to evolve variable sizes of solutions in this planning approach, we set an upper bound, $MaxLen$, on the individual length. The value of $MaxLen$ should be chosen to ensure GA search quality while not incurring too much computation cost.

2.1.3.2 Population Initialization

The members of the initial population are randomly generated. The lengths of the initial population of solutions are set to reasonable values and do not exceed $MaxLen$.

2.1.3.3 Fitness Evaluation

The goal of planning is to find a solution that satisfies the following three criteria: (a) the solution contains no invalid operators; (b) the sequence of operators leads the system from the initial state to a state that satisfies all goal conditions;

and (c) the cost of the solution is minimized. As the indirect encoding method can eliminate invalid operators in a solution, the evaluation of solutions only focuses on the second and third criteria. Accordingly, the fitness function has two components: the goal fitness f_g and the cost fitness f_c.

The *goal fitness function*, $f_g, 0 \leq f_g \leq 1$, evaluates the match quality between the final state of a solution and the goal specifications. To determine the final state of a solution, we go through every operator from the beginning to the end of a solution. Initially, we set the current state as the initial state of the system. We start from the first operator and change the system to the state after this operator is performed. We repeat the same process on each succeeding operator until we finish all of the operators in the solution. A better match between the final state and the goal specifications results in a higher goal fitness. The goal fitness is typically dependent on the characteristics of the planning problem.

The *cost fitness function*, $f_c, 0 \leq f_c \leq 1$, evaluates the total cost of a solution. The cost of a solution depends on the cost of individual operators and is problem specific. The cost may be related to the latency of executing an operator, the number of arithmetic operations, the amount of data to be transferred, and so on. A solution with low cost has a high cost fitness. In the very simple case when all operators have the same cost, the cost fitness is given by:

$$f_c = \frac{MaxLen - individual\ length}{MaxLen} \tag{2.1}$$

The *overall fitness function* reflects the two aspects of merit:

$$f = a \times f_g + (1 - a) \times f_c \tag{2.2}$$

where a is the weight of the goal fitness and $0 \leq a \leq 1$.

2.1.3.4 Selection and Genetic Operators

Selection

Tournament selection is used to select the individuals that will be the parents of the next generation. In this selection scheme, we randomly pick up two individuals from the current population and compare their fitness values. The individual with the higher fitness value is chosen to be a parent. This process continues until we have generated a new population of the same size as the current one. We do not use elitism to retain the best solution over generations.

Crossover

We implement three different crossover mechanisms: random, state-aware, and mixed. In each case, the children created replace their parents.

Random crossover is similar to GA one-point crossover. Given two parents, we randomly select one crossover point on each parent. The two parents exchange portions of their genetic code relative to the two crossover points. Two children are created; each child inherits a portion of the genetic code from both parents.

A potential problem with random crossover is that the selected crossover points may be associated with different system states. Because we use an indirect encoding method, the mapping from floating point numbers to operators is dependent on the system state, see Section 2.1.3.1. Therefore, it is very likely that the genes to the right of the crossover points will be mapped to a different sequence of operators after crossover although they are still represented by the same floating point numbers.

State-aware crossover addresses the problem of state mismatching in random crossover. We randomly select a crossover point from the first parent. We restrict the crossover point of the second parent to those that match the first crossover point. Two states match if the same genetic code will be mapped to the same sequence of operators from these two states. If no such crossover can be found, we do not perform the crossover and both parents are included in the population of the next generation. State-aware crossover attempts to preserve partial solutions that have been evolved in the search.

Mixed crossover combines random and state-aware crossover. We randomly select the first crossover point and check if state-aware crossover can be performed. If so, we perform the state-aware crossover on the two parents. Otherwise, we randomly select the second crossover point and carry out a random crossover.

Mutation

Every gene has equal probability of being mutated. In every mutation, a new randomly generated floating point number replaces the old one.

2.1.3.5 A Multi-Phase Search Procedure

We use a multi-phase approach to build solutions to a planning problem incrementally. We divide the GA search into multiple phases. Each phase is an independent GA run and consists of a fixed number of generations. In the first phase, we take the initial state of the system as the state where the search starts. When a phase ends, the best solution found in this phase is stored and the final state of the solution is taken as the initial state for the search in the subsequent

phase. The GA search ends when a valid solution is found at the end of one phase, or a predefined number of phases have finished. The final solution of a GA run is the concatenation of the best solutions from all phases. The procedure of a multi-phase GA consists of following steps.

```
(1) Start GA. Initialize population.
(2) While the stopping condition is not met, do
    (a) While fewer than the specified number of generations are
        evolved in the current phase, do
        (i)   Evaluate each individual in the population.
        (ii)  Select individuals for the next generation.
        (iii) Perform crossover and mutate operations on selected
              individuals.
        (iv)  Replace old population with new population.
    (b) Select the best solution for this phase and keep it.
    (c) If a valid solution is found, go to step 3. Otherwise,
        randomly initialize population and start the next phase. The
        search in the new phase starts from the final state of the
        best solution in the previous phase.
(3) Construct the final solution by concatenating the best solutions
    from all phases.
```

The rationale of using a multi-phase GA is to divide a large search problem, such as planning, into smaller problems so that each small problem can be tackled separately. The search is expected to reach closer to the problem goal after each individual phase. Partial solutions that evolved in previous phases are kept and prevented from disruptions caused by genetic operations.

One method of further improving the effectiveness of a multi-phase GA is to specifically assign different search goals for each individual phase so that the

search becomes more "goal-driven." This method, however, requires domain-specific knowledge and may not be applicable to all planning problems. We address this issue in Section 2.1.5 and present a heuristic for subgoal division and ordering.

2.1.4 Experiments and Performance Evaluation

We test our GA approach to planning on two classical planning problems, the Towers of Hanoi and the Sliding-tile puzzle. Each experiment is run multiple times and the average performance is reported here. In each individual run we use a different random seed.

2.1.4.1 Towers of Hanoi

In the Towers of Hanoi problem, there are three stakes, A, B, C, and n disks, $\mathcal{D}_1, D_2, \ldots, D_n$ of increasing size. $\mathcal{D}_1$ is the smallest disk and $\mathcal{D}_n$ is the largest disk. Initially, all of the disks are on stake A. The goal is to move all of the disks to stake B in a minimum number of steps. In each step, only one disk can be moved from one stake to another stake. Larger disks are not allowed to be moved on top of smaller disks. The minimum number of steps to reach a goal has been proven to be $2^n - 1$ [25]. Figures 2.1 and 2.2 show the initial and goal configurations for the 5-disk Towers of Hanoi problem.

We define six operators for the Towers of Hanoi problem. They are: *move*(A, B), *move*(A, C), *move*(B, A), *move*(B, C), *move*(C, A), and *move*(C, B). The operator *move*(A, B) moves the disk on top of stake A to the top of stake B.

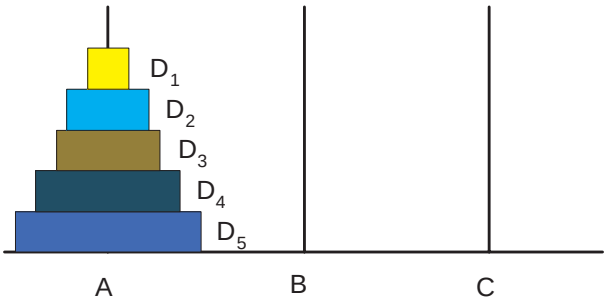


Figure 2.1: The initial configuration of the 5-disk Towers of Hanoi problem.

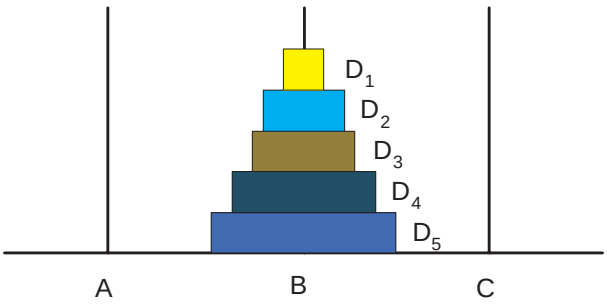


Figure 2.2: The goal configuration of the 5-disk Towers of Hanoi problem.

Other operators are defined similarly. An operator may not be valid for all problem configurations. For instance, we cannot execute the operator *move*(A, B) if stake A is empty or the disk on top of stake A is larger than the disk on top of stake B. An indirect encoding method can be used to eliminate invalid operators in a solution.

A feature of this problem is that larger disks are more important than smaller ones for a plan to succeed. A minimum of 2^{i-1} operators are required to move $\mathcal{D}_i$ from stake A to stake B between the initial and goal configurations, regardless of the positions of other disks. To evaluate the goal fitness, we assign greater weight to larger disks to reinforce their importance: $\mathcal{D}_i$ has a weight of 2^{i-1}. The total

weight of all n disks is $2^n - 1$. The goal fitness is calculated using the following equation:

$$f_g = \frac{total\ weight\ of\ all\ disks\ on\ stake\ B\ in\ the\ final\ state}{2^n - 1} \tag{2.3}$$

The cost fitness is given by equation 2.1. In our experiments, we set the size of initial individuals to the length of the optimal solution, $2^n - 1$. The value of $MaxLen$ is $10 \times (2^n - 1)$.

In this experiment, we use random crossover and test both the single-phase GA and multi-phase GA approaches with the same number of generations. Table 2.1 shows the parameters for this experiment. For the single-phase GA the maximum number of generations allowed is 500; for the multiple-phase GA every phase contains 100 generations and the maximal number of phases allowed is 5.

We perform ten runs in each case and pick the individual with the highest goal fitness in each run. Then we average the fitness and the length of these individuals. We also calculate the average number of generations required to find the best solution of a run. Table 2.2 summarizes these results.

Our data show that the multi-phase algorithm performs better than the single phase GA in terms of goal fitness. The multi-phase GA can find a valid solution in every run for the 5-disk and 6-disk cases. Although the multi-phase GA cannot find a valid solution in some runs for the 7-disk case, it evolves a solution that has higher goal fitness than the single-phase GA. In addition to the improved quality of solutions, the multi-phase algorithm generally requires fewer generations to find the best solution of a run.

The multi-phase algorithm evolves longer solutions than the single-phase GA in the 6-disk and 7-disk problems. This result is probably due to the fact that the

Table 2.1: Parameter settings used in the Towers of Hanoi planning experiments.

Parameter	Value
Population Size	200
Number of Generations	500
Crossover Type	Random
Crossover Rate	0.9
Mutation Rate	0.01
Selection Scheme	Tournament
Tournament Size	2
Weight of f_g	0.9
Weight of f_c	0.1
Number of Disks	5, 6, and 7
Maximal Number of Phases in Multi-phase GA	5

limit of individual length in the multi-phase algorithm is larger than the limit for the single-phase GA. In our experiments, every run can have up to five phases, so the maximum allowed individual length in multi-phase algorithm is five times higher than the one for the single-phase GA.

Still, the multi-phase algorithm is not guaranteed to find a valid solution as the problem size scales up. We believe the problem is partially due to the fact that the goal fitness fails to accurately evaluate the distance between a given final state and the problem goals. For instance, even though we assign more credit to large disks in evaluating the goal fitness, a partial solution might reach a state in which all disks except the largest one are on stake B. This solution will receive a

Table 2.2: Experimental results for the Towers of Hanoi problem. CI = Confidence Interval.

GA Type	Number of Disks	Avg (95%CI) Goal Fitness	Avg (95%CI) Solution Size	Avg (95%CI) Gen. to Find the Best Solution
Single-phase	5	1.0 (0)	49.16 (2.11)	50.04 (10.16)
	6	0.947 (0.011)	99.42 (4.34)	228 (37.36)
	7	0.603 (0.012)	116.46 (6.35)	313.98 (34.86)
Multi-phase	5	1.0 (0)	47.64 (2.44)	43.18 (6.68)
	6	1.0 (0)	101.38 (3.59)	125.42 (14.40)
	7	0.743 (0.070)	156.02 (15.25)	167.12 (21.35)

goal fitness slightly less than 0.5. This state, however, is as far from the goal state as the initial state. Indeed, to reach the goal, all these disks have to be moved away from stake B before the largest disk can be moved to disk B. This difficulty indicates that good heuristic functions still play important roles in improving the performance of our approach.

2.1.4.2 Sliding-tile Puzzles

Sliding-tile puzzles consist of a number of moving blocks and a board on which the blocks can slide. Such problems are sometimes used in AI textbooks to illustrate heuristic search methods [26]. For example, Russell and Norvig [27] discuss the 4×4 Sliding-tile puzzle shown in Figure 2.3. Given an initial configuration such as the one in Figure 2.3(a), the goal is to reach the goal configuration shown in Figure 2.3(b) by moving the blocks without lifting them from the board. Solutions

do not exist for every possible combination of initial and goal configurations of the problem. If we define permutation as an action of exchanging the positions of two tiles on the board, the Sliding-tile puzzles can be categorized into two classes. In the first class, the initial configuration is an even permutation of the goal configuration; in the second class, the initial configuration is an odd permutation of the goal configuration. Johnson and Story show that a solution exists only in the first class of the Sliding-tile puzzles [28].

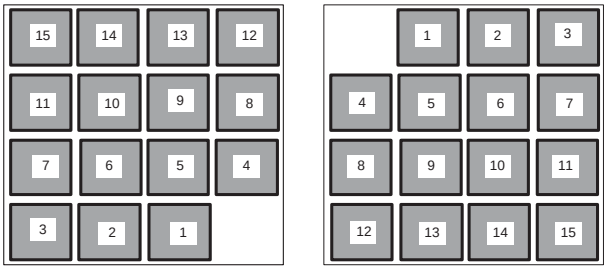


Figure 2.3: The initial and goal configurations of a 4×4 Sliding-tile puzzle. (a) The initial configuration. (b) The goal configuration.

We define four operators for the Sliding-tile puzzles. They are *move-up*, *move-down*, *move-left*, and *move-right*. The operator *move-up* exchanges the position of the empty tile with one right below the empty tile. Other operators are defined in a similar way. Clearly, not all operators are valid for all problem configurations. For instance, if the empty tile is located in a corner of the board, only two operators are allowed.

In this experiment, we set the initial size of a solution to be $\frac{n^2 \times (n^2-1)}{2}$, where n is the number of blocks in every row or column. This expression is the number of comparisons needed to sort a set of n^2 values. While the Sliding-tile puzzle is not the same as a sorting problem, e.g., there are restrictions on the tiles that can

be exchanged, we believe that this expression gives a reasonable size with which to start the GA. Previous GA studies have found evidence that a variable length GA will evolve to a necessary solution length regardless of the initial individual lengths [29].

The distance between the current state and the goal configuration is given by the *Manhattan distance* of all tiles [27]. The upper bound on the distance between any two states in a $n \times n$ problem is $(n-1) \times 2 \times (n^2-1)$, where $(n-1) \times 2$ is the longest distance that a single tile may need to move and n^2-1 is the number of tiles. The goal fitness is:

$$f_g = 1 - \frac{Manhattan\ distance\ between\ final\ state\ and\ goal\ configuration}{(n-1) * 2 * (n^2-1)} \tag{2.4}$$

The cost fitness is given by equation 2.1. For the Sliding-tile puzzle, the value of $MaxLen$ is $10 \times \frac{n^2 \times (n^2-1)}{2}$.

In this problem, we test all three crossover mechanisms with up to five individual phases in each run. Table 2.3 shows the parameter settings for the Sliding-tile puzzle experiments.

We perform 50 GA runs for each experiment and select the individual with the highest goal fitness in every run as the solution. In addition to fitness and individual length, we also record the number of runs that find a valid solution and the average computation time for each run. Table 2.4 summarizes our results.

Interestingly, the performance of the three crossover types are very close. All of them can find a valid solution in at least 45 out of 50 runs for the 3×3 case. As the problem size grows, the search performance degrades sharply. The average

Table 2.3: Parameter settings for the Sliding-tile puzzle experiments.

Parameter	Value
Population Size	200
Number of Generations	500
Crossover Type	Random / State-aware/ Mixed
Crossover Rate	0.9
Mutation Rate	0.01
Selection Scheme	Tournament
Tournament Size	2
Weight of f_g	0.9
Weight of f_c	0.1
Board size (n)	3 and 4
Maximal Number of phases in multi-phase GA	5

Table 2.4: Experimental results for the Sliding-tile puzzle. CI = Confidence Interval.

Type of Crossover	# of Tiles	Avg (95% CI) Goal Fitness	Avg (95% CI) Solution Size	Success Runs	Average Time (sec)
Random	9	0.99 (0.009)	35.92 (1.88)	45	13.79
	16	0.948 (0.007)	174.32 (7.67)	5	65.89
State-aware	9	0.998 (0.005)	33.76 (1.86)	49	11.99
	16	0.951 (0.006)	164.12 (9.38)	4	64.61
Mixed	9	1.0 (0)	32 (1.38)	50	11.55
	16	0.952 (0.006)	153.64 (6.38)	5	63.22

Table 2.5: The number of runs when a valid solution is found in each phase for the random, state-aware, and mixed crossover strategies.

Phase	Random	State-aware	Mixed
1	16	42	50
2	23	6	0
3	4	1	0
4	1	0	0
5	1	0	0

size of the solutions increases faster than linearly as the number of tiles increases. The computation time depends heavily on the individual length.

We further investigate the contribution of multiple phases to the search of a solution. We record the number of runs required to find a valid solution in each phase for the 3×3 case. Table 2.5 lists the result for all three crossover mechanisms.

In most of the runs, a valid solution is found within the first two phases. Mixed crossover finds a solution faster than the other two crossover mechanisms. It can always find the best solution in the first phase. State-aware crossover has a greater probability of finding a valid solution in the first phase than the random crossover. Random crossover does not search as fast as the other two crossover mechanisms, but using multiple phases helps it to find a valid solution before the end of the second phase with a very high probability.

2.1.5 Recursive Subgoals

In this section, we present an effective heuristic, called *recursive subgoals*, for subgoal division and ordering. This approach is only applicable to problem domains that contain conjunctive goals and the division of recursive subgoals maintains the serializability among subgoals.

2.1.5.1 Subgoal Ordering and Interaction

The method of finding and achieving subgoals is pervasive in solving many problems. General Problem Solving (GPS) [30, 31], a problem solving program developed by Newell et al., incorporates the heuristic of means-ends analysis in reaching the goals of a problem. The basic idea of means-ends analysis is to match the current state and the goal state by finding the most important difference between the two states. This process consists of multiple steps. In each step, a subgoal is created to eliminate the differences by applying operators in a given problem.

In a planning problem with conjunctive goals, reaching a goal state requires achieving every subgoal defined in the problem. An intuitive and efficient way to solve this type of planning problem is to order the subgoals and reach them one after another until every subgoal is reached. The subgoals in most problem domains, however, are not completely independent. Strong correlations may exist among subgoals. Achieving one subgoal can make the search for the other subgoals easier, more difficult, or even impossible.

Korf presents a detailed study on the interaction of subgoals for a planning problem with conjunctive goals [32]. He classifies three different types of inter-

actions among subgoals: *independent subgoals*, *serializable subgoals*, and *non-serializable subgoals*. If a set of subgoals is independent, reaching any arbitrary subgoals does not affect the difficulty of reaching the rest of the subgoals. Problems with independent subgoals are easy to solve because we can reach the problem goal by approaching every subgoal individually. As a result, the cost of the search is the total amount of cost devoted to every individual subgoal. This type of interaction, however, rarely occurs in planning problems. In some planning problems, it is possible to specify an ordering of the subgoals that have the following property: every subgoal can be reached without violating any subgoal conditions that have been met previously during the search. Such subgoals are called serializable subgoals. The search becomes easier if we are able to recognize this type of subgoal correlation and specify a serializable ordering. On the other hand, if such an ordering does not exist among the subgoals, the subgoals are called non-serializable subgoals.

There is no universal method of dividing and ordering subgoals into serializable subgoals. In addition, proving the serializability of a sequence of subgoals is as difficult as proving the existence of solutions for a planning problem [32]. Therefore, Korf's classification of subgoal interactions is not appropriate for evaluating the difficulty of a planning problem. Barrett and Weld [33, 34] extend the classification of serializable subgoals based on the probability of generating a sequence of serializable subgoals from a randomly ordered set of subgoals. They define *trivially serializable subgoals* for those subgoals that are always serializable given any possible sequences. If a set of subgoals is not trivially serializable, violation of previously met goal conditions might occur during the search for the complete solution. As the cost of backtracking the previous subgoals is exponentially high, a planning problem is tractable only if the probability of a random sequence of subgoals being non-serializable is sufficiently low so that the cost for

backtracking does not dominate the average cost of the algorithm. Otherwise, a planning problem is intractable. These subgoals are called *laboriously serializable subgoals*.

A correct ordering among subgoals is critical for the performance of planning algorithms. Thus, the study of subgoal correlations has acquired the attention of the planning community. One school of thought attempts to pre-process the control knowledge gained from the specifications of operators and goals to construct a total order on a group of subgoals, before the search begins [35, 36, 37, 38]. A second category includes online ordering methods that focus on detecting and resolving goal condition conflicts from an existing partially ordered plan [39, 40].

Another direction of research attempts to extend the expressive power of plans. The term *non-linear planner* refers to those planning algorithms that use a non-linear structure in the formation of plans. Iterative actions and conditions in a typical non-linear plan are represented by single entities. Early work on non-linear planning includes [41, 42, 43]. Recent work attempts to apply induction rules to gain more control over the process of plan formation [44, 45]. These non-linear planning approaches work well only on domains that are recursive in nature and are not applicable to other domains.

2.1.5.2 Planning with Recursive Subgoals

We introduce a strategy of dividing planning goals into a sequence of serializable subgoals. Informally, our strategy is to decompose a planning problem recursively into a set of subgoals and then to define a strict ordering of these subgoals.

We begin our formal description of recursive subgoals with the introduction of the state space graph of a planning problem.

Definition 2. Let $S = \{s_1, s_2, \dots\}$ be a set of all possible *states* of a planning system. Let $\mathcal{O} = \{o_1, o_2, \dots\}$ be a set of *operators* defined for a planning problem. The *goal* of a planning problem can be represented by $\mathcal{G}$ as a set of atomic conditions (see also the definition in Section 2.1.1).

Definition 3. The *state space* of a planning problem can be represented by a directed graph $G = \{V, E, f_e, s_{init}, S_{goal}, f_s, f_o\}$, where

1. $V = \{v_1, v_2, \dots\}$, a set of vertices.
2. $E = \{e_1, e_2, \dots\}$, a set of directed edges.
3. Every edge e_i connects a pair of vertices $\{v_j, v_k\}$, where v_j and v_k are source and destination vertices of an edge, respectively. $f_e \colon E \rightarrow V$ is a function that maps an edge to its source and destination vertices.
4. s_{init} is the initial state of a planning problem. $s_{init} \in S$.
5. S_{goal} is the set of all system states that meet every condition in $\mathcal{G}$. $S_{goal} \subseteq S$.
6. $f_s \colon V \rightarrow S$ is a function that maps every vertex v_i in V to a distinct system state s_i that can be reached from the initial state s_{init}. $f_s(v_i) = s_i$. $f_s(V) \subseteq S$. A planning problem is solvable if $S_{goal} \cap f_s(V) \neq \phi$. For the rest of the notation in Section 2.1.5.2, we assume that a planning problem is solvable.
7. Edges represent the transitions between two system states in $f_s(V)$. $f_o \colon E \rightarrow \mathcal{O}$ is a function that maps every edge e_i in E to an operator o_i. This function does not enforce a one-to-one mapping, i.e. $\exists\ i$ and j, where $i \neq j$ and $f_o(e_i) = f_o(e_j)$.

Definition 4. Let $GOAL = \{g_1, g_2, \ldots, g_n\}$ be a set of *subgoals* defined for a planning problem.

Any subgoal g_i of a planning problem can be represented by $\mathcal{P}_i$ as a set of atomic conditions with the following four properties:

1. $\mathcal{P}_i \subseteq \mathcal{G}$. Subgoals are easier to reach than the goal of a problem because the conditions for subgoals are subsets of the conditions for the problem goal.

2. $\mathcal{G} = \bigcup \mathcal{P}_i, 1 \leq i \leq n$. The problem goal can be reached when we reach a state that meets the conditions for all the subgoals.

3. Let $f_{gs}\colon GOAL \rightarrow S$ be a function of mapping a subgoal g_i to a set of all states that can be reached from s_{init} and meet the conditions for g_i. Clearly, $S_{goal} \subseteq f_{gs}(g_i) \subseteq f_s(V)$. If $\mathcal{P}_i = \phi$, $f_{gs}(g_i) = f_s(V)$; if $\mathcal{P}_i = \mathcal{G}$, $f_{gs}(g_i) = S_{goal}$.

4. Let G_i be the state space graph that consists of all states in $f_{gs}(g_i)$ and transitions between the states. G_i is a subgraph of G.

According to Korf [32], a set of subgoals is serializable if a specific ordering among them exists. Although an optimal solution is not guaranteed to be found, this ordering ensures that a problem is always solvable by following the sequence of the subgoals without ever violating any previously reached subgoals.

We use this definition and give a formal definition of serializable subgoals based on the state space graph of a planning problem.

Definition 5. A set of subgoals in $GOAL$ is *serializable* if it has the following properties:

1. *GOAL* contains an ordered list of subgoals. g_1 is the first subgoal and g_n is the last subgoal. The search for a solution follows the order of the subgoals.

2. $\mathcal{P}_n = \mathcal{G}$ and $f_{gs}(g_n) = S_{goal}$. That is, the set of conditions for the last subgoal is the same as the goal of the problem. If the last subgoal is reached, the problem is solved.

3. $\mathcal{P}_1 \subseteq \mathcal{P}_2 \subseteq \cdots \subseteq \mathcal{P}_{n-1} \subseteq \mathcal{P}_n$. That is, the set of conditions for a subgoal is a subset of the conditions for all subsequent subgoals.

4. $f_{gs}(g_n) \subseteq f_{gs}(g_{n-1}) \subseteq \ldots f_{gs}(g_2) \subseteq f_{gs}(g_1)$. That is, the set of all states that satisfy the conditions for a subgoal is a subset of all states that satisfy the conditions for every preceding subgoal. This property indicates that the state space of a search algorithm can be reduced after reaching intermediate subgoals.

5. Let $G_i = \{V_i, E_i, f_i, s_{init}, S_{goal}, f_s, f_o\}$ be the state space graph of subgoal i, $V_n \subseteq V_{n-1} \subseteq V_{n-2} \subseteq \cdots \subseteq V_1 \subseteq V$. As a result, G_i is a subgraph of G_j, for every i and j, where $1 \leq j \leq i \leq n$.

6. We define $Adjacent(v_i, v_j, G) = true$ if there exists an edge in G that connects v_j from v_i. We define $Connect(v_i, v_j, G) = true$ if $Adjacent(v_i, v_j, G) = true$ or, $\exists v_k$, $Connect(v_i, v_k, G) = true$ and $Adjacent(v_k, v_j, G) = true$. In other words, $Connect(v_i, v_j, G) = true$ if and only if there is a sequence of edges that connects vertex v_j from v_i.

   For instance, in Figure 2.4, $Adjacent(v_1, v_2, G) = Adjacent(v_2, v_3, G) = true$. $Connect(v_1, v_2, G) = Connect(v_2, v_3, G) = Connect(v_1, v_3, G) = true$.

   If a sequence of subgoals is serializable, a graph G_i that corresponds to any subgoal g_i has the following property: for any $v_i \in V_i$, $\exists v_j \in V_{i+1}$,

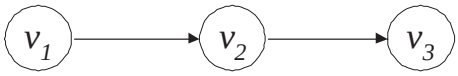


Figure 2.4: An example showing the relation of reachability between vertices in a graph.

Reachable$(v_i, v_j, G_i) = true$. That is, every state that meets the conditions of subgoal g_i can reach at least one state within the state space of subgoal g_{i+1} without violating the conditions set for subgoal g_i. Therefore, serializable subgoals ensure that a solution can be found if it exists. Figure 2.5 gives a graphical representation of this serializability property.

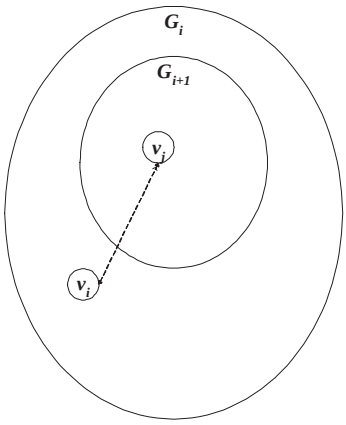


Figure 2.5: A graph showing $v_i \in V_i$, $v_j \in V_{i+1}$, and *Reachable*$(v_i, v_j, G_i) = true$.

The *recursive subgoal* strategy offers a simple and effective solution to the formation and ordering of subgoals from a single goal. This strategy divides the goal of a planning problem recursively into a sequence of subgoals. These subgoals, which will be shown by examples in Section 2.1.5.4, have the following property: reaching one subgoal results in a reduction of a problem to the same

problem at a smaller scale. A formal definition of recursive subgoals is given below.

Definition 6. A sequence of subgoals is *recursive* if it meets the following condition:

Let P be a set of the same problems of different scales. $P = \{P_1, P_2, \ldots, P_m\}$. P_i is smaller than $P_{i'}$, if $i < i'$. Then reaching subgoal g_j in P_i and reaching subgoal g_{j+1} in P_{i+1} are essentially the same problem for $1 \leq j \leq i < m$. Let $G_{i,j}$ be the state space graph corresponding to subgoal g_j of P_i. Then $G_{i,j} \cong G_{i+1,j+1}$; i.e., $G_{i,j}$ and $G_{i+1,j+1}$ are isomorphic.

The division of recursive subgoals does not guarantee serializability among subgoals. We consider three different scenarios as to the applicability of this approach.

1. If a solution exists in any configuration of the problems at any scale, the division of recursive subgoals always preserves the subgoal serializability. An example of a domain belonging to this category is the Tower of Hanoi (see Section 2.1.4.1) in which any two configurations are reachable from each other.

2. If a solution does not always exist in any configuration of a problem at any scale, but reaching one recursive subgoal never leads a problem at a smaller scale to an unsolvable configuration, we can still preserve the subgoal serializability on this problem. We show in Section 2.1.5.4 that the Sliding-tile puzzle falls into this category.

3. Recursive subgoals are non-serializable if we cannot avoid the situation of backtracking any previous recursive goals during the search for a complete solution.

2.1.5.3 Applying Recursive Subgoals to the GA-based Planning Algorithm

If the goal of a planning problem is divided into recursive subgoals, we can apply the multi-phase GA approach to search for solutions to every subgoal. The number of necessary phases to reach a subgoal depends on the difficulty of subgoals. Only when a subgoal is reached in a phase can GA proceed to search for the next subgoal in subsequent phases. The final solution is the concatenation of the solutions to all subgoals that have been attempted in a single GA run. The following pseudo code illustrates the search procedure of this algorithm.

```
(1) Start GA. Initialize population.
(2) Set the first subgoal of the problem as the current search goal.
(3) While the specified number of phases are not finished or the
    final goal is not reached, do
    (a) While the specified number of generations for a phase are
        not finished, do
        (i)   Evaluate each individual in the population.
        (ii)  Select individuals for the next generation.
        (iii) Perform crossover and mutation.
        (iv)  Replace old population with new population.
    (b) Select the best solution for this phase and keep it.
    (c) If the current subgoal is reached, set the next subgoal
        as the current search goal.
    (d) Randomly initialize population and start the next phase.
        The search starts from the final state of the best solution
        in the previous phase.
(4) Construct the final solution by concatenating the best
```

```
solutions from all phases.
```

2.1.5.4 Case Study: the Sliding-Tile Puzzle

Figure 2.6 shows one approach to create recursive subgoals for solving a 4×4 Sliding-tile puzzle. The first subgoal is to have the tiles located in the fourth row and fourth column in their desired positions, see Figure 2.6(a). After the first subgoal is reached, the problem is reduced to a 3×3 Sliding-tile puzzle. Then we work on the second subgoal: moving the remaining tiles in the third row and third column to the correct positions, shown in Figure 2.6(b). After the second subgoal is reached, the problem is reduced to a 2×2 Sliding-tile puzzle, which is very easy to solve. The puzzle is solved after the third subgoal is reached, as shown in Figure 2.6(c).

Johnson and Story also show that if we move any tiles in the Sliding-tile puzzle, we can always maintain the parity of the permutation between the current configuration and the goal configuration [28]. If in the original problem the initial configuration is an even permutation of the goal configuration (i.e., the original problem is solvable), after reaching one recursive subgoal we can always find an even permutation between the current configuration and the goal configuration in the reduced problem. Hence, the reduced problem is solvable as long as the original one is solvable. The goal serializability is preserved in the Sliding-tile puzzle because we are able to reach a subgoal without moving the tiles that have been set in place in previous subgoals.

The recursive subgoal strategy can be applied to any possible configuration of a Sliding-tile puzzle. In a goal configuration the empty tile can be located at any position. If the empty tile is already in one of the corners, we choose

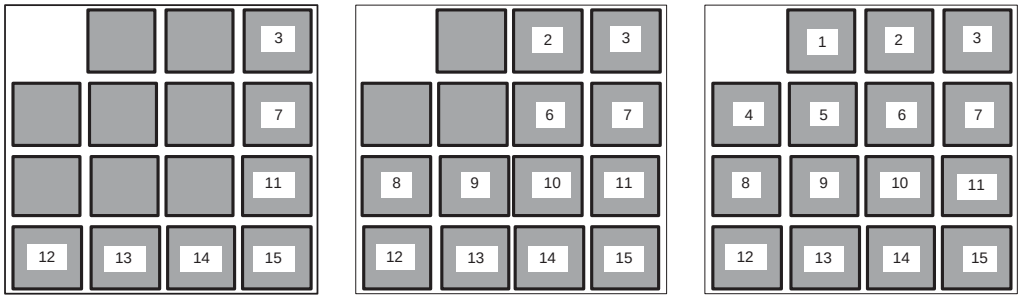


Figure 2.6: The steps for solving a 4×4 Sliding-tile puzzle using the recursive subgoal strategy. (a) The first subgoal. (b) The second subgoal. (c) The third subgoal.

those tiles in the row and column that are farthest to that corner to be in the first subgoal. If the empty tile is not in a corner, we first move it to the nearest corner. The number of moves depends on how far a tile is from the nearest corner. The tiles that are located in the innermost positions of a board are the farthest to the corners. If n is odd, at most $n - 1$ moves are needed; if n is even, at most $n - 2$ moves are needed. After the relocation of the empty tile, the new goal configuration replaces the original one as the goal of the problem. As every operator in the Sliding-tile puzzle is reversible, a reversed sequence of the operators that move the empty tile to the corner will lead the system from the new goal configuration to the original one. The final solution is the solution to the new goal appended by this reversed sequence of operators. Figure 2.7(a) and Figure 2.7(b) show an example of changing the goal configuration in a 4×4 Sliding-tile puzzle. In our experiments, the empty tile is always in top-left corner in the goal configuration.

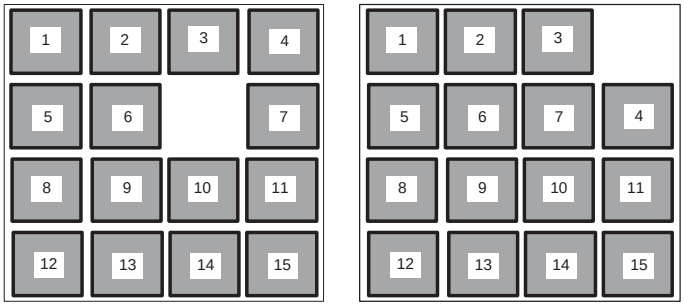


Figure 2.7: An example showing the reconfiguration of problem goals for the recursive subgoal strategy. (a) The original goal configuration. (b) The new goal configuration in which the empty tile is moved to the nearest corner.

2.1.5.5 Performance Evaluation

We evaluate the effectiveness of the recursive subgoal strategy on the Sliding-tile puzzle discussed in Section 2.1.5.4. We compare the performance of the GA-based planning approach with and without the recursive subgoal strategy incorporated (we call it *single-goal approach*). Table 2.6 shows the parameters for this experiment.

In the single-goal approach, the goal fitness is evaluated with the Manhattan distance of all $n^2 - 1$ tiles between the final state of the plan and the goal configuration. The smaller the distance, the higher the goal fitness. In the recursive subgoal approach, we decompose the $n \times n$ Sliding-tile puzzle into $n-1$ subgoals, $\{g_1, g_2, \ldots, g_{n-1}\}$. After the first subgoal is reached, the problem is reduced to a $(n-1) \times (n-1)$ Sliding-tile puzzle. In every subgoal g_i, we focus on the $2 \times (n-i)+1$ tiles that need to be moved to the correct positions. The goal fitness is evaluated with the Manhattan distance of these $2 \times (n-i)+1$ tiles between the final state and the goal configuration.

Table 2.6: Parameter settings used in the experiment.

Parameter	Value
Population Size	200
Crossover Type	Random
Crossover Rate	0.9
Mutation Rate	0.01
Selection Scheme	Tournament
Tournament Size	2
Number of Generations in Each Phase	100

We test both the recursive subgoal strategy and single-goal approach on 4×4, 5×5, 6×6, and 7×7 Sliding-tile puzzles. For each problem size we run both approaches 50 times. In a 4×4 problem, each run has up to 15 phases. We double the number of phases each time the problem size increases by one scale but use the same population size of 200 for all problem sizes.

The experimental results show that the single-goal approach finds solutions in 10 out of 50 runs on the 4×4 Sliding-tile puzzle and none for any larger problems. Table 2.7 shows in experiments where recursive subgoal strategy is incorporated, the number of runs that reach every subgoal. The recursive subgoal strategy significantly improves the search performance. It finds solutions to the 4×4 problem in 35 out of 50 runs. The performance even improves as the problem size increases because more phases are allowed for all subgoals. Table 2.8 reports the average number of phases needed to achieve each subgoal from those runs that find a valid solution. The result indicates that achieving a subgoal does not make the subsequent subgoals more difficult. We observe that the number of

phases needed to reach subgoal g_i is very close to the number of phases needed to reach subgoal g_{i+1} in the next larger problem.

Table 2.7: Experimental results for the recursive subgoal strategy on the Sliding-tile puzzles: the number of runs out of 50 runs that the GA can reach each subgoal g_1-g_6.

Problem Size	4×4	5×5	6×6	7×7
g_1	44	50	50	50
g_2	37	50	50	50
g_3	35	50	49	50
g_4	N.A.	50	49	50
g_5	N.A.	N.A.	49	50
g_6	N.A.	N.A.	N.A.	50

Table 2.8: Experimental results for the recursive subgoal strategy on the Sliding-tile puzzles: average number of phases needed to reach each subgoal from its previous subgoal.

Problem Size	4×4	5×5	6×6	7×7
g_1	6.86	9.34	18.50	28.56
From g_1 to g_2	1.36	5.02	8.32	16.14
From g_2 to g_3	1.07	2.34	5.65	8.74
From g_3 to g_4	-	1.00	2.12	5.34
From g_4 to g_5	-	-	1.00	2.70
From g_5 to g_6	-	-	-	1.00

Next, we study the effect of the parameters on the performance of the approach. We use the parameter settings listed in Table 2.6 as the baseline settings and vary the population size, the crossover rate, and the mutation rate separately. We keep the other parameters the same as the baseline settings while varying each of the above parameters. In each test case, we run the approach 50 times and calculate the number of successful runs (i.e., the runs that find valid solutions) and the average number of phases needed in successful runs. We also evaluate the efficiency of the approach by calculating the average computational time of 50 runs in each case.

Table 2.9 and Figure 2.8 show the performance comparison in cases with different population sizes. The results indicate that noticeable performance gains can be achieved with an enlarged population, which gives the GA more opportunity of sampling in the search space. A population size of 100 is not sufficient to produce competitive results to larger populations. The runs with a population size of 400 need fewer phases to find solutions than runs with the baseline settings. A large population, however, incurs higher computational cost. Figure 2.9 shows results on the execution time in each test case. The execution time of the approach increases quickly as the population size increases.

Table 2.9: The number of successful runs (out of 50) for population size from 100 to 400.

Population Size	4×4	5×5	6×6	7×7
100	21	29	27	42
200	35	50	49	50
400	48	50	50	50

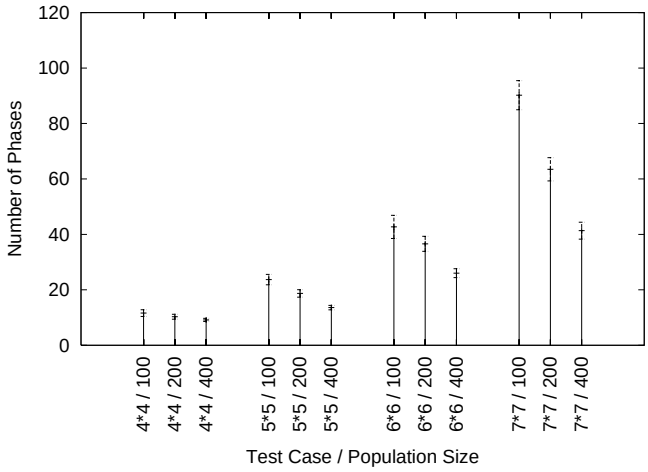


Figure 2.8: The average number of phases (with 95% confidence intervals) needed to find a solution for successful runs with population size varying from 100 to 400.

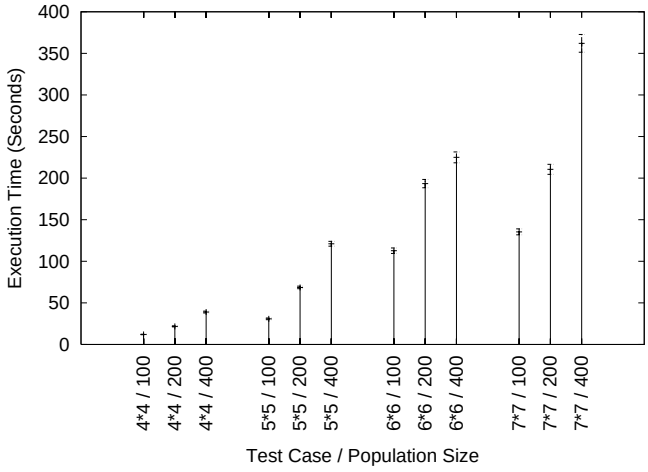


Figure 2.9: The average execution time (with 95% confidence intervals) of 50 runs for population size varying from 100 to 400.

Table 2.10 and Figure 2.10 show the performance comparison in cases with different crossover rates. We test crossover rate of 0.5, 0.8, and 1.0 as well as the baseline settings of 0.9. The results indicate that varying the crossover has little effect on the search performance.

Table 2.10: The number of successful runs (out of 50 runs) for crossover rate varying from 0.5 to 1.0.

Crossover Rate	4×4	5×5	6×6	7×7
0.5	38	49	49	50
0.8	39	49	47	50
0.9	35	50	49	50
1.0	40	48	48	49

Table 2.11 and Figure 2.11 show the performance comparison in cases with different mutation rates. A lower mutation rate (0.005) and a higher mutation rate (0.05) than the baseline settings are tested. All test cases exhibit consistent search results, which indicates that the mutation rate has little effect on the search performance. We suspect the reason is that the crossover method applied in this approach is very disruptive and it already produces ample opportunities for exploring the search space. As a result, the usefulness of a mutation operator is significantly reduced.

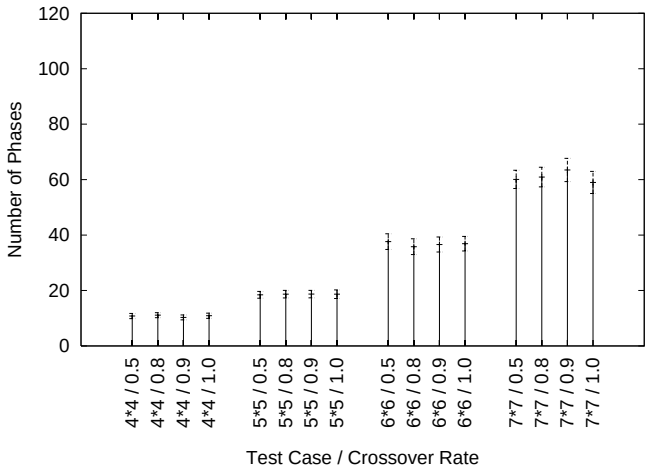


Figure 2.10: The average number of phases (with 95% confidence intervals) needed to find a solution for successful runs with crossover rate varying from 0.5 to 1.0.

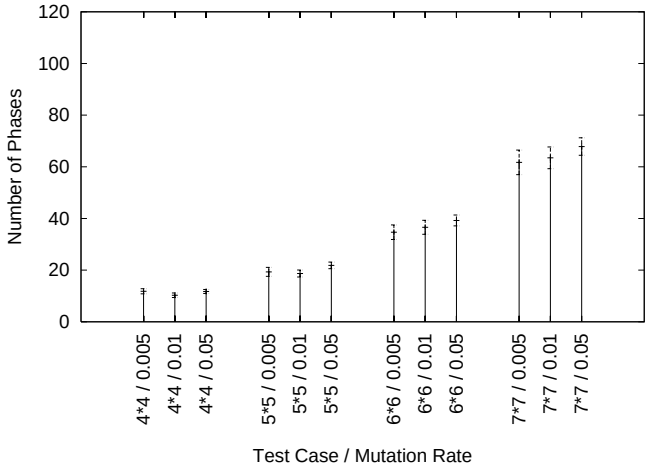


Figure 2.11: The average number of phases (with 95% confidence intervals) needed to find a solution for successful runs with mutation rate varying from 0.005 to 0.05.

Table 2.11: The number of successful runs (out of 50 runs) for mutation rate varying from 0.005 to 0.05.

Mutation Rate	4×4	5×5	6×6	7×7
0.005	33	48	48	48
0.01	35	50	49	50
0.05	40	48	48	50

2.1.5.6 Concluding Remarks on the Recursive Subgoal Strategy

We have introduced a search strategy for planning problems with conjunctive goals and combine this search strategy with a novel GA-based planning algorithm. Our strategy transforms the goal of a planning problem into a sequence of recursive subgoals. As a result, the search for a complete solution consists of a number of independent stages. After reaching a subgoal, the problem is reduced to a similar problem but at a smaller scale. This strategy is applicable to a larger class of problems characterized by the fact that the construction of recursive subgoals guarantees the serializability of the subgoals. The experimental results on the Sliding-tile puzzle indicate that, although the recursive subgoal strategy may not find optimal solutions, it is able to achieve better search performance than the traditional single-goal planning approach and solve larger instances of problems than the existing domain-specific planning approaches. Additional experiments on genetic parameters reveal that the population size has much stronger influence on the performance of the search than crossover and mutation rates have. A large population improves the quality of search but it also results in higher computational cost.

Although in Section 2.1.5.2 we identify three classes of planning domains relative to the applicability of this strategy, a crisp criterion to decide if our strategy is applicable for a given problem proves to be a formidable task. It is also very difficult to define the concept of "similar" planning problems. Informally, we say that a 5×5 Sliding-tile puzzle is reduced to a 4×4 one and it is intuitively clear why these problems are similar, but formalizing this concept is very difficult.

2.2 Planning for Large-Scale Distributed Systems

We are interested in planning for large-scale distributed systems. Specifically, our work focuses on the development of the planning service for the middleware. We outline the role of a planning service for the middleware, formulate the problem of planning, present a genetic-based algorithm for the problem, and evaluate the performance of the algorithm on a real-world scientific computing domain.

2.2.1 Problem Formulation

Planning plays an important role in improving the intelligence of a large-scale distributed system. The role of a planning service is to automatically compose original plans for users' computing requests, and more often, for replanning to adapt an existing plan to new computing environments.

Before we formulate the problem of planning for large-scale distributed system, we need to first specify the input and output of planning, i.e., the case description and process description for a computation, respectively.

2.2.1.1 Process Description

A *process description*, also called a plan, is a formal description of the complex problem a user wishes to solve. For the rest of the dissertation, we use the terms "plan" and "process description" interchangeably.

A process description specifies all activities to be executed for a computation and the precedence relations among them. There are two types of activities defined in a process description: end-user activities and flow control activities.

Every *end-user activity* corresponds to an end-user computing service available to the middleware. Like the operators for traditional planning problems, every end-user activity has preconditions and postconditions. The *preconditions* of an activity specify the set of all conditions for the activity to be executed. The *postconditions* of an activity specify the set of conditions that must hold after the successful execution of the activity. An activity is *valid* only if all preconditions are satisfied before it is executed. A process description is *valid* only if all end-user activities that should be executed are valid.

Unlike end-user activities, *flow control activities* do not have associated computing services. They are used to control the execution of end-user activities in a process description. We define six flow control activities: `Begin, End, Choice, Fork, Join`, and `Merge`. This basic set of flow control activities is sufficient to specify a wide range of execution flow patterns including sequential, parallel, conditional, and iterative execution.

Every plan should start with a `Begin` activity and conclude with an `End` activity. These `Begin` and the `End` activities can occur only once in a plan.

The *direct precedence* relation reflects the causality among activities. If activity $\mathcal{B}$ can only be executed directly after the completion of activity $\mathcal{A}$, we

say that $\mathcal{A}$ is a *direct predecessor* activity of $\mathcal{B}$ and that $\mathcal{B}$ is a *direct successor* activity of $\mathcal{A}$. An activity may have multiple predecessor activities and multiple successor activities. We use the term "direct" rather than "immediate" to emphasize the fact that there may be a gap in time from the instance an activity terminates and the instance its direct successor activity is triggered. For the sake of brevity we drop the word "direct" and just use the term "predecessor activity" and "successor activity" to denote the precedence relations.

A `Choice` flow control activity has one predecessor activity and multiple successor activities. `Choice` can be executed only after its predecessor activity has been executed. Following the execution of a `Choice` activity, only *one* of its successor activities may be executed. There is a one to one mapping between the transitions connecting a `Choice` activity with its successor activities and a *condition set* that selects the unique activity from the successor activities that will actually gain control. Several semantics for this decision process are possible.

A `Fork` flow control activity has one predecessor activity and multiple successor activities. The difference between `Fork` and `Choice` is that after the execution of a `Fork` activity, all its successor activities are triggered.

A `Merge` flow control activity is paired with a `Choice` activity to support the conditional and iterative execution of activities in a plan. `Merge` has at least two predecessor activities and only one successor activity. A `Merge` activity is triggered after the completion of any of its predecessor activities.

A `Join` flow control activity is paired with a `Fork` activity to support concurrent activities in a plan. Like a `Merge` activity, a `Join` activity has multiple predecessor activities and only one successor activity. The difference is that a `Join` activity can be triggered only after all of its predecessor activities are completed.

Based on the above description, we give the BNF grammar for the process description used by the planning services. The symbol S denotes the start symbol.

```
S ::= <ProcessDescription>
<ProcessDescription> ::= BEGIN <Activities> END
<Activities> ::= <SequentialActivities> | <ConcurrentActivities>
                | <IterativeActivities> | <SelectiveActivities>
                | <Activity>
<SequentialActivities> ::= <Activities> ; <Activities>
<ConcurrentActivities> ::= FORK <Activities> ; <Activities> JOIN
<IterativeActivities> ::= ITERATIVE <ConditionalActivity>
<SelectiveActivities> ::= CHOICE <ConditionalActivity> ;
                          <ConditionalActivitySet> MERGE
<ConditionalActivitySet> ::= <ConditionalActivity>
              | <ConditionalActivity> ; <ConditionalActivitySet>
<ConditionalActivity> ::= { COND <Conditions> } { <Activities> }
<Activity> ::= <String>
<Conditions> ::= ( <Conditions> AND <Conditions> )
                 | ( <Conditions> OR <Conditions> )
                 | NOT <Conditions>
                 | <Condition>
<Condition> ::= <DataName>.<Attribute> <Operator> <Value>
<DataName> ::= <String>
<Attribute> ::= <String>
<Operator> ::= < | > | = | <= | >=
<Value> ::= <String>
<String> ::= <Character> <String> | <Character>
<Character> ::= <Letter> | <Digit>
<Letter> ::= a | b | ... | z | A | B | ... | Z
```

```
<Digit> ::= 0 | 1 | ... | 9
```

Figure 2.24 in Section 2.2.3.1 gives an example process description. The nodes represent activities; the arrows represent precedence relations among the activities. This process description consists of seven end-user activities and six flow control activities. The pair of `Choice` and `Merge` activities specifies that all activities bounded by the two activities should be executed iteratively until a pre-specified condition associated with the `Choice` activity is satisfied. The pair of `Fork` and `Join` activities specify that the three activities, "P3DR2", "P3DR3", and "P3DR4", are independent to each other and thus can be executed in parallel.

2.2.1.2 Case Description

Along with a process description, a computing task should also include a case description. A case description provides additional information for a particular instance of the task execution that a user wishes to perform, e.g., it provides the location of the actual data for the computation, additional constraints related to security, cost, or the quality of the solution, a soft deadline, and/or user preferences [2]. A computing task corresponding to a process description may be executed many times, thus multiple case descriptions related to a single process description typically exist.

2.2.1.3 A Planning Service

A planning service is one of the core services in the middleware. The function of a planning service is to generate valid process descriptions, or plans, to satisfy

users' computing needs. For the rest of the dissertation, we use the terms *plan* and *process description* with essentially the same meaning.

A planning service accepts planning requests from a coordination service in the middleware. The planning requests are part of the case description given by users for specifying the conditions, constraints, and preferences for executing a computing task. The assignment received by the planning service includes: 1) the set of the initial data available to the end user, 2) the goal of planning, which is often expressed in terms of the results of computations expected by the end user, and 3) other useful information. Once a process description is created, the planning service sends it to the coordination service and possibly archives it in the knowledge base. Figure 2.12(a) shows the exchange of messages between the planning service and the coordination service for a standard planning request.

In addition to ab-initio generation of valid process descriptions, the planning service is involved in replanning. *Replanning* is triggered by the coordination service whenever the state of the environment is such that the execution of a valid process description cannot continue. When replanning is required, the coordination service sends to the planning service all available data, including the initial set of data and the data modified or created during the execution of the computing task.

Conceptually, replanning has the same attributes as planning, but with one major difference: during replanning, the planning service has to improve the robustness of plans. To achieve this goal, the planning service needs to interact with the runtime environment and avoid reusing in a new plan those activities that prevent the previous plan from successful execution. In other words, the planning service should know whether an activity used in a new plan is executable or not. There are two possible methods of acquiring this knowledge. With the

first method, the knowledge is given directly by the coordination service. This knowledge acquired by the coordination service may be very incomplete. As the coordination service only knows which activity in the process description fails in execution, the planning service might still not able to know whether the adoption of other activities may succeed. With the second method, the planning service gets support from the other services in the middleware. This method consists of three steps. First, the planning service asks the information service for a brokerage service that is available in the system. Second, the planning service contacts with the brokerage service to get a group of end-user services that can possibly provide the execution of the activity. Third, the planning service communicate with end-user service for the availability of execution of this activity. An activity can be included in the new plan only if there is at least one service that can provide the execution of the activity. Replanning, however, does not fully guarantee the success of the plan execution because the state of resources needed by various activities typically changes frequently. A valid activity during the process of replanning may fail during executing. Figure 2.12(b) shows the flow of communications between the planning service and other services during replanning.

2.2.1.4 Formulation of Planning

An essential part of formulating the problem of planning for large-scale distributed system is to relate this problem to the traditional AI planning problems, i.e., to relate end-user services to operators, and relate the case description of a computation to the initial state and the goal of planning. The method of formulating the problem is based on the ontologies defined for the middleware.

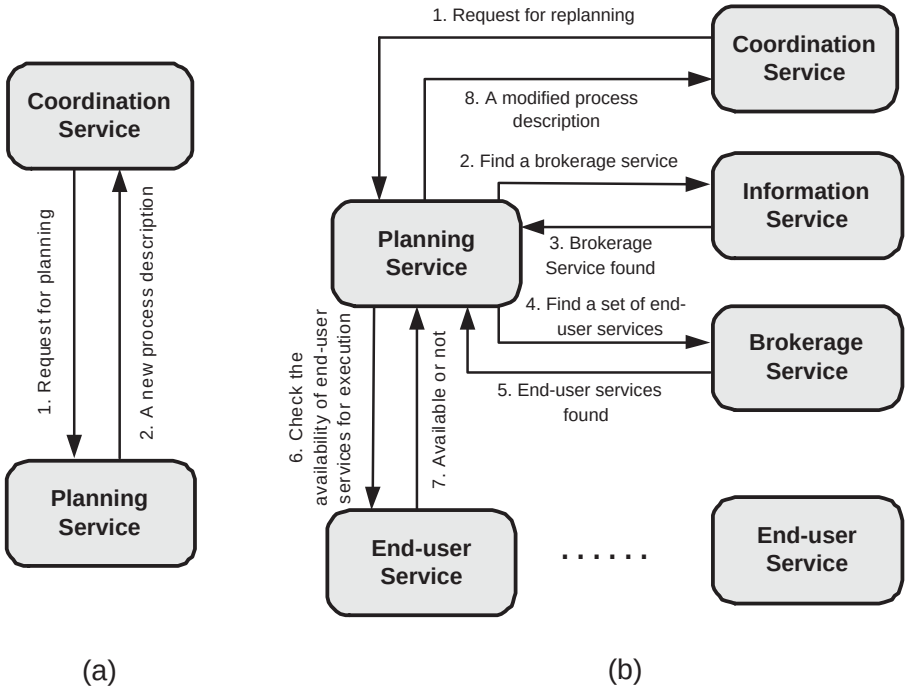


Figure 2.12: The interactions between the planning service and other services during (a) planning, and (b) replanning.

The ontologies shown in Chapter 1.3.3 consist of a number of classes. Each class corresponds to one type of entity defined for a large-scale distributed system. There may exist semantic correlations among classes and the correlation is represented by the arrows shown in Figure 1.3. For instance, a complete computing task is given by a process description along with a case description for each instance of its execution. The class "Task", therefore, has two slots that link a computing task to instances for classes "Process Description" and "Case Description", respectively. By first accessing the instance for class "Task" and then referencing the instances for classes "Process Description" and "Case Description", we are able to retrieve the complete knowledge (including its process description and case descriptions) for a given computing task.

Based on the structure of the ontology, we are able to build the knowledge for a computation onto multiple levels. Using the same example of a computing task for illustration, we can construct the knowledge related to a computing task on two levels. The primary level contains the knowledge directly retrieved from the instance in class "Task" (which contains the most general information for the computing task such as the id and the initiator of the task). The secondary level stores the knowledge retrieved from the corresponding instances in classes "Process Description" and "Case Description". These instances provides supplementary information (e.g., the process description and the case description) for a given task.

The above method of knowledge building can be applied to any other classes in the ontology, including both the basic classes shown in Figure 1.3 and the extended classes for a specific computing domain. Figure 2.13 shows the basic structure of ontology for storage and access of knowledge for a two-dimensional image file for viruses. The knowledge can be constructed on three levels shown in Figure 2.14. The lower the level in a hierarchy, the more specific the knowledge is related to an image file.

Using the above method, we can not only build the initial set of data for a computation, but also specify the goals for a computation. The only difference is that conditional expressions are used to specify the goals. For instance, if the goal of a computation is to create a 3D structure of a virus whose must meet a pre-specified resolution (e.g., 8.0), we can express the goals on different levels based on the ontology structures. Figure 2.15 shows the basic structure of ontology for a 3D virus structure. Figure 2.16 shows the expression of the goals of a computation onto multiple levels based on the ontology structure.

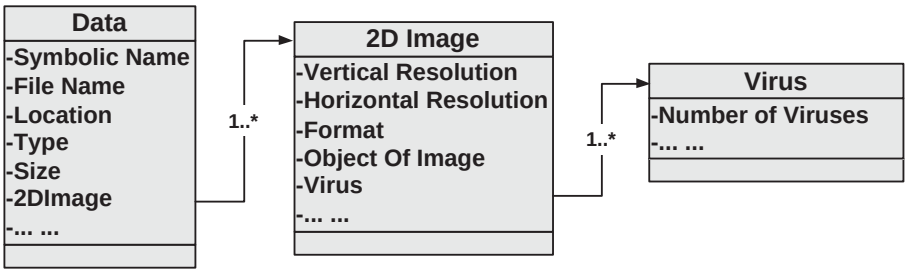


Figure 2.13: The ontology that supports the storage and access of knowledge related to a two-dimensional image file for viruses. Class "Data" stores the general information for a data file; class "2D image" stores information related to a two dimensional image; and class "virus" stores the information related to the object of the image: viruses.

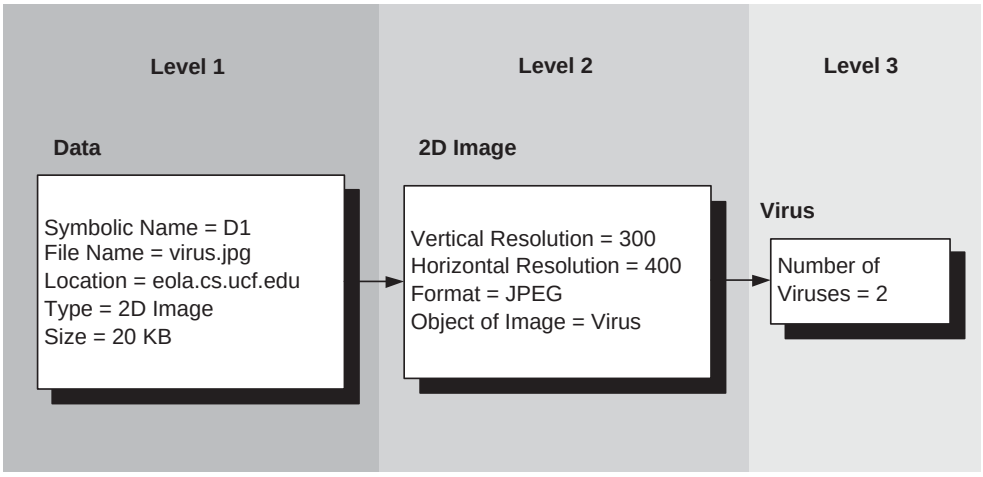


Figure 2.14: The three-level structure of knowledge related to a two-dimensional image file of viruses. Each level of knowledge is retrieved from instances stored in the corresponding classes for the ontology.

Likewise, we can also apply the same method to construct the knowledge for all end-user services available in a middleware. End-user services are the counterpart of operators in the traditional AI planning. Like an operator, an end-user service has three attributes: preconditions, postconditions, and cost.

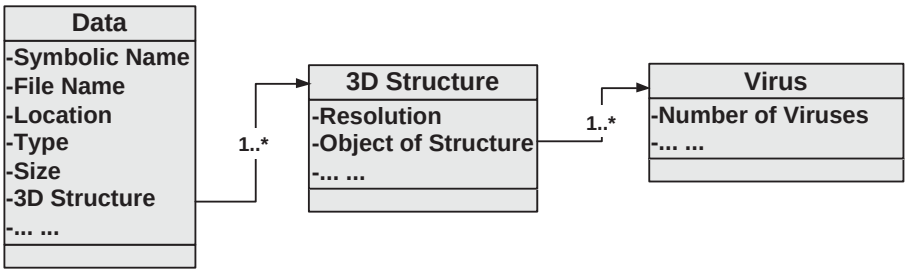


Figure 2.15: The ontology that supports the storage and access of knowledge related to 3D structure of a virus. Class "Data" stores general information for a data file; class "3D structure" stores knowledge related to the attributes of a 3D structure; and class "virus" stores the information related to the object of the structure: viruses.

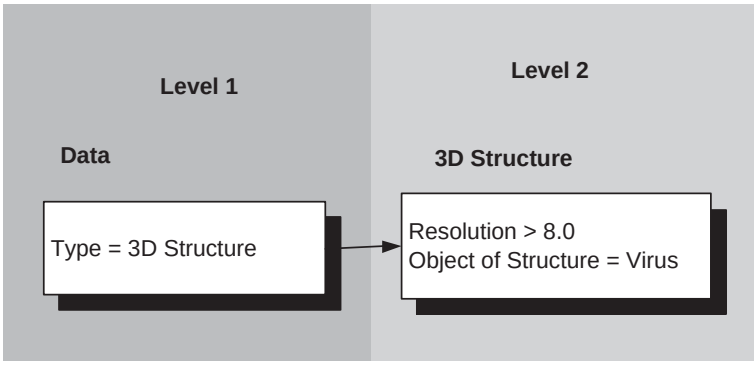


Figure 2.16: The multi-level structure of goal conditions for a computation, which is to create a 3D virus structure whose resolution should be greater than a specified value (in this case 8.0).

The preconditions of a service specifies all conditions that must be satisfied for the service to be executed. Each condition in the preconditions has the format: "CLASS.ATTRIBUTE OPERATOR VALUE", where "CLASS" refers to one of the classes in the ontology, "ATTRIBUTE" represents one of the attributes for the class, "OPERATOR" is a conditional operator such as "$\leq$" and "=", and "VALUE" can be given by a constant, a set of discrete constants, or a range of

continuous values (specified with two constants: a lower bound and an upper bound).

The postconditions of a service specify the results of the computation, i.e., all data that are modified or created after the execution of the service. Each condition in the set of postconditions has the format: "CLASS.ATTRIBUTE = VALUE", where "CLASS" specifies one of the classes in the ontology, "ATTRIBUTE" represents an attribute for the class, and "VALUE" can be a constant, a set of discrete constants, or a range of continuous values. Unlike preconditions, only "=" can be used as an operator in postconditions, denoting that the value is assigned to the attribute after the service is executed.

The cost of a service is used to quantify the computational cost of executing the service. The computational cost can be measured in multiple aspects such as the amount of resources requested and the computation time. The computational cost of a service may vary over multiple invocations of the same service as it may heavily depend on the size of the input data or some environmental factors. Therefore, the cost of end-user service is typically very difficult to quantify and is considered as an optional attribute for an end-user service.

All knowledge related to the preconditions and postconditions for an end-user service can be stored on multiple levels based on the ontology structure. Figure 2.17 shows the definition of end-user service "P3DR" whose preconditions and postconditions are defined on multiple levels of a hierarchical structure.

With a hierarchical structure to define initial set of data, goal conditions, and end-user services, we can formulate the problem of planning for a large-scale distributed problem using a multi-level structure. For a given computation, we can formulate different planning problems depending on the specificity of knowledge embedded for planning. If we do not apply domain specific knowledge

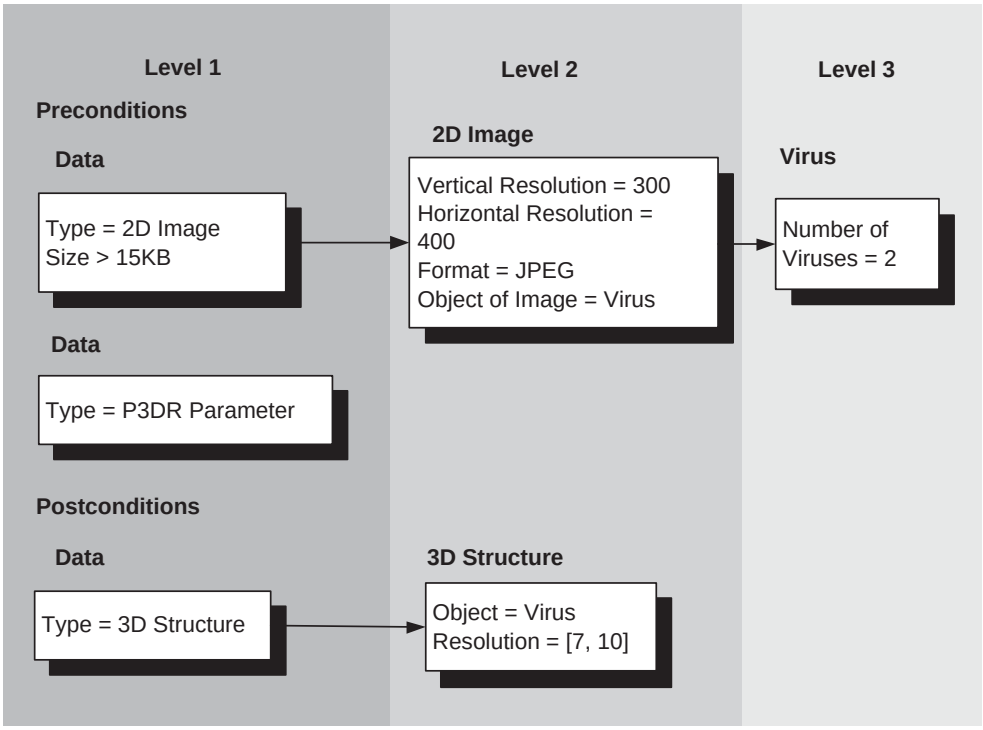


Figure 2.17: A graphical representation of the definition of end-user service "P3DR". The preconditions and postconditions of this service are defined on multiple levels of a hierarchical structure. The function of the service is to build a 3D structure of a virus from a group of its 2D images. The computation of the service requires a group of virus images and a parameter file that is used to control the process of the computation. The output of the computation is a file that stores the 3D structure of the virus.

for planning, we do not need to use all knowledge for planning. Generally, the more specific knowledge used for planning, the more levels of knowledge that must be embedded for the formulation of planning. For instance, if we take knowledge represented in Figures 2.14, 2.16, and 2.17 as the initial data, the goal conditions, and the available end-user service for a computation, we can formulate three different planning problems: the first problem only uses the knowledge residing

on the first level, the second problem uses knowledge on the top two levels, and the third one uses the entire knowledge for the computation.

2.2.1.5 Classifications of Planning Problems

The problem of planning can be classified with multiple aspects. In this study, we are interested in the classification on two aspects.

a) *Planning vs. Replanning*

A typical AI planning problem assumes a static environment in which the initial state and the set of available operators do not change during the course of planning. Planning for a large-scale distributed system may also ignore the current status of the environment and assume that the resources required by a computation can always be satisfied. A distributed system, however, is a dynamic computing environment. New services and resources may be supplied to the system at any time; the existing ones may become unavailable. The change of the computing environment, in many cases, is unpredictable. As a result, a valid plan may not succeed in execution when certain resources or computing services become unavailable. Replanning is a process that aims to improve the success in plan execution by including the knowledge related to the run-time environment during planning. Replanning incorporates up-to-date knowledge regarding the current status of the distributed system such as the availability resources for the computation. Replanning, however, cannot fully guarantee the success of plan execution as the status of the computing environment may change very frequently. The knowledge related to the run-time environment available to a planning service may become obsolete when a plan is executed.

b) *Deterministic vs. Non-deterministic Planning*

A planning problem is deterministic if the system state can be completely determined on all stages of planning (i.e., we have the complete knowledge regarding the initial state of the system and the outcome of all operators can be fully determined). The Sliding-tile puzzles, for instance, is a deterministic problem as the initial configuration of the tiles and the postconditions of all four operators are known to the planner. The solution to a deterministic planning problem typically has a linear structure in which all operators in a plan are executed sequentially.

Recent study in AI planning has paid special attention to planning in non-deterministic domains. A planning domain is non-deterministic if either the initial state and/or the outcome of some operators cannot be fully determined at planning time. The goal of non-deterministic planning is to find a plan that can reach the goal in spite of the non-determinism of the domain.

Non-deterministic planning can be classified into different problems based on the source of uncertainty and on whether this uncertainty persists during plan execution. The term "Conformant Planning" refers to the class of problems in which the initial state is not completely known and there is no sensor action allowed to acquire the complete knowledge of the system state during plan execution [46]. "Contingent Planning", on the other hand, allows the use of certain sensor actions to detect system state so that a plan can react to different sensor results at execution time [47]. In the third class of non-deterministic planning problems, we have the complete information of the initial state of the system, while there exists some operators that have several possible outcomes.

The notion of "weak planning", "strong planning", and "strong cyclic planning" are introduced in [48] to classify different non-deterministic planning problems based on the quality of solutions. Solutions to week planning have a non-zero

possibility of reaching the goal, but it cannot guarantee that the goal can always be reached; solutions to strong planning are guaranteed to reach the goal; and solutions to strong cyclic planning is able to reach the goal with a finite number of operators, but the number of operators to be performed cannot be determined at planning time. Iterative execution of operators may be unavoidable before some desirable conditions are satisfied during plan execution.

The planning problem that we formulate for large-scale distributed systems can be non-deterministic as we may not have the complete knowledge of the results of some end-user services. For instance, we may only know the type of the data to be produced by a service, but we may not know the other attributes of the data (e.g., the size of the data), and this type of knowledge may be important in checking the validity of subsequent services in a plan. Hence, the uncertainty of service execution prevents us from evaluating the validity of services in a plan, and as a result, makes it very difficult to evaluate the validity of solutions. Based on the assumption that we always have the complete knowledge of the initial state of system at planning time, our planning problem belongs to the third class of non-deterministic problems. We allow iterative execution of services in a solution to deal with the non-determinism resulting from executing end-user services.

2.2.2 A Genetic-Based Approach for Non-deterministic Planning

This section presents a genetic-based approach to the formulated planning problems. This approach is extended from the GA-based planning approach discussed

in Section 2.1.3. A distinct feature of this approach is to allow the evolution of a non-linear structure for plans to deal with non-determinism in planning domains.

The following sections discusses the important features of this approach, including the internal representation of plans, the solution initialization, solution evaluation, and the genetic operators.

2.2.2.1 Solution Encoding

A simple genetic algorithm uses a linear binary string to encode candidate solutions. As we use a non-linear structure for the process description, we must use a non-linear representation scheme to encode solutions, accordingly.

In this approach, we use a tree structure to represent and evolve process descriptions. The tree representation has been widely used in genetic programming to evolve solutions that achieve the desired functions [13]. Therefore, this approach is more GP-based than GA-based due to the representation scheme used.

A plan tree consists of a set of nodes. The nodes can be either terminal nodes or controller nodes. Every *terminal node* is a leaf in a plan tree corresponding to an end-user activity in the process description. On the other hand, *controller nodes* are the internal nodes and must have at least one child node in a plan tree. Controller nodes are used to direct the plan execution, and thus have similar functions to the flow control activities in a process description. However, there does not exist a one-to-one correspondence between controller nodes in the plan and flow control activities in the process description. We now provide some details of the semantics of each type of controller nodes and show how they are correlated to the flow control activities.

We define four types of controller nodes: sequential, concurrent, selective, and iterative.

1. A *sequential node* requires that all activities corresponding to its children be performed sequentially. The sequence of their execution is specified by the relative location of each node among its siblings. Activities are executed from left to right. The leftmost child of a sequential node is executed first; the rightmost child of a sequential node is executed last. Only when the activity of its rightmost child completes, the block controlled by the sequential node terminates and the flow control is transferred to the next control structure.

   A sequential node does not have a corresponding flow control activity in a process description. As the arrows in a process description specify the sequence of activity execution, we can convert a sequence of activities in a process description into a tree structure with the sequential node as the root. Figure 2.18 gives an example of such conversion.

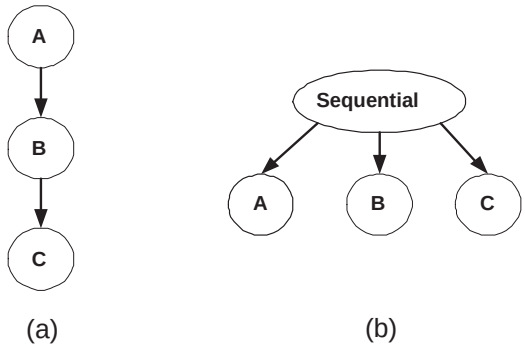


Figure 2.18: Process description versus plan tree for sequential activities. (a) a partial process description consisting of a sequence of activities; (b) the corresponding plan tree with the sequential node as the root node.

2. A *concurrent node* informs the environment that all activities that correspond to its children can be executed either sequentially or concurrently. If the activities are executed sequentially, they can be executed in any order. Only after all these activities are executed can the execution of the concurrent block of activities be completed.

   Each concurrent node corresponds to a pair of `Fork` and `Join` activities. Figure 2.19 gives an example of a partial process description with concurrent execution of activities and the corresponding plan tree.

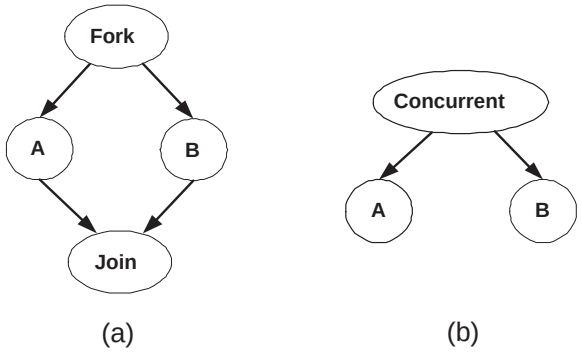


Figure 2.19: Process description versus plan tree for concurrent activities. (a) a partial process description consisting of a set of concurrent activities; (b) the corresponding plan tree with the concurrent node as the root node.

3. A *selective node* informs the environment that only one of the activities corresponding to its children have to be executed. The execution of a selective block can be finished as long as one of the activities is executed.

   Each selective node corresponds to a pair of `Choice` and `Merge` activities. Figure 2.20 gives an example of a partial process description with selective execution of activities and the corresponding plan tree.

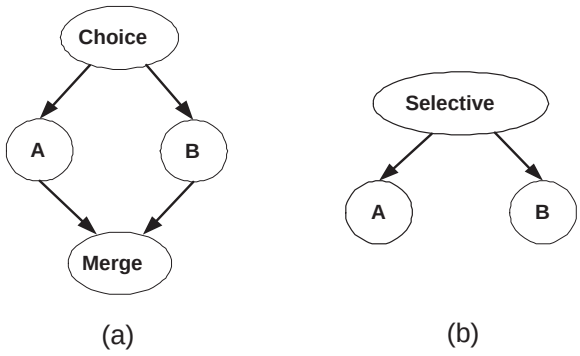


Figure 2.20: Process description versus plan tree for selective activities. (a) a partial process description consisting of a set of selectively executed activities; (b) the corresponding plan tree with the selective node as the root node.

4. An *iterative node* requires that all activities that correspond to its child nodes must be executed iteratively until some stopping conditions are met.

   Each iterative node corresponds to a loop in a process description. A loop is formed where a transition in a process description terminates at an activity that has been executed before the activity as the source of the transition. When we convert from a process description to a plan tree, we insert all nodes within a loop as the children of the iterative node. The sequence of children follows the execution order of the activities in the loop. Figure 2.21 gives an example of this conversion.

When we convert a complete process description to a plan tree, the above methods of conversion should be applied recursively, in a top-down manner, until a complete plan tree is generated. We can also use the similar method to convert a plan tree to a process description.

The *size* of a plan tree is defined as the number of nodes in the tree. We set an upper limitation, S_{max}, to the size of plan trees during the evolution of solutions. The purpose of setting a limitation is to prevent the unlimited growth

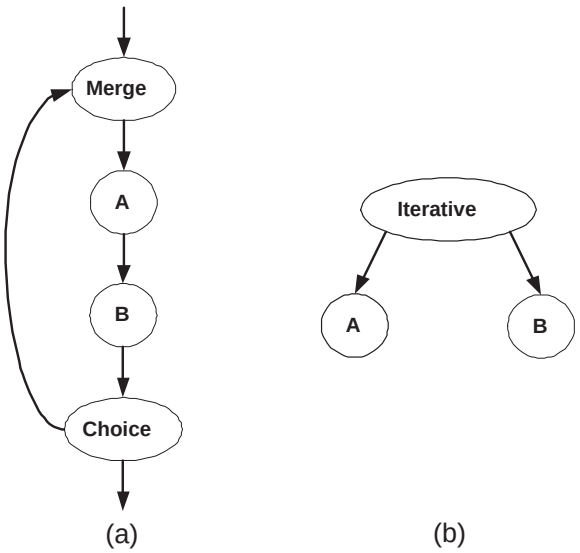


Figure 2.21: Process description versus plan tree for iterative activities. (a) a partial process description consisting of a set of iteratively executed activities; (b) the corresponding plan tree with the iterative node as the root node.

of trees, also called "bloat", a commonly observed problem in GP [49]. The value of S_{max} should be set properly to ensure the efficiency of the search without compromising the quality of solutions.

2.2.2.2 Solution Initialization

In the initialization phase, we randomly generate a population of trees as candidate solutions. These trees may not encode valid solutions for a given problem, but they must conform to regulations of the tree structure defined in Section 2.2.2.1. The size of every initial tree cannot exceed S_{max}, the upper bound for the tree size.

The initialization of each plan tree consists of two steps. In the first step, we generate an arbitrary tree structure of a given size. The size of an initial tree is randomly chosen between one and S_{max}. In the second step, we instantiate each node in the tree. Every internal node is instantiated with a controller node that is randomly selected from the four controller nodes. Every terminal node is instantiated with an end-user activity that is randomly selected from the set of end-user activities available to the middleware.

2.2.2.3 Plan Evaluation

The fitness of a plan gives a rough evaluation of the quality of a plan. The evaluation of a plan consists of three independent aspects: the validity of plan, the result of plan execution against goals, and the efficiency on plan representation.

1. Plan validity fitness f_v: can every activity in a plan be executed?

   An activity can be executed only if all its preconditions meet the system state right before the execution of the activity. To evaluate the plan validity fitness, we need to simulate the execution of a plan and go through each activity in a plan.

   During the simulation process, we follow the sequence of execution of activities and verify their validity. The initial state of the system is given by the set of all initial data and their attributes. For each activity, we check if the current system state satisfies all preconditions of the activity. There are three possible results for checking the validity of an activity: valid, invalid, and undetermined. a) An activity is valid if the current system state satisfies every precondition of the activity. If an activity is valid, we

update the system state to the one after this activity is executed. The new system state will include all new and modified data resulting from the execution of the activity; b) An activity is invalid if there exists at least one precondition such that the current state cannot satisfy. If an activity is invalid, we do not update the system state and proceed to check subsequent activities; c) If the validity of an activity cannot be determined due to the uncertainty of the system state, we mark the result of the validity check as "undetermined" and update the system state the same way as for a valid activity.

This process of validity check is continued until we have finished checking all activities in a plan. In case there is `Choice` in a plan and we cannot determine which successor activity to perform next, we need to enumerate each possible flow of execution and simulate the execution of a plan multiple times. If the execution of a single activity is simulated multiple times, each instance of its execution is counted in the validity check. We assign full credits to valid activities and partial credits to activities whose validity cannot be determined at planning time. The fitness of plan validity is determined by the proportion of these two types of activities over all activities in a plan. The fitness of plan validity can be calculated with the following equation:

$$f_v = \frac{number\ of\ valid\ activities\ + 0.5 \times number\ of\ undetermined\ activities}{total\ number\ of\ activities\ that\ are\ executed} \tag{2.5}$$

2. Goal fitness f_g: How does the execution of a complete plan reach the goal specifications of the planning problem?

After finishing the execution of a complete plan, we reach the final state. The evaluation of goal fitness, as in traditional AI planning domains, is usually problem specific, meaning the "closeness" between the final state and the goal specifications is largely dependent on the characteristics of the computing task. Generally speaking, a "closer" match between the final state and the goal specifications results in a higher goal fitness. The following equation shows a simple goal fitness function in which we assume that each goal specification has equal weight in evaluating the overall goal fitness. As the final state of a plan execution may not be fully determined at planning time, we may not be able to fully determine whether a goal can be satisfied or not. We assign partial credits to these goals in the fitness function.

$$f_g = \frac{number\ of\ satisfied\ goals\ + 0.5 \times number\ of\ undertermined\ goals}{total\ number\ of\ goals\ specified\ in\ the\ problem} \tag{2.6}$$

If a plan is simulated multiple times due to the conditional execution of some activities, the goal fitness is given as average goal fitness of all instances of execution.

3. The efficiency of plan representation f_r.

   The efficiency of a plan representation is determined by the number of nodes (including both activity nodes and controller nodes) in a plan tree. We use the following equation to calculate this fitness.

$$f_r = 1 - \frac{number\ of\ nodes\ in\ a\ plan\ tree}{S_{max}} \tag{2.7}$$

   Clearly, $0 \leq f_r < 1$. A smaller plan tree receives a higher f_r.

The overall fitness of a plan is the weighted sum of all three aspects of fitness.

$$f = w_v \times f_v + w_g \times f_g + w_r \times f_r, \tag{2.8}$$

where w_v, w_g, and w_r are the weights of the three aspects of fitness, respectively, and

$$w_v + w_g + w_r = 1 \tag{2.9}$$

2.2.2.4 Genetic Operators

Genetic operators are the driving forces to push the evolution forward. The operators we use include crossover and mutation.

Crossover takes place between a pair of plan trees. We apply a crossover method that is commonly used in GP. This method consists of four steps. First, we select two trees as parents and decide if they can be crossed over. The probability of two trees taking part in crossover is determined by a parameter called crossover rate. Second, if the two trees are not crossed over, we keep them and terminate the crossover process. Otherwise, we randomly select a node from each parent. Third, we switch the subtrees associated with the selected nodes between the two parents. As a result, we create two new plan trees, each containing partial plans from both parents. Finally, we replace the parents with the new trees. In case the size of a new tree exceeds S_{max}, crossover fails and both parents are kept. Figure 2.22 shows a simple example of how the crossover works on plan trees.

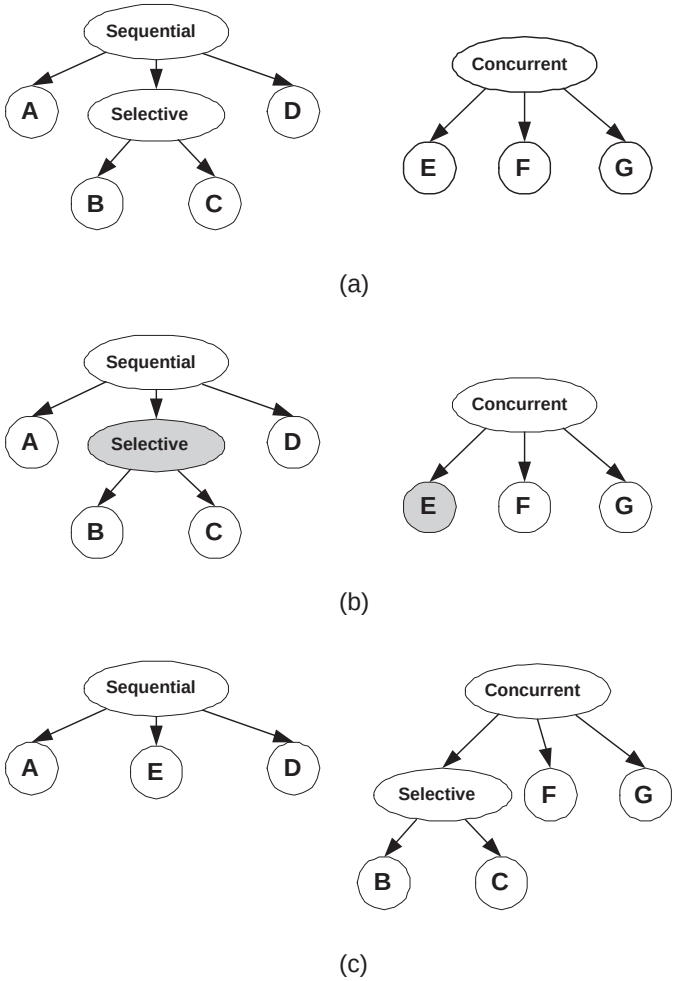


Figure 2.22: An example of crossover performed on two plan trees. (a) two original trees are selected as parents; (b) a node is selected from each parent; (c) two new plan trees are created by switching the subtrees associated with the selected nodes.

Each *Mutation* consists of three steps. First, we randomly select a node in the tree to be mutated. The probability of a node being selected is determined by a parameter called "mutation rate". Second, we randomly generate a tree, using the same method as plan initialization. Third, we replace the subtree associated with the selected node with the randomly generated tree. If, however,

the new tree exceeds the size limitation, mutation fails and we keep the original tree. Figure 2.23 illustrates a simple example of mutation on a plan tree. Node "Selective" is selected and the subtree associated with the node is replaced by a randomly generated tree.

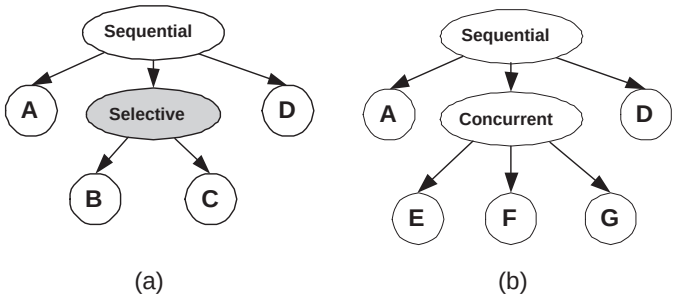


Figure 2.23: An example of mutation performed on a plan tree. (a) a node is selected to be mutated; (b) the subtree associated with the selected node is replaced by a randomly generated tree.

2.2.3 Performance Study

We conduct two experiments to study the performance of the planning approach. In the first experiment, we apply the planning approach to finding execution plans for a real-world computing problem: the 3D reconstruction of virus structure. We next evaluate the scalability of this approach in a simulation environment.

2.2.3.1 A Case Study on 3D reconstruction of virus structure

We use the computation for 3D reconstruction of virus structure in electron microscopy as the test case for evaluating the performance of the planning approach.

Given a set of 2D images of a virus, and an initial model of the electron density map, the goal of the computation is to construct a 3D model of the virus at the finest possible resolution given the physical limitations of the experimental instrumentation. Once we have a detailed electron density map of the virus structure we can proceed to atomic level modeling, namely placing of groups of atoms, secondary, tertiary, or quaternary structures on the electron density maps.

The computation is composed of several steps [4]. In the first step, we extract the 2D virus projections from the micrographs. Then we determine the initial orientation of individual views using an "ab initio" orientation determination program (POD). Next, we execute an iterative computation consisting of 3D reconstruction followed by orientation refinement. The parallel program used for reconstruction is called P3DR and the parallel program for orientation refinement is called POR. The iterative process stops whenever no further improvement of the electron density map at that resolution is noticeable. Then we use a correlation procedure to determine the resolution of the electron density map. The parallel program used for correlation is called PSF. The iterative computation is then repeated at a higher resolution, possibly discarding some of the input data that do not correlate well with the rest. A new approach is now implemented; we create two streams of input data, e.g., by assigning odd numbered virus projections to one stream and even numbered virus projections to the second stream. Then we construct two models of the 3D electron density maps and determine the resolution by correlating the two models. The process description, shown in Figure 2.24, consists of seven end-user activities and six flow control activities. The pair of `Choice` and `Merge` activities in this workflow is used to control the iterative execution for resolution refinement. The computation ends when the resolution is better than the one specified as computation goal. Figure 2.25 shows the corresponding plan tree.

Table 2.12 shows the parameter settings used in the experiment. We test the algorithm ten times and select the individual with the highest fitness in the final generation as the solution. Then we calculate the average fitness, validity fitness, goal fitness, and the size of solutions over ten runs, shown in Table 2.13.

Table 2.12: Parameter Settings in the experiments.

Parameters	Values
Population Size	200
Number of Generations	20
Crossover Rate	0.7
Mutation Rate	0.001
S_{max}	40
w_v	0.2
w_g	0.5

Table 2.13: Experiment results collected from the best solutions of ten runs.

Performance Criteria	Values
Average Fitness	0.928
Average Validity Fitness	1.0
Average Goal Fitness	1.0
Average Size of solutions	9.7

The experimental results show that this planning approach is able to consistently find valid execution plans that reach the goals of the computation. Although this approach cannot find the process description that perfectly matches the one shown in Figure 2.24, the best solution found can always reach 1.0 in

both the validity and goal fitness. The solutions found by the approach are also small, with the corresponding plan trees having an average of less than 10 nodes.

2.2.3.2 A Simulation Study

We next evaluate the scalability of this planning approach in a simulation environment. We test different cases by varying the number of available end-user services in the environment. Only a portion of the end-user services is needed to achieve the goal of the computing task. We test the case where the optimal solution contains 10 end-user services, and the environment contains 15, 20, 30, and 50 end-user services. A larger number of end-user services requires a search in a larger solution space. In the case with 15 end-user services, we set the population size to 200 and the number of generations to 500 (i.e., 100000 fitness evaluations are performed in each run). In larger problems, we increase the number of fitness evaluations allowed in a run with either a larger population, or more generations, or both. In cases with 20, 30, and 50 end-user services, the number of fitness evaluations that are allowed to perform in a run is 150000, 300000, and 800000, respectively. We test each case 50 times and report the average goal fitness of the best solutions and the execution time of each run (with 95% confidence intervals). Table 2.14 lists the parameter settings for this experiment. Table 2.15 shows the results.

The results show that this approach requires more fitness evaluations and thus longer execution time to maintain the quality of solutions to problems with larger search spaces. For instance, this approach reaches an average of 0.95 in the goal fitness in both cases with 15 and 20 end-user services. However, 50% more fitness evaluations are performed in the case with 20 end-user services than the one with

Table 2.14: Parameter Settings in the simulation study.

Parameters	Values
Population Size	from 200 to 1600
Number of Generations	from 500 to 4000
Crossover Rate	0.8
Mutation Rate	0.01
S_{max}	30
w_v	0.2
w_g	0.5
Size of Optimal Solutions	10
Number of Available Services	15, 20, 30, 50

15 end-user services. Comparing the experimental results on different parameter settings for the cases with 30 and 50 end-user services, we notice that having more generations in a run is generally more effective in improving the search quality than having a larger population. However, in the case with 50 end-user services, a population of 200 produces the lowest goal fitness of all cases. This result indicate that a larger population is still needed to provide sufficient resources for evolution in larger problems.

Table 2.15: The average goal fitness and execution time (in seconds) for different test cases. CI = confidence interval.

Number of End-user Services	Population Size / # of Generations	Average (95%CI) Goal Fitness	Average (95%CI) Execution Time
15	200 / 500	0.95 (0.034)	42.41 (3.26)
20	300 / 500	0.95 (0.033)	57.33 (4.28)
20	200 / 750	0.95 (0.033)	53.91 (4.75)
30	600 / 500	0.892 (0.063)	98.47 (10.03)
30	400 / 750	0.896 (0.056)	92.77 (8.06)
30	300 / 1000	0.900 (0.064)	91.86 (7.41)
30	200 / 1500	0.916 (0.055)	99.59 (9.59)
50	1600 / 500	0.922 (0.046)	249.04 (20.15)
50	800 / 1000	0.928 (0.015)	229.08 (22.70)
50	400 / 2000	0.942 (0.043)	219.75 (20.41)
50	200 / 4000	0.878 (0.069)	219.24 (26.33)

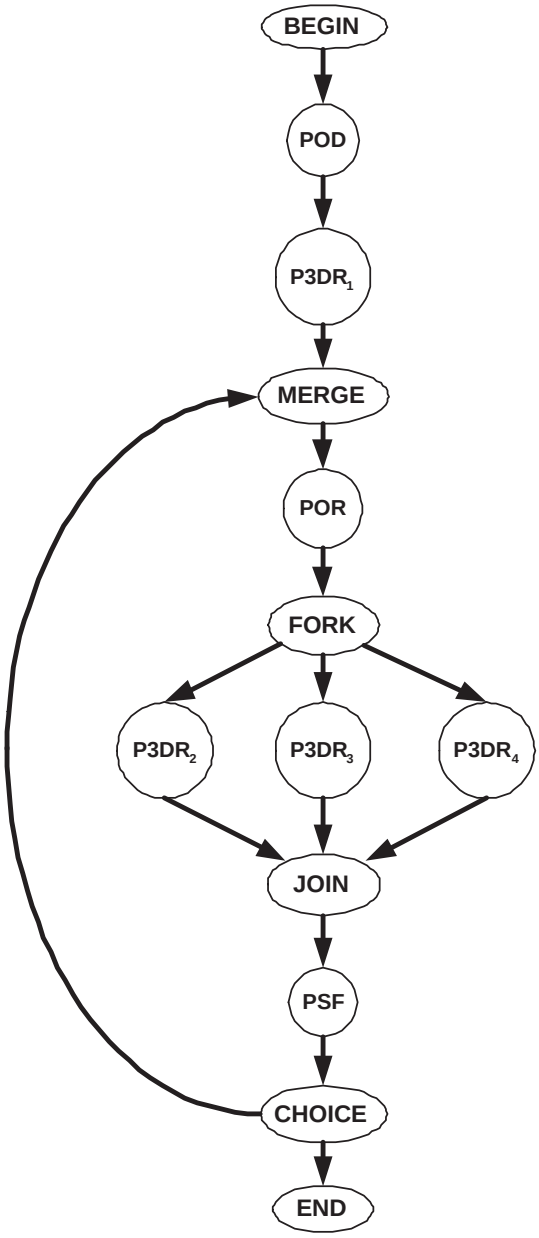


Figure 2.24: A process description for the 3D reconstruction of virus structures. POD - "ab initio" parallel orientation determination program. P3DR - the parallel program used for 3D reconstruction. POR - the parallel program for orientation refinement. PSF - parallel program to compute the correlation of the structure factors.

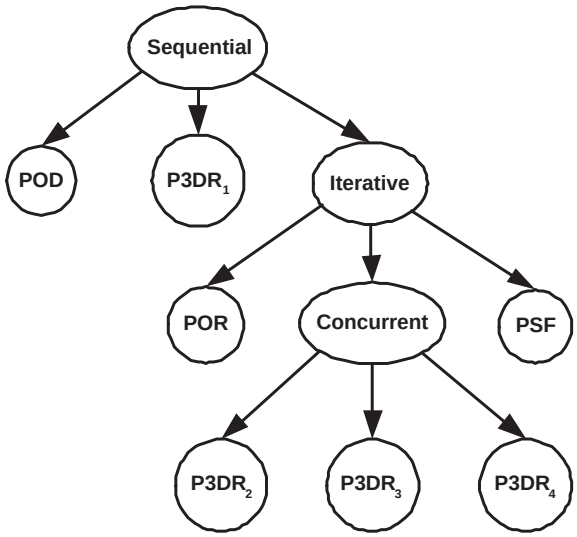


Figure 2.25: The corresponding plan tree to the process description for the 3D reconstruction of virus structures.

CHAPTER 3

SCHEDULING ALGORITHMS AND SCHEDULING SERVICES

The function of scheduling for a large-scale distributed system is to map the execution of each computing activity in a process description to an available computing node in the system and specify the time each activity is executed. The goal of scheduling is to optimize the total execution time of the computation (i.e., the duration between the time that the first activity is executed and the time the last activity finishes execution) while achieving load balance, ensuring a large throughput, or any other objects.

Scheduling for a large-scale distributed system bears much resemblance to the problem of multi-processor scheduling, but is much more difficult due to the scale and the complexity of the computing environment. In this chapter, we first discuss the problem of multi-processor scheduling and present a genetic algorithm approach to the problem. We next focus on the problem of scheduling for large-scale distributed systems. We formulate the problem and present an approach that is extended from the approach for multi-processor scheduling.

3.1 Introduction to Multi-processor Task Scheduling

Multi-processor task scheduling is an extensively studied problem in the field of parallel programming and processing. By decomposing a complete program into a set of smaller tasks, we are able to schedule these tasks on parallel processors and reduce the execution time. The procedure of multi-processor scheduling consists of several steps. First, a program to be executed is sent to a task decomposer that divides the program into a group of smaller computing tasks. The result of decomposition is a task graph that specifies the tasks to be scheduled. Next, the task graph is sent to a scheduler for an optimized schedule for task execution. Finally, the execution schedule is sent to the administrator of the processors. The administrator is responsible for assigning the task execution to each processor based on the schedule. Figure 3.1 gives a graphical illustration of these steps. Our work is focused on the second step of the procedure, i.e., producing an optimized schedule for task execution in a multi-processor system.

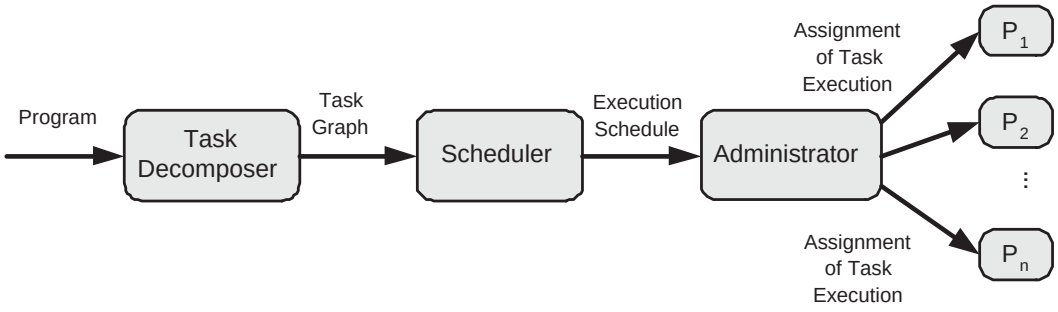


Figure 3.1: The procedure of scheduling the execution of a program in a multi-processor system.

The goal of multi-processor scheduling is to assign tasks to a number of processors in such a way that minimizes the total execution time of the program, also called *makespan*. Traditional multi-processor scheduling assumes that all proces-

sors have the same computing ability, i.e. a single task has the same execution time on all processors, and that all processors are fully connected by communication links with the same capacity. In addition, we assume that tasks are scheduled to a *static* computing environment. Both the topology (or interconnection) of the processors and their processing abilities do not change during scheduling. Tasks may have data dependencies, which raise additional precedence restrictions that the scheduler must follow in order to generate valid schedules. If two dependent tasks are assigned to different processors, a pre-specified amount of communication cost for data transfer between different processors must be applied. An optimal schedule should meet the following three criteria: 1) the order of task execution abides by the precedence restrictions of the tasks; 2) every task must be assigned to at least one processor; and 3) the makespan cannot be further reduced.

The multi-processor scheduling problem is given by a *Directed Acyclic Graph* (DAG). A DAG specifies the tasks that are to be scheduled, represented by nodes, and the data dependencies between the tasks, represented by arrows. The direction of the arrows indicates the direction of the precedence between tasks. In addition, the execution cost of each task and the communication cost between each pair of dependent tasks are given in a DAG. Figure 3.2 shows the DAG for the 14-node LU Decomposition task scheduling problem. The numbers beside the nodes specify the execution costs of the tasks. The numbers beside the arrows specify the communication costs between dependent tasks. Figure 3.3 shows an example schedule on four processors.

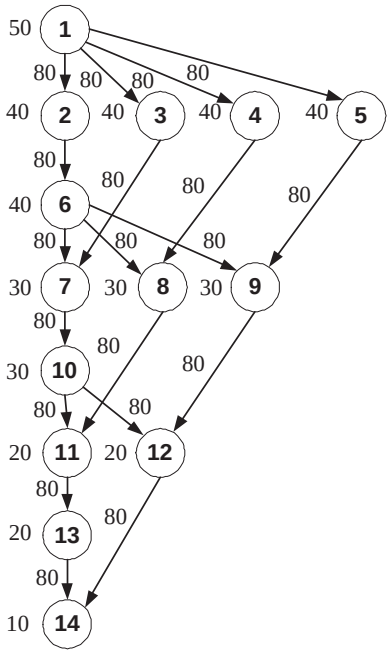


Figure 3.2: The DAG for the 14-node LU Decomposition task scheduling problem.

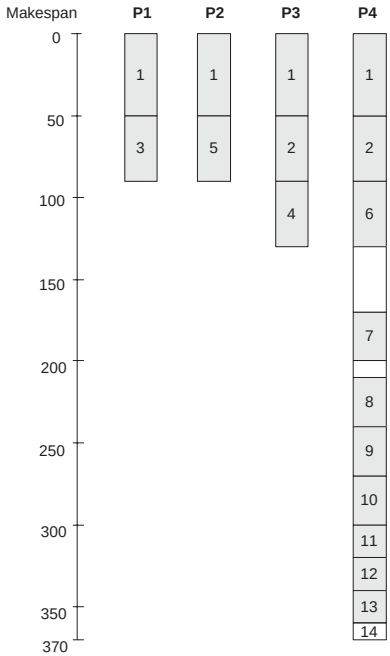


Figure 3.3: An example schedule for the 14-node LU Decomposition task scheduling problem on four processors.

3.2 Introduction to Scheduling for a Large-scale Distributed System

In a large-scale distributed system, a computing task is given by its processor description and case description. The process description specifies the workflow of the computation and contains a set of computing activities to be scheduled. The case description provides additional information related to the execution of the computation, e.g., the location of the input data, the deadline for finishing the execution, etc. Both the process and case descriptions are sent to the scheduling service for a valid schedule of the computation. The role of a scheduling service is to map the execution of the computing activities to the processors in a system and specify the time for their execution. After a schedule is produced, resource requests for execution are sent to the administrators that manage the processors on which the computation is scheduled. Multiple administrators may exist in a large-scale distributed system. Consequently, a resource request is sent to each administrator. A schedule is successful only if all resource requests are satisfied. Figure 3.4 shows the model of scheduling a computation in a large-scale distributed system.

There are noticeable similarities between multi-processor scheduling and scheduling for a large-scale distributed system. For instance, the goal in both problems is to assign the execution of a set of tasks to computing nodes to reduce the total execution time. Tasks in both problems may have data dependencies and the order of their execution must conform to the precedence restrictions. There are, however, major differences between these two problems and are listed as follows.

First, the granularity of the tasks to be scheduled is different for the two problems. Multi-processor task scheduling is a fine-grained scheduling problem.

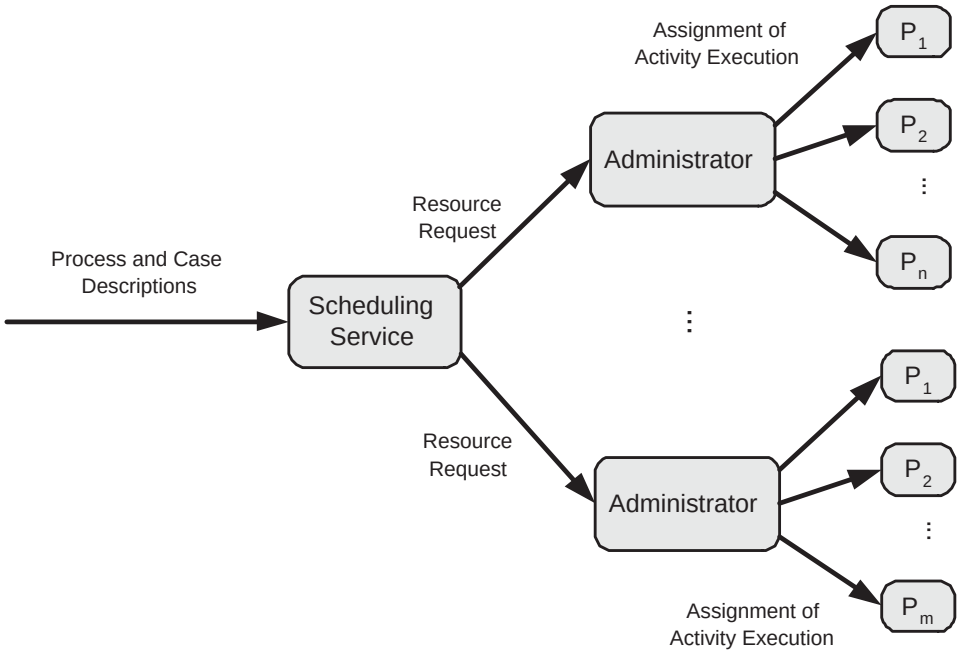


Figure 3.4: The model of scheduling a computation in a large-scale distributed system.

Tasks to be scheduled typically represent functions in a program to be executed and their execution time is typically short. In contrast, scheduling for a large-scale distributed system is a coarse-grained scheduling problem. The computing activities to be executed may require a large amount of computing resources and much longer execution time. Their execution may require the support from a cluster of computing nodes rather than a single node. If a computing activity is scheduled on a cluster of nodes, assigning its execution on each node in a cluster is handled by a local scheduling service for the cluster. Therefore, multiple levels of scheduling may be required for scheduling a computation for a large-scale distributed system.

Second, all processors in a multi-processor system belong to a single administration domain. The administrator of the multi-processor system has complete control over the use of the processors. We also assume that there are no other concurrent computations running on the same processors. As a result, after a schedule is produced, the request for executing each task in a schedule can always be accepted and thus the success of a schedule can be guaranteed. This assumption, however, rarely holds in a large-scale distributed system, where computing nodes may belong to different administration domains and there are a large number of concurrent tasks competing for the same computing resources. The scheduling service cannot guarantee that the requests for task execution can always be accepted. After a schedule is produced, the scheduling service must submit the resource requests to the administrators that manage the scheduled computing nodes. The success of a schedule depends largely on whether these requests can be accepted. If a request is rejected, the scheduling service must either modify the schedule and submit new requests to the administrators, or report failure. As a result, multiple stages of scheduling may be necessary for scheduling a computation for a large-scale distributed system.

Third, multi-processor scheduling involves a simple homogeneous computing environment. All processors are fully connected with each other and they have the same processing ability. A multi-processor system is a stationary computing environment, i.e., the status of the system does not change over the course of scheduling and task execution. A large-scale distributed system, however, is highly heterogenous and dynamic. Computing nodes with different processing ability and architecture coexist in a system; the nodes may run on different platforms, support the execution of different applications, and are connected directly or indirectly by communication links with different latency and bandwidth. In addition, the status of a large-scale distributed system may change quickly over time. As a result, scheduling for a large-scale distributed system must be able to handle both the heterogeneity and dynamism of the system.

3.3 A GA-based Algorithm for Multi-processor Task Scheduling

In this section, we focus on the problem of multi-processor scheduling. We first discuss a GA-based algorithm for multi-processor task scheduling. We next evaluate the performance of this algorithm on a set of benchmark task graphs and compare its performance with traditional deterministic scheduling algorithms. Finally, we evaluate the performance of this algorithm in heterogeneous and dynamic problem environments.

3.3.1 Classifications of Scheduling Algorithms

Finding an optimal solution to the multi-processor task scheduling problem has been proven to be NP-hard except for some special cases [50]. There are numerous approaches to solving the multi-processor task scheduling problem. These algorithms can be categorized into different classes according to different criteria [51].

1. Number of processors: bounded vs. unbounded

   Some scheduling algorithms assume that there are an unlimited number of processors available to the scheduled tasks. These algorithms will use as many processors as possible in order to reduce the makespan of the schedule. If, however, the number of processors used by a schedule is more than the number actually available in a given problem, a mapping process is required to merge the tasks in the proposed schedule onto the actual number of available processors. On the other hand, algorithms that assume a bounded number of processors do not use additional processors other than the ones given by a scheduling problem. Therefore, this mapping process is not needed.

2. Task duplication: allowed vs. not allowed

   Some algorithms restrict the assignment of a task to only one processor. As a result, a task is not allowed to be executed on multiple processors. Although this class of algorithms is usually simpler, they have difficulty finding an optimal schedule in problems where the inter-task communication cost is relatively high. Algorithms that allow task duplication are able to take advantage of the implied parallelism in task execution. By assign-

ing a single task to multiple processors, we can remove some unnecessary communication cost and further minimize the makespan.

3. Deterministic vs. non-deterministic algorithm

   Based on the existence of randomness during the search process, we can classify the search algorithms into deterministic and non-deterministic algorithms. Deterministic algorithms typically use domain-specific heuristics to find solutions. They are efficient algorithms as the search is narrowed down to a very small portion of the search space; however, the performance of these algorithms is heavily dependent on effectiveness of the heuristics. Therefore, they are not likely to produce consistent results on a wide range of problems. On the other hand, the combinatoric process in non-deterministic algorithms requires sufficient sampling of solutions in the search space and have shown robust and consistent performance on a variety of search problems. Genetic algorithms [52, 53, 54], simulated annealing [55, 56, 57], and Tabu search [58] have been successfully applied to task scheduling. Non-deterministic algorithms, however, are less efficient and have considerably higher computational cost than deterministic algorithms.

The GA-based scheduling algorithm presented in Section 3.3.3 is a non-deterministic algorithm, assumes a bounded number of processors, and allows task duplication.

3.3.2 Previous Work on Applying GA to Scheduling

GAs have been applied to the task scheduling problem in a number of ways [51, 52, 53, 54, 59, 60, 61, 62, 63]. The existing algorithms can be categorized into

two approaches: either use GA to determine the priorities of task assignment for list scheduling techniques, or use GA directly for task assignment.

List scheduling techniques are widely used in scheduling algorithms. The basic idea of list scheduling is to assign each task a priority and then schedule the tasks in the order of their priorities. Variable heuristic functions have been used to prioritize tasks. Examples of using GA to determine the order of task assignment for list scheduling techniques include [59, 62]. These approaches encode the solution with a vector of length n where n is the number of tasks to be scheduled. Each value of the vector represents a task priority for Task $t_i, i = 0, ..., n$. Tasks are ordered by increasing i. The initial population consists of one individual with priority values based on the longest path to an exit node on the DAG and the remaining individuals consisting of randomly permuted priority values from the first individual. Traditional crossover and mutation operators are used to generate new individuals. The job of the GA is to generate new combinations of priority values. Tasks are sorted based on priority value, then are scheduled using basic list scheduling techniques. Kwok and Ahmad [53] use a coarse-grained parallel GA in combination with a list scheduling heuristic. Individuals are again vectors of length n where n is the number of tasks to be scheduled. The elements of a vector represent the tasks themselves and the order of the tasks gives the relative task priorities. As with other ordering problems such as the Travelling Salesman Problem, a number of order-based crossover operators are discussed. Mutation involves swapping tasks.

Alternatively, GAs have also been used to directly evolve task assignment and order in processors. Hou et al. [52] use a GA to evolve individuals consisting of multiple lists, with each list representing the tasks assigned to one processor. Crossover exchanges tasks between corresponding processors from two different

individuals. Mutation exchanges tasks within a single individual. This approach restricts the actions of genetic operators to ensure the validity of evolved individuals. As a result, some parts of the search space may be unreachable. Correa et al. [61] improve upon Hou's method to allow the entire search space to be searched. Tsuchiya et al. [54] implement a GA scheduler that allows task duplication: one task may be assigned to multiple processors. They compare their GA to DSH, a heuristic-based list scheduling algorithm, and show that the GA is able to find comparable or better solutions. Wang et al. [63] use a GA to evolve schedules to a heterogeneous environment. The solution is encoded with two strings: a mapping string for assigning tasks to machines, and a schedule string for specifying the sequence of tasks on each machine. Specially designed genetic operators are employed to guarantee the validity of the evolved solution. All of these GA approaches require special methods to ensure the validity of the initial population and to ensure that crossover and mutation do not destroy the validity of solutions. Therefore, all individuals generated by these systems must encode "executable" schedules. Zomaya et al. [64] incorporate heuristics in the generation of the initial population of a GA and perform a thorough study of how GA performance varies with changing parameter settings. The genetic-based algorithm we present in Section 3.3.3 is used to directly evolve task schedules.

3.3.3 An Incremental Genetic-based Algorithm

Many of the existing GA-based scheduling algorithms use special methods for solution initialization and genetic operation. Although these methods are effective in reducing the search space of a GA and maintaining the validity of solutions, they may also result in biased search which in turn may deteriorate the quality

of solutions. For instance, Hou's method [52] uses restricted genetic operators which are found to render parts of the solution space unreachable and the order in which tasks are specified in [59, 62] affects the likelihood that two tasks will be crossed over.

Keeping this lesson in mind, we attempt to minimize the amount of arbitrary human input in our GA design, particularly in our problem representation and fitness function. We implement a novel GA approach for task scheduling on multiprocessor systems. Our GA extends the traditional GA [65, 66] in two major ways.

First, we use a dynamically adaptive representation that allows a GA to evolve both the structure and the value of the solutions. Individuals have variable lengths and may contain non-coding regions (regions that do not contribute to encoding a solution). Both valid and invalid individuals may exist in the population during evolution.

Second, we use a dynamically adaptive, incremental fitness function that initially rewards for simple search goals and gradually increases the difficulty of the goals until a complete optimized solution is found. Previous experiments have shown that given the number of possible orderings of tasks in processors, the percentage of the orderings being valid is very small. If a GA is not restricted to only work with valid individuals, the chance of randomly finding a valid ordering, let alone a good valid ordering, may be very low. Restricting a GA to only form valid individuals, however, may introduce unexpected biases in the system and such systems may require extensive revision with each new problem. Instead of placing restrictions on the individuals that can be formed or using special operators or repair mechanisms to ensure validity, we use this incremental fitness function to encourage the formation of valid solutions by successively combining

rewarded task sequences. Previous work has shown that gradually increasing the difficulty of a GA fitness function can result in the formation of more complex solutions [67].

The basic algorithm is the same as a traditional GA. Details that are specific to our approach are described below.

3.3.3.1 Solution Encoding

The importance of tightly-linked or compactly encoded building blocks in a GA representation has long been recognized [65, 66, 68]. Compactly arranged building blocks (building blocks with low defining length) are expected to be more likely to be transmitted as a whole by the genetic operators during a reproduction event [69]. Location independent problem representations, where the information content is not fixed at specific locations on a GA individual, have been proposed in a number of studies as a way to help a GA identify and maintain tightly-linked building blocks. Such representations allow for rearrangement of encoded information [20, 23, 70, 71, 72, 73, 74, 75, 76], overlapping encodings which can be more space efficient [71, 77, 78], and the appearance of non-coding regions which affects crossover probability [22, 23, 49, 71, 79, 80, 81, 82]. In some location independent representations, the arrangement of encoded information will determine what is expressed [20, 73, 74] even though the actual encoded content is not determined by its location. We use such a representation in which the meaning of an encoded element is independent of its location on an individual, but its location does determine whether or not it is expressed.

Each individual in a GA population consists of a vector of cells. We define a *cell* to be a pair of task and processor: (t, p). Each cell indicates that Task t

is assigned to be processed on Processor p. The number of cells in an individual may vary, so individuals in a GA population will vary in length. Figure 3.5 shows an example individual. The first cell of this individual assigns Task 4 to Processor 1, the next cell assigns Task 2 to Processor 4, etc.

(4,1)(2,4)(7,3)(2,3)(4,1)(5,4)(6,3)(1,1)(3,2)

Figure 3.5: An example individual.

The cells on an individual determine which tasks are assigned to which processors. The order in which the cells appear on an individual determines the order in which the tasks will be performed on each processor. In the encoding process, individuals are read from left to right. The task-processor pairs that are read early are assigned first. Thus, the order in which tasks will be performed on each processor depends on the order in which the task-processor pairs appear on an individual. For example, the individual shown in Figure 3.5 results in the processor assignments and ordering of tasks shown in Figure 3.6. Invalid task orderings will have their fitness value penalized by the fitness function.

Processor 1	Task 4	Task 1	
Processor 2	Task 3		
Processor 3	Task 7	Task 2	Task 6
Processor 4	Task 2	Task 5	

Figure 3.6: Assignment of tasks from individual in Figure 3.5.

As we allow task duplication in solutions, the same task may be assigned more than once to different processors. The example individual in Figure 3.5

assigns Task 2 to processors 3 and 4. Tasks may not be assigned to the same processor more than once. If a task-processor pair appears more than once on an individual, only the first (leftmost, since individuals are read from left to right) pair is active and encoded in the solution. Any remaining identical pairs are essentially non-coding regions and are not encoded in the solution. In the example from Figure 3.5, the second instance of (4,1) is not scheduled into the processor lists in Figure 3.6.

The initial population is created with randomly generated individuals. Each individual consists of exactly one copy of each task. As a result, the length of all individuals in an initial population is equal to the number of tasks specified in the target DAG. Each task is randomly assigned to a processor. The initial population, however, may not encode valid solutions as the task sequences are randomly generated and may not obey precedence restrictions.

3.3.3.2 Genetic Operations

We use both crossover and mutation in our algorithm. Slight modifications are necessary to work with this representation. The modified versions of these genetic operators are described here.

Crossover

We use random one-point crossover. A crossover point is randomly chosen between two adjacent genes from each parent. The segments to the right of the crossover points are exchanged to form two offspring. Figure 3.7 shows an example of random crossover.

Randomly select parent 1 crossover point: 2

Randomly select parent 2 crossover point: 4

Parent 1	(4,1)(2,4)| (3,3)(2,3)(5,4)(1,1)(3,2)
Parent 2	(4,3)(3,3)(5,2)(3,4)| (2,4)(3,3)

Random crossover produces

Offspring 1	(4,1)(2,4)| (2,4)(3,3)
Offspring 2	(4,3)(3,3)(5,2)(3,4)| (3,3)(2,3)(5,4)(1,1)(3,2)

Figure 3.7: Random one-point crossover randomly selects crossover points on each parent and exchanges the right segments to form offspring.

The parameter of crossover rate gives the probability that a pair of parents will undergo crossover. In addition, if a crossover operation generates an offspring individual that exceeds the maximum allowed genome length, crossover does not occur. Parents that do not crossover transform unchanged into offspring, but they may still undergo mutation.

Mutation

Each cell has equal probability of being mutated. The probability is given by a parameter called mutation rate. The expected number of mutations per individual is equal to the mutation rate multiplied by the length of an individual. If a cell is selected to be mutated, either the task number or the processor number of that cell will be randomly changed.

3.3.3.3 Selection

We use binary tournament selection to form a new generation of solutions. We randomly select two individuals each time and compare their fitness. We always choose the individual with higher fitness and copy it to the new generation. We continue this selection process until we have selected a new population of the same size as the current one. We don't use elitism method to keep the best solution over generations.

3.3.3.4 Fitness Evaluation

The fitness function consists of two independent parts. The first part of the fitness function, $task\_fitness$, focuses on ensuring that all tasks are performed and scheduled in valid orders. The second part of the fitness function $processor\_fitness$, attempts to minimize the execution time of valid schedules. The actual fitness, $fitness$, of a GA individual is a weighted sum of the above two partial fitness values.

Calculating *task_fitness*

The *task_fitness* component of the fitness function evaluates whether all tasks are represented and in valid order. A pair of tasks is independent if neither task relies on the data output from the other task for execution. The scheduling of a pair of tasks to a single processor is valid if the pair is independent or if the order in which they are assigned to the processor matches the order of their dependency. The scheduling of a group of tasks to a single processor is valid if the order of every pair of tasks in the group is valid. A solution is valid if all of its processor schedules are valid.

Because of the complexity of the solutions, we develop an incremental fitness function that changes over time. We initially reward for finding short valid sequences of tasks. Over time, we increase the length of the sequences that can be rewarded, encouraging the GA to find and maintain longer valid sequences. Eventually the valid sequences will be long enough that the individuals will represent full valid solutions. As in positive reinforcement training, this strategy rewards for small steps toward the goal, to encourage the algorithm to find the complete goal.

The $task\_fitness$ component of an individual's fitness is based on two main components: the percentage of valid sequences of a given length and the percentage of the total tasks specified by an individual. Initially the fitness function will reward for short sequences of valid tasks. A sequence of tasks is valid if the tasks in the sequence are independent to each other or their order of assignment abides their precedent relations. When the average fitness of the GA population exceeds a threshold fitness, the length of the sequence for which the GA searches is increased, thus increasing the difficulty of the fitness function.

(a) *Calculating raw fitness*: The raw fitness of an individual reflects the percentage of sequences of a given length in an individual that are valid sequences. For example, suppose we are working on LU Decomposition task graph shown in Figure 3.2. Processor 3 in Figure 3.6 has been assigned three tasks. If the current sequence length is two, Processor 3 contains two sequences of length two. Processor 3 contains only one valid sequence of length 2, the sequence `Task2-Task6`. The sequence `Task7-Task2` is not a valid sequence because Task 7 cannot be executed before Task 2.

Assume that the problem to be solved involves P processors and T tasks. Evolution will occur in eras, $era = 0, 1, 2, ..., E$. Initially, $era = 0$. The maximum

era count, $E \leq T$, is a user defined parameter value. The era counter, *era*, is increased when the average population fitness exceeds a user defined threshold, *thresh*, and when the number of individuals with the current maximum fitness exceeds a user defined threshold, *thresh_maxfit*. Unless otherwise specified, we use $thresh = 0.75$ and $thresh\_maxfit = 0.1$.

Let $numtasks(p), p = 1, ..., P$, indicate the number of tasks assigned to processor p. To calculate the raw fitness of a processor, we need to consider two things: the first $era + 1$ (or fewer) tasks assigned to the processor, and all task sequences of length $era + 2$. The first component is important because as *era* increases, the likelihood of processors containing fewer than $era + 2$ tasks increases. We need to reinforce the GA for these shorter sequences in order for them to eventually build up to the measured sequence length.

We will first determine the contribution of the first $era + 1$ or fewer tasks in a processor. Let

$$subseq(p) = \begin{cases} 1 & \text{if } numtasks(p) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

and let

$$valseq(p) = \begin{cases} 1 & \text{if the first } era + 1 \text{ or fewer tasks in Processor } p \text{ are in valid order} \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

Equations 3.1 and 3.2 refer to individual processors. To calculate the contribution over all processors (the contribution for the entire individual), we let

$$Subseq = \sum_{p=1}^{P} subseq(p)$$

$$Valseq = \sum_{p=1}^{P} valseq(p).$$

We will next determine the contribution of all sequences of length $era + 2$ in a processor. Let

$$s(p) = \text{number of sequences of length } era + 2 \text{ in Processor } p \tag{3.3}$$

and let

$$v(p) = \text{number of valid sequences of length } era + 2 \text{ in Processor } p. \tag{3.4}$$

Combining equations 3.3 and 3.4 to determine the contribution over all processors we let

$$S = \sum_{p=1}^{P} s(p) \qquad V = \sum_{p=1}^{P} v(p).$$

The $raw\_fitness$ for an individual is then calculated with the following equation

$$raw\_fitness = \frac{Valseq + V}{Subseq + S}. \tag{3.5}$$

(b) *Calculating the task ratio*: In addition to encouraging the system to find valid sequences of tasks, we also want to encourage the system to include at least one copy of each task in each solution. We define the *task_ratio* to be the percentage of distinct tasks from the total tasks in the problem that are represented in an individual. The *task_ratio* is calculated with the following equation:

$$task\_ratio = \frac{\textit{number of distinct tasks specified in an individual}}{\textit{total number of tasks specified in the problem}} \tag{3.6}$$

This factor penalizes solutions that do not contain at least one copy of every task. Once all tasks are represented in an individual, we assign a value of one to $task\_ratio$.

(c) *Calculating task fitness*: The effective $task\_fitness$ of an individual is the product of equations 3.5 and 3.6.

$$task\_fitness = raw\_fitness * task\_ratio \tag{3.7}$$

This value makes up the first component of the fitness of a GA individual.

Calculating *processor_fitness*

The $processor\_fitness$ component of the fitness function encourages GA to minimize the makespan of valid schedules. As the length of a solution can only be measured if a solution exists, we assign $processor\_fitness$ to zero for all individuals that do not encode valid solutions.

Suppose t is the run time for a solution represented by an individual. Let $serial\_len$ equal the length of time required to execute all tasks serially on a single processor and let $super\_serial\_len = P * serial\_len$ where P is the number of processors. Any reasonable solution should give $t \ll super\_serial\_len$, making $super\_serial\_len$ a safe but reasonable upper bound for the makespan of schedules. The goal of the GA is to minimize t. The $processor\_fitness$ first calculates the difference between $super\_serial\_len$ and t then calculates what proportion of $super\_serial\_len$ this difference represents:

$$processor\_fitness = \frac{super\_serial\_len - t}{super\_serial\_len}. \tag{3.8}$$

As a result, $processor\_fitness$ is inversely proportional to t. As the run time of a solution decreases, the amount that $processor\_fitness$ contributes to the individual's full fitness increases.

It is important to note that although the theoretical maximum value of *processor_fitness* is 1.0, in practice, this value can not be achieved. For *processor_fitness* to equal 1.0, the run time, t, of a solution would have to be zero. Since all tasks obviously require non-zero execution time, t will never be zero for valid schedules.

Calculating *fitness*

The full fitness of an individual is a weighted sum of the $task\_fitness$ and $processor\_fitness$:

$$fitness = (1 - b) * task\_fitness + b * processor\_fitness, \tag{3.9}$$

where $0.0 \leq b \leq 1.0$. Unless otherwise specified, we use $b = 0.2$.

If an individual does not encode a valid solution, we are unable to evaluate *processor_fitness*. As a result, $processor\_fitness = 0$ and

$$fitness = (1 - b) * task\_fitness + b * 0 = (1 - b) * task\_fitness. \tag{3.10}$$

3.3.4 Performance Evaluation

We evaluate the performance of this GA algorithm on both static and dynamic environments. In the experiments on static environments, we evaluate the GA on both a homogeneous environment and a heterogeneous environment. In all experiments, we compare the performance of the GA with three heuristic-based list scheduling algorithms, ISH [83], DSH [83], and CPFD [84]. These algorithms have been widely used by the researchers as benchmark algorithms in task scheduling [85]. DSH and CPFD allow task duplication; ISH does not. CPFD has been shown to consistently outperform other state-of-the-art scheduling algorithms [84].

The following parameter settings were empirically determined to be good values for our GA. Unless otherwise specified, we use the parameters listed in Table 3.1 in our experiments:

We use a variable length representation with a maximum length of $2 \times T$ where T is the number of tasks in the problem.

Table 3.1: Parameter settings for GA.

Parameter	Value
Population Size	400
Number of Generations	3000
Crossover Type	Random one-point
Crossover Rate	0.8
Mutation Rate	0.005
Selection Scheme	Tournament
Tournament Size	2

3.3.4.1 Comparison with traditional list scheduling methods

In this experiment, we select nine task graphs as test cases. Six of the cases use the 14-node LU decomposition task graphs [54], 15-node Gauss-Jordan task graphs [54], and 16-node Laplace task graphs [86], each with two different inter-task communication cost settings. The other three problems have 15, 17, and 18 nodes, and are selected from [87], [88], and [86], respectively. Table 3.2 lists the nine test problems.

Table 3.3 shows the best solutions obtained for each problem by each method. Because the GA is a stochastic algorithm, we perform 50 runs for each problem and also report its average results.

The GA outperforms traditional methods on one problem (P5), performs as well as the best traditional method on six problems, and achieves the second best performance on two problems (P7 and P9). Doubling the GA population size allows the GA to also outperform traditional methods on Problem P6. Results

Table 3.2: The list of test problems.

Problem Id	Task Graph	Communication Cost
P1	Gauss-Jordan	25
P2	Gauss-Jordan	100
P3	LU Decomposition	20
P4	LU Decomposition	80
P5	15-node	Variable
P6	17-node	Variable
P7	18-node	Variable
P8	Laplace	40
P9	Laplace	160

indicate that, given sufficient resources, the GA is able to equal or outperform traditional scheduling methods.

Interestingly, the data in Table 3.3 suggest that the advantages of task duplication in these scheduling methods are particularly noticeable on problems with longer communication times. Problems P1 and P2 share the same DAG and differ only in their communication times: Problem P2 has a significantly longer communication time than Problem P1. The same holds true for Problems P3 and P4 and Problems P8 and P9. Problems P2, P4, P5, P7, and P9 have communication times that are larger than the task execution times (significantly larger for P5) and show noticeable improvement when using methods that allow task duplication. Problems P1, P3, P6, and P8 have communication times that are equal or less than task execution times and show little improvement with the additional of task duplication.

Table 3.3: Minimum makespan found by ISH, DSH, CPFD, and GA. CI = confidence interval. *In a second set of runs in which the population size is doubled, the GA finds a minimum makespan of 36 and average makespan of 36.92 with a 95% confidence interval of 0.17.

Test Problem	ISH	DSH	CPFD	GA		
				Best	Average	95% CI
P1	300	300	300	300	300	0
P2	500	400	400	400	430	5.28
P3	260	260	260	260	263.4	2.06
P4	400	330	330	330	370	6.41
P5	650	539	446	438	445.92	6.05
P6	41	37	37	37*	37.78*	0.24*
P7	450	370	330	350	380.6	4.76
P8	760	760	760	760	782.8	5.63
P9	1220	1030	1040	1040	1101.8	14.23

An examination of scalability to larger problems finds that GA performance declines as the problem size increases. GAs tend to require larger populations to maintain performance as problem size increases, e.g. when P4 is scaled up to be a 27-node problem, a GA using population size 400 finds a minimum makespan of 680; a GA using population size 800 finds a minimum makespan of 650. These results indicate that a GA requires sufficient resources in order to find good solutions.

A comparison of execution times finds that the cost for having sufficient resources is a longer execution time. ISH, DSH, and CPFD consistently post run times of less than one second for the problems that we tested. The GA requires

significantly longer execution times. Table 3.4 gives the average number of generations and seconds to find a good solution using the GA. Traditional methods clearly outperform the GA in terms of execution time.

Table 3.4: Average number of generations and average clock time (in seconds) using a GA. CI = confidence interval.

Test problem	Average (95% CI) generations to best solution	Average (95% CI) time to best solution	Average (95% CI) time for one run
P1	682.26 (191.30)	30.70 (11.01)	129.37 (11.18)
P2	1011.88 (219.07)	54.49 (13.37)	164.20 (14.54)
P3	934.34 (213.50)	28.15 (6.73)	95.05 (4.43)
P4	1333.36 (247.21)	60.78 (13.89)	140.17 (12.10)
P5	871 (180.38)	36.16 (7.74)	137.98 (12.39)
P6	1375.46 (246.84)	80.36 (16.92)	187.35 (15.52)
P7	1316.62 (248.24)	77.03 (14.46)	178.36 (14.90)
P8	1168.18 (195.33)	73.97 (12.77)	192.12 (12.98)
P9	1627.72 (237.77)	130.83 (20.15)	248.69 (28.43)

Figure 3.8 shows an example of how a typical GA run proceeds. Figure 3.8(a) shows the evolution of population fitness. The top line shows the best population fitness at each generation. The bottom line shows the average population fitness at each generation. The vertical lines indicate the generations at which the *era* counter is incremented. The start of each era is indicated at the top of the graph. The average population fitness climbs within each era. Each time the *era* counter is incremented, however, the difficulty level of the fitness function increases and the average fitness of the population drops. After about six eras in this run, there are apparently enough valid task sequences to allow the remaining

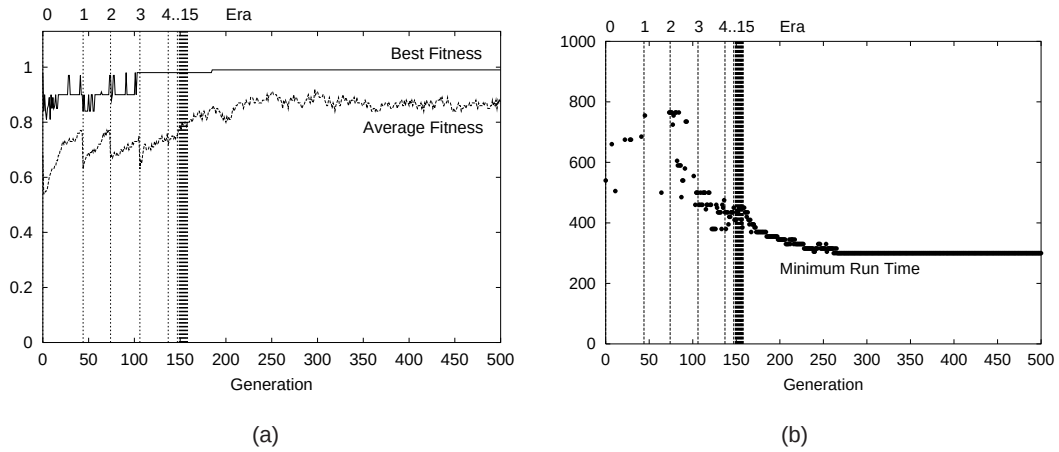


Figure 3.8: Evolution of (a) population fitness and (b) minimum makespan in response to increasing eras.

eras to increment once per generation until the maximum $era = 15$ is reached. Figure 3.8(b) shows the evolution of run time or makespan in the same run. The minimum makespan can only be calculated from a valid solution. Early in the run, with lower values of era, valid solutions are found only sporadically. Over time, valid solutions are found more consistently and the minimum makespan decreases steadily.

3.3.4.2 Evaluation of the Effectiveness of the Fitness Function

The fitness function of a GA can have a significant impact on the effectiveness of the algorithm. Our fitness function has several parameters that can vary. We test the sensitivity of the GA to variations in these values. Specifically, we examine GA performance on problems P1 and P3 where $b \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$ and $thresh \in \{0.25, 0.5, 0.75, 1.0\}$.

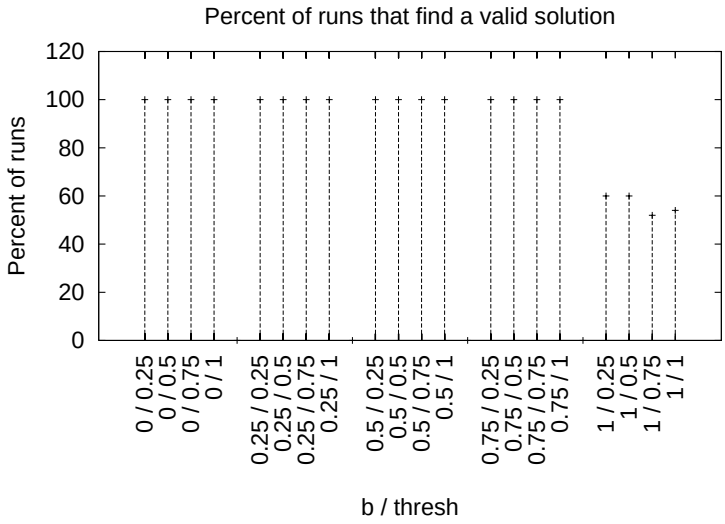


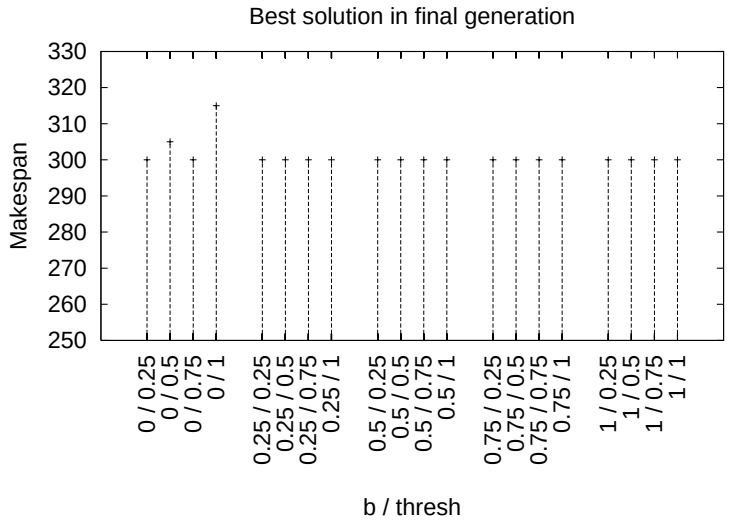
Figure 3.9: Problem P1: Percent of runs that find a valid solution. X-axis indicates $b/thresh$ values.

Figures 3.9 and 3.10 show the results for Problem P1. Similar results were obtained for Problem P3.

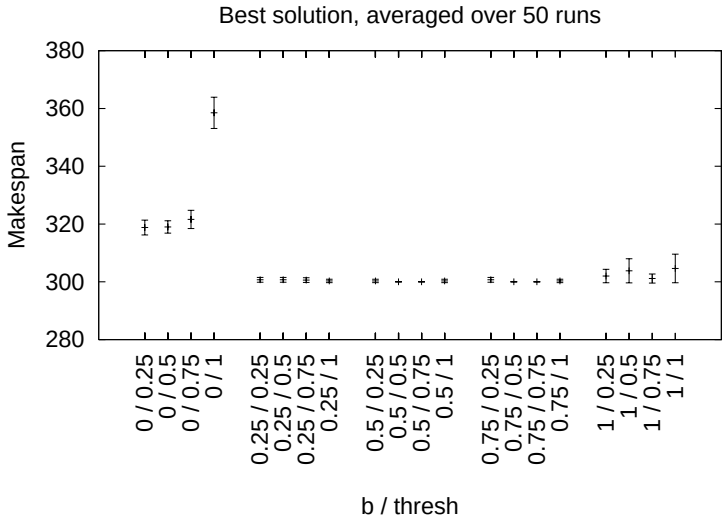
Figure 3.9 shows the percent of runs in each experiment that find at least one valid solution. Figure 3.10 shows (a) the minimum makespan achieved by the GA and (b) the minimum makespan found in the final generation averaged over 50 runs. Data are obtained for each combination of b and $thresh$ values. The x-axis labels indicate the $b/thresh$ combination for each set of runs.

Intermediate values of b consistently produce good performance, finding a minimum makespan of 300 and average best makespans of 300 or slightly above. The narrow confidence intervals in Figure 3.10(b) indicate that most runs are able to find a best solution at or close to 300. Varying values of $thresh$ appear to have little impact on the results.

Extreme values of b have a noticeable negative impact on GA performance. Setting $b = 0.0$ produces suboptimal, though still respectable, results, with min-



(a)



(b)

Figure 3.10: Problem P1: X-axis indicates $b/thresh$ values. (a) Minimum makespan. (b) Average best makespan averaged over 50 runs* with 95% confidence intervals. *When $b = 1.0$, not all 50 runs are able to find valid solutions. The average values shown are calculated only from those runs that do find valid solutions.

imum makespans as high as 315. When $b = 0.0$, the fitness function consists of only the $task\_fitness$ portion which focuses only on finding valid solutions (it rewards for valid substrings of tasks and rewards for having at least one copy of each task). The length (makespan) of a solution is irrelevent as the fitness function does not give any reward for shorter solutions. As a result, the GA is able to find solutions; however, the lack of pressure for smaller solutions is apparent as all of the GA runs with $b = 0.0$ find significantly longer solutions that those runs with intermediate values of b. The larger 95% confidence interval indicates a wider range of makespan values found.

Setting $b = 1.0$ makes it difficult for the GA to find valid solutions. Figure 3.9 shows that the GA is unable to find a valid solution in every run when $b = 1.0$. When valid solutions are found, however, the GA finds good solutions, although not as consistently as with intermediate values of b. When $b = 1.0$, the fitness function consists only of the $processor\_fitness$ component which is activated only if an individual encodes a valid solution. Only when an individual encodes a valid solution will the fitness function return a non-zero value. As a result, there is no feedback for partial solutions. With no fitness reward for partial solutions consisting of short valid task orderings, the GA has difficulty finding valid solutions. Thus, rewarding for valid partial orderings appears to be an important component of the algorithm's success.

The $thresh$ parameter appears to have little impact on the quality of solutions found for intermediate values of b. For $b = 0.0$ and $b = 1.0$, performance declines with increasing values of $thresh$.

Table 3.5: Minimum makespan found by ISH, DSH, CPFD, and GA on a heterogeneous problem. CI = confidence interval.

Test Problem	ISH	DSH	CPFD	GA		
				Best	Average	95% CI
P1	300	300	345	315	333.7	3.70
P2	500	440	460	420	462	8.91
P3	320	310	260	260	288.4	5.50
P4	400	350	360	350	396.9	8.13
P5	549	470	606	539	559.9	8.65
P6	46	42	43	40	42.94	0.60
P7	470	410	370	360	413.2	10.33
P8	840	840	860	810	889.4	14.40
P9	1220	1130	1210	1060	1187	23.95

3.3.4.3 Comparison using heterogeneous processors

We also compare the four algorithms in a more complex environment in which the processors are heterogeneous. In this experiment, we double the processing time for processor 2 and triple the processing time for processor 4. Processors 1 and 3 remain unchanged, Table 3.5 compares the quality of the solutions found. Our GA exhibits the best performance on five of the problems. On Problems P3 and P4, CPFD and DSH, respectively, perform equally as well as the GA. On Problems P1 and P5, the GA comes in second to DSH. Among the traditional methods, DSH appears to perform better than CPFD on heterogeneous problems.

3.3.4.4 Experiments on Dynamic Environments

The results on stationary problem environments indicate that, while a GA can find very competitive solutions, its execution times are likely to be longer than traditional methods. Why then would one choose to use a GA over faster traditional methods?

We expect the strengths of this GA approach to be in its flexibility and adaptability in non-stationary environments [89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]. Non-stationary problem environments are those in which the desired solution changes over time. Such environments can be difficult for traditional scheduling algorithms; most must be re-started, many reconfigured or re-programmed, for each new situation. We believe that the flexibility of our GA and its representation will allow it to automatically adapt to changes in the fitness evaluation with no change or interruption to the algorithm itself.

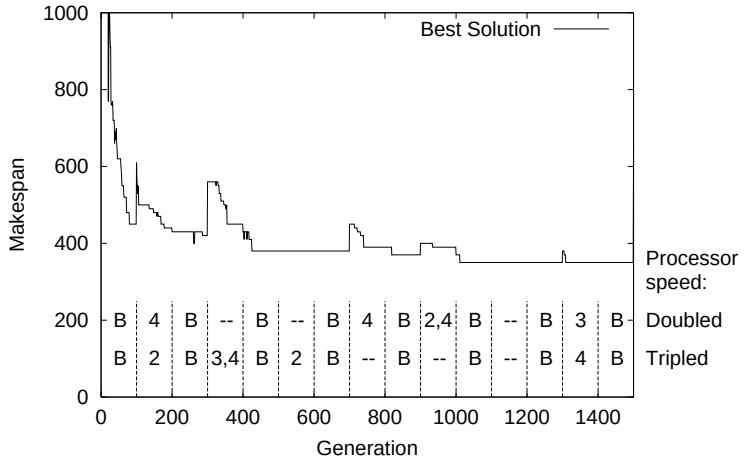
The experiments in this section investigate the GA's ability to adapt in a non-stationary environment where processor speeds can change over time. Once a GA run has started, no human intervention or interruptions are allowed; the GA must adapt automatically to changes in the target problem. Within a multiprocessor system, processor loads may vary depending on the number of tasks under execution and how they are distributed among the processors. As a processor's load increases, its execution speed is expected to decrease. Ideally, as processor loads change, the system will automatically redistribute workload among the processors to take advantage of processors with low load and minimize assignments to processors with high load. An algorithm that is able to adapt automatically to such changes can significantly improve the efficiency of managing a multiprocessor system. In addition, the underlying problem has now changed and become

more difficult: multi-processor task scheduling for heterogeneous processors in non-stationary environments.

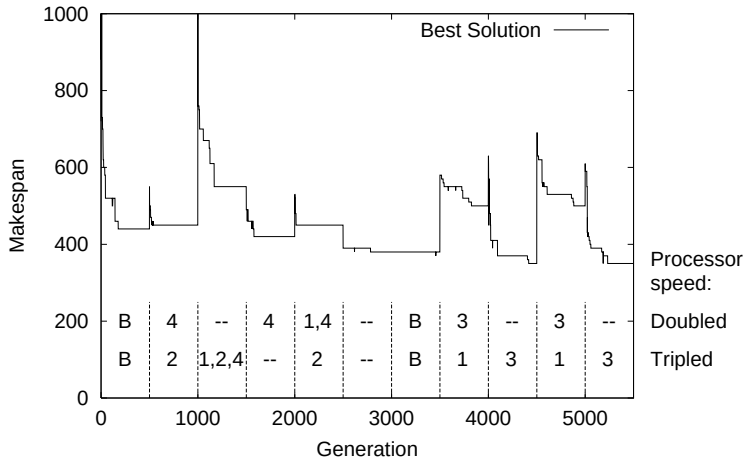
In this experiment, a GA run begins by finding a task schedule for four identical processors at minimal load. We call this situation the *base target*. We change the target problem by increasing processor speeds to double and triple the minimal speed. At fixed intervals, each processor has a 30% chance of doubling its speed and a 20% chance of tripling its speed. We call these situations *modified targets*. We test intervals of $I = \{100, 500\}$ generations.

Figure 3.11 shows in two example runs how the evolved solutions of a GA change as processor speed changes. "B" indicates the base target. Integer values indicate modified processors. Processors are randomly selected to be modified. The optimal makespan for the base target is 330. Figure 3.11(a) shows an example run with $I = 100$ where each interval with a modified target is followed by an interval with the base target. Figure 3.11(b) shows an example run with $I = 500$ where multiple consecutive intervals can have modified targets. The base target is assigned to generations 0-500 and generations 3000-3500 to provide a baseline comparison.

Results indicate that this GA approach is able to automatically adapt to changes in the target solution. In both examples, the GA continues to improve the solutions generated throughout a run. As expected, makespan increases sharply after a target change to a modified target, but solutions immediately begin to improve. Figure 3.11(b) shows less stable solutions than Figure 3.11(a). We speculate that there are two potential causes for this difference. First, longer intervals of 500 generations give the GA more time to optimize solutions and converge the population for the current target. As a result, it is less likely that the population will have solutions that perform well for other modified targets.



(a)



(b)

Figure 3.11: Two example GA runs. Evolution of best solution in a non-stationary environment in which the processor speed changes at fixed intervals. "B" indicates the base target. Integer values indicate modified processors.

Second, not having a base target in between each modified target gives the GA no time to "neutralize" its solutions between modified targets. Overall, the GA is able to evolve near optimal solutions in both experiments, even when processor speeds are increased.

3.4 A Scheduling Algorithm for Large-scale Distributed Systems

This section is focused on a scheduling algorithm for large-scale distributed systems. This algorithm is extended from the GA-based algorithm for multiprocessor scheduling (discussed in Section 3.3). We formulate the problem of scheduling for large-scale distributed systems, present the algorithm, and evaluate its performance in simulation environments.

3.4.1 Problem Formulation

We make the following assumptions before we formulate the problem of scheduling for a large-scale distributed system.

(a) The scheduling of a computation is at the activity level, i.e., the goal of a scheduling algorithm is to assign the execution of each end-user activity to an available computing node in a system.

(b) Although it is generally difficult to estimate the computational cost and the size of the output data from the execution of each activity, we assume that the statistics on previous executions of this activity can be used for an estimation.

(c) Not all nodes support the computation for each end-user activity. We assume that a scheduling algorithm is able to know the set of all computing nodes that support the computation for each end-user activity in a process description. An activity is always assigned to a computing node that supports its execution.

(d) We assume that a scheduling algorithm is able to track the current status of a large-scale distributed system, including the processing ability of the computing nodes and the bandwidths of the communication links between the nodes. These data are important for producing an optimized schedule or adapting an existing schedule to the new computing environment.

With the above assumptions, we formulate the problem of scheduling for large-scale distributed systems. This scheduling problem requires the following four aspects of input data.

(a) The process description of a computing task. The process description is either given by a user or created by a planning service. The process description may not reflect the actual flow of control if it contains iterative and/or conditional execution of activities.

(b) The estimated computational cost of each end-user activity in a process description. The computational cost typically depends on the size or content of the input data and can rarely be estimated accurately. The computational cost may be different over successive iterations. A rough estimation can be obtained from the history of activity execution.

(c) The estimated size of data transferred between dependent activities in a process description. Again, this aspect of input data cannot be estimated accurately and depends on a lot of factors. In addition, the size of data transfer between tasks may vary over different invocations of the activity execution.

Statistics on the history of task execution or domain-specific knowledge may provide useful guidance for an estimation.

(d) The information regarding the set of computing nodes capable of performing each end-user activity in a process description, their processing abilities as well as the network topology and the bandwidths of the communication links. All above information is available to a scheduling algorithm.

The output of the scheduling algorithm is an execution schedule of end-user activities that minimizes the total execution time of a computation.

3.4.2 Interactions with Other Services in the Middleware

When an execution plan is produced by the planning service, a request for scheduling a computation is sent from the coordination service to the scheduling service. The request contains the process and case descriptions of the computing task. After the request is received, the scheduling service generates a deterministic DAG based on the process description (will be discussed in Section 3.4.3). The scheduling service next contacts each end-user service that supports the execution of an activity in the DAG for an estimation of the execution time and the available time periods for executing the activity. The scheduling service also needs to interact with the monitoring service for the current conditions of the computing environment. When all above information is available, the scheduling service applies the algorithm and generates a schedule for execution. The scheduling service next contacts all end-user services to which the computation is scheduled for requesting the execution at scheduled times. If any of the requests are not accepted, the scheduling service needs to modify the schedule and send

the requests to the end-user services again. If a schedule cannot be produced, the scheduling service reports failure to the coordination service. The coordination service may either ask the planning service for replanning, or send a request to plan switching service for switching the execution to another plan. Otherwise, the scheduling service sends the schedule to the coordination service for execution. As the decision of the execution of some end-user activities is determined by the runtime results of other activities, we may not be able to schedule all activities in a process description and finish the execution at one time. After the scheduled activities finish execution, the coordination service may send another request for scheduling subsequent activities. The above process continues until the computation finishes completely. Figure 3.12 shows the typical flow of communications between the scheduling service and other services in the middleware during the course of scheduling a computing task.

3.4.3 A Modified GA-Based Algorithm

The existing GA-based algorithm for multi-processor scheduling can only accept task graphs given by deterministic DAGs in which all activities must be executed exactly once. The scheduling algorithm cannot schedule activities whose execution is determined by run-time computation results. We call a task graph *non-deterministic* if there exists conditional or iterative execution of tasks. The existing algorithm should be modified to schedule for non-deterministic task graphs.

Some studies have been reported on scheduling task graphs that contain conditional execution of tasks. Chou and Abraham [100] introduce a computational model that allows conditional and concurrent task execution. They apply the

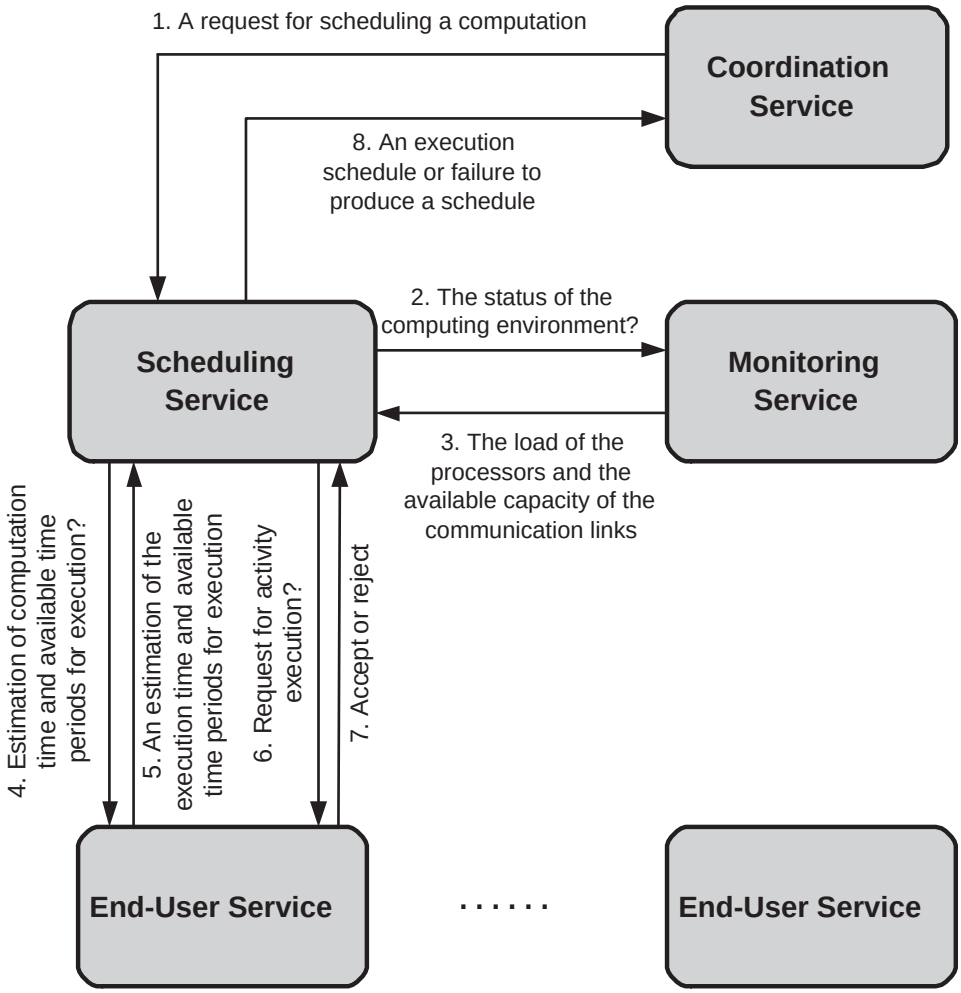


Figure 3.12: The typical flow of communications between the scheduling service and other services in the middleware during the course of scheduling a computing task.

policy iteration algorithm to optimize the task assignment. Towsley [101] builds a computational model that contains conditional tasks and iterative tasks with bounded number of iterations. They apply the shortest path method to allocate the tasks to processors and minimize the total communication and computation costs. El-Rewini and Ali [102] present a two-step approach to scheduling task graphs with conditional branches. The first step of this approach tries to explore the similarity between conditional execution branches and minimize the degree of non-determinism in a task graph as much as possible. The second step generates an optimal schedule for each execution instance (that cannot be further unified in the first step) and then merges these schedules to form the final solution. All above studies are based on well-defined models in which the probability of task execution is already known or can be estimated before the whole computation begins.

Other studies address the problem of scheduling iterative tasks on parallel processors. One class of successful techniques, called *graph unfolding* [103, 104], intends to exploit the inter-loop parallelism of task execution by unfolding loops in a task graph before applying the scheduling algorithms. Another class of techniques, called *software pipelining* [105, 106, 107, 108, 109], is widely used by compilers to overlap the execution of instructions across different iterations. Various methods of software pipelining have been proposed. One example method of using software pipelining to achieve greater parallelism is to reorganize the execution of instructions within a loop without changing the overall behavior of the complete computation.

We present a modified approach to scheduling for large-scale distributed systems. This approach has the following two features.

First, the scheduling process is divided into multiple steps. In each step, we create a deterministic DAG based on the process description. The DAG only contains the activities whose execution can be determined at the current stage. We do not include any activities in which the decision of their execution is determined by the execution results of other activities. We next apply the GA-based multi-processor scheduling algorithm, build a schedule for the current DAG, and send the schedule for execution. After the execution of the schedule finishes, we retrieve the computation results, determine the successor activities to be scheduled, and generate a new DAG for the next step. This process continues until the execution of the complete computation finishes. The following pseudo code shows the brief procedure for this scheduling approach.

```
While the computation of a task has not finished, do
  (a) Determine the activities to be scheduled and generate a DAG.
  (b) Estimate the execution time of each activity in the DAG.
  (c) Record any changes on the runtime environments.
  (d) Apply the GA-based scheduling algorithm and produce a schedule
      for the DAG.
  (e) Wait until the execution of the scheduled activities finishes.
  (f) Collect the computation results.
End while.
```

We use the same computation described in Section 2.2.3.1 as an example to demonstrate the creation of DAGs from a process description. A process description for the computation of 3D reconstruction is given in Figure 2.24. Some activities in this process description need to be executed multiple times to reach an expected resolution, but the number of iterations cannot be determined before

the computation starts. Figure 3.13(a) shows the DAG for the first iteration and Figure 3.13(b) shows the DAG for the rest of the iterations.

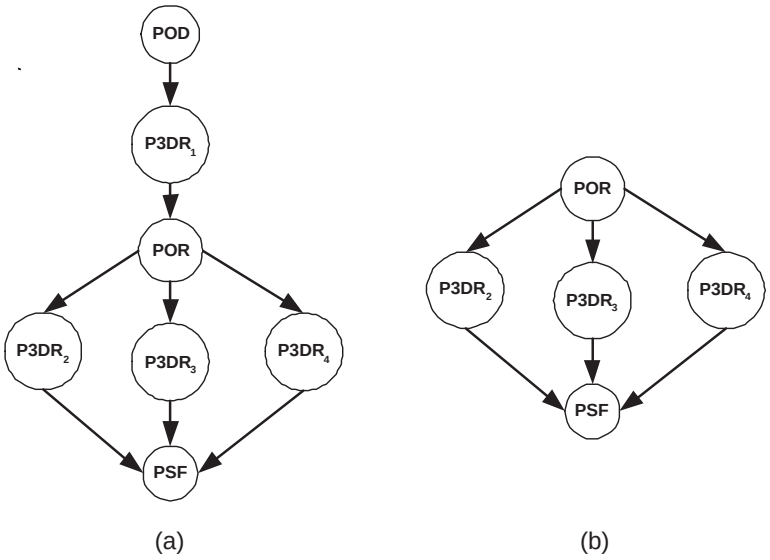


Figure 3.13: An example to demonstrate the transformation between the process description shown in Figure 2.24 and DAGs. (a) the DAG for the first iteration (b) the DAG for the rest of the iterations.

Second, we keep and reuse the previous schedules as backup solutions to improve the efficiency of the scheduling algorithm. The GA-based scheduling algorithm generates a group of diversified schedules besides the best one selected as a final solution. In a dynamic problem environment, these backup schedules may exhibit good performances for a future problem condition. Simply discarding them is a waste of genetic resources. Reusing these schedules can effectively reduce the computational cost of scheduling in subsequent steps.

We present a semi-static approach to scheduling for large-scale distributed systems. This approach requires the scheduling service to maintain a reference table that stores the valid schedules that have been evolved (note that the GA-based scheduling algorithm may evolve both valid and invalid solutions). Both

schedules and DAGs are stored in the table. We set the limit on the number of schedules for each DAG. When the number of schedules for a DAG has reached the limit, the oldest schedule (i.e., the schedule that was stored earliest) is removed from the table. When a new DAG is generated (step a. in the procedure), we retrieve all schedules for that DAG and compare their makespan under the current problem conditions. We select the best schedule as a solution. If, however, there is no valid schedule for the DAG, we rerun the scheduling algorithm, store all valid solutions in the table (and possibly remove some existing schedules), and select the best one as the solution. This semi-static approach can effectively reduce the computational time as we do not need to run the scheduling algorithm for each request of scheduling. The modified procedure of the approach is given as follows:

```
While the computation of a task has not finished, do
  (a) Determine the activities to be scheduled and generate a DAG.
  (b) Estimate the execution time of each activity in the DAG.
  (c) Record any changes on the networking environments.
  (d) Find all matching schedules for the DAG.
  (e) If no matching schedule can be found
      (i)   Run GA and select the best schedule as the solution.
      (ii)  Store all valid schedules in the population in the
            reference table.
      else
      (iii) Evaluate all backup schedules and select the best schedule
            as the solution.
  (f) Wait until the execution of the scheduled activities finishes.
  (g) Collect the runtime results of execution.
End while.
```

3.4.4 Experimental Results

We evaluate the performance of the scheduling algorithm with two experiments. In the first experiment, we evaluate the applicability of the scheduling algorithm in a large-scale multi-processor system (i.e., with a large number of processors). In the second experiment, we evaluate the effectiveness of the semi-static approach in a simulation environment.

3.4.4.1 Performance Evaluation on a Large-scale Multi-processor System

In this experiment, we study how well the GA-based scheduling algorithm performs when the number of processors increases. We repeat the same experiment in Section 3.3.4.1 but with 10, 20 and 50 processors. We use the same parameters shown in Table 3.1 and test the GA algorithm on the same set of task graphs shown in Table 3.2. We test each task graph 50 times and select the best solution of each run as the result. For each task graph, we calculate the best, the average, and 95% confidence interval of the results from all 50 runs. Tables 3.6, 3.7, and 3.8 show the experimental results on 10, 20, and 50 processors, respectively and compare the results with ISH, DSH, and CPFD on the same number of processors.

The results indicate that the quality of solutions found by the three list scheduling algorithms does not change much with more processors. DSH and CPFD finds better solutions to Problems P3, P5, and P6 on ten processors than those on four processors. None of these algorithms can further improve the solutions to any task graphs when we have 20 or 50 processors.

Table 3.6: Minimum makespan found by ISH, DSH, CPFD, and GA-based scheduling algorithm on ten-processor systems.

Test Problem	ISH	DSH	CPFD	GA		
				Best	Average	95% CI
P1	300	300	300	300	300	0
P2	600	400	400	400	449.2	6.36
P3	260	240	240	260	263.2	2.05
P4	400	300	330	350	371.6	5.50
P5	761	438	438	361	383.53	10.70
P6	41	36	36	36	36.66	0.16
P7	450	320	320	350	377.8	3.81
P8	760	760	760	760	766.4	3.87
P9	1220	1030	1040	1040	1082.4	12.04

The performance of GA, however, degrades on several problems as the number of processors increases. Given more processors, GA tends to find solutions in which tasks are evenly distributed among processors, which is not favorable as the unnecessary communication costs between dependent tasks may result in a longer makespan. Nevertheless, the overall performance of GA is still comparable to DSH and CPFD. GA improves the solution to Problem P5 and still outperforms the other algorithms on this problem.

3.4.4.2 Performance Evaluation on a simulation environment

We evaluate the performance of the semi-static scheduling approach on a simulation environment. We use the 14-node LU Decomposition task graph shown in

Table 3.7: Minimum makespan found by ISH, DSH, CPFD, and GA-based scheduling algorithm on twenty-processor systems.

Test Problem	ISH	DSH	CPFD	GA		
				Best	Average	95% CI
P1	300	300	300	300	300	0
P2	600	400	400	440	468.8	5.03
P3	260	240	240	260	267.2	2.69
P4	400	300	330	350	396.6	6.62
P5	761	438	438	361	413.06	21.39
P6	41	36	36	36	36.78	0.16
P7	450	320	320	360	385.6	4.42
P8	760	760	760	760	770	4.19
P9	1220	1030	1040	1050	1133.4	13.24

Figure 3.2 in the experiment. All tasks in this task graph are executed multiple times. In the baseline setting, the computational time of tasks and the data transfer times among tasks are the same as shown in Figure 3.2 and we assume there are four fully connected computing nodes with the same processing ability. In each loop, the execution time of a task and the data transfer time between each pair of dependent tasks have a 30% chance of doubling and a 20% chance of tripling the time in the baseline settings. Likewise, each processor has a 30% chance of doubling and a 20% chance of tripling its processing times. The above settings of the computing environment in one loop are independent of the other loops during the course of the computation. The changes on the problem settings may cause an optimized schedule fail to produce consistently good results over different loops. The scheduling algorithm is invoked and a best schedule is pro-

Table 3.8: Minimum makespan found by ISH, DSH, CPFD, and GA-based scheduling algorithm on fifty-processor systems.

Test Problem	ISH	DSH	CPFD	GA		
				Best	Average	95% CI
P1	300	300	300	300	300	0
P2	600	400	400	440	512.8	9.01
P3	260	240	240	260	276.4	2.43
P4	400	300	330	350	412.4	7.33
P5	761	438	438	369	512.88	21.02
P6	41	36	36	36	37.36	0.20
P7	450	320	320	380	407.6	4.47
P8	760	760	760	760	785.2	4.79
P9	1220	1030	1040	1050	1182.4	15.03

duced in each loop, either selected from a group of existing schedules or created from scratch.

We run two cases to evaluate the performance of the semi-static approach. In the first case, we run the scheduling algorithm only for the first loop. For the rest of the loop, a best schedule is selected from the exiting solutions (i.e., with the highest fitness). In the second case, we rerun the scheduling algorithm in each loop. In both cases, we run five loops and compare the fitness of the best schedule produced in each loop. Table 3.9 shows the results.

Apparently, rerunning the scheduling algorithm for each loop produces better results than simply selecting a solution from existing ones, because an optimal solution for a previous problem condition may not fit for a new problem condition.

Table 3.9: The fitness of the best solution for each loop of task execution, results produced from two scheduling algorithms: with and without the semi-static approach.

Algorithm	Loop 1	Loop 2	Loop 3	Loop 4	Loop 5
Scheduling without the Semi-static Approach	0.918	0.941	0.928	0.936	0.947
Scheduling with the Semi-static Approach	0.956	0.965	0.965	0.968	0.969

Nevertheless, the semi-static approach still produces fairly good solutions, with an average fitness of 0.934.

We expect that increasing the diversity of solutions in a population can improve the performance of the semi-static approach because a diversified population contains more resources for a variety of problem conditions. We conduct additional experiments in which we vary the crossover and mutation rates of the scheduling algorithm. We test four crossover rates: 0.4, 0.6, 0.8 (the baseline setting), and 1.0, and five mutation rates: 0.001, 0.005 (the baseline setting), 0.01, 0.02, and 0.05. For each test case, we run the scheduling algorithm 50 times using the semi-static approach, and each run consists of 10 loops. We study the effect of these two parameters on the performance of the approach. The performance is measured by the average fitness of solutions and the percentage of times that a new solution is selected for a current problem condition (also called a solution switch). Figure 3.14 and 3.15 show the results.

The results indicate that the performance of the semi-static approach is sensitive to the mutation rate. The experiments with a mutation rate of 0.02 produce the best results. As the mutation rate decreases, the probability of finding a

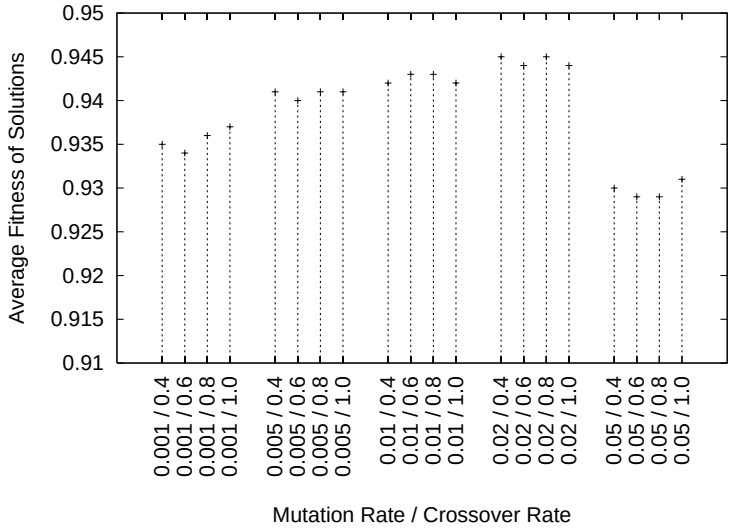


Figure 3.14: The average fitness of the solutions produced by the semi-static scheduling approach using different crossover and mutation rates.

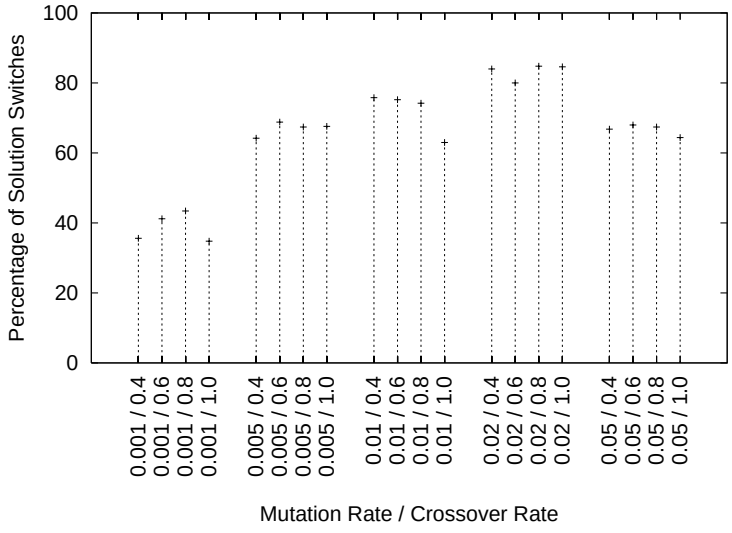


Figure 3.15: The percentage of times that a solution switch occurs during the execution of the semi-static scheduling approach using different crossover and mutation rates.

better solution for a current problem condition decreases as well. As a result, the runs with low mutation rates cannot produce as good solutions as the ones using mutation rate of 0.02. On the other hand, a mutation rate of 0.05 is too high for the scheduling algorithm to produce consistently good solutions. Compared to the mutation rate, the crossover rate has much less impact on the performance of the approach and thus its effect can be ignored.

CHAPTER 4

PLAN SWITCHING

A large-scale distributed system is a dynamic computing environment. The availability of resources and services to a computing task typically changes quickly over the course of the computation. As a result, the success of plan execution cannot be guaranteed. There are two basic strategies to overcome this uncertainty: replanning and plan switching. Replanning, as an online strategy, is a process of either creating a new plan or adapting the existing plan to the current conditions of the computing environment (by using resources or services currently available to the system). Replanning introduces additional time for execution, as the process of replanning and executing a computing task cannot be overlapped. Plan switching, as an offline strategy, attempts to build a family of alternative plans in advance. When the current plan fails in execution, we find an alternative plan that can continue the execution of the computing task and migrate the execution directly from the current plan to the alternate. If plan switching is successful, we can continue the execution without replanning.

A request for plan switching is invoked when failure occurs during the execution of a computation. The procedure of plan switching consists of four steps. First, we retrieve the current state of the execution of the computation. Second, we look for a backup plan so that the computation may continue from that plan and still reach the goal of the computation. Third, if such a plan can be found,

we call the scheduling service for a new execution schedule based on the backup plan. Finally, we move all data related to the computation from the current nodes to the scheduled nodes and continue the computation. The execution of the backup plan may fail again due to the dynamics of the computing environment. As a result, we may need to perform plan switching multiple times during the execution of a computation.

This chapter addresses the problem of plan switching and presents an approach to this problem. The main idea of the approach is to locate the execution points from alternative plans for a given task in parallel with the execution of the current plan. When the execution cannot proceed, we continue the execution of a computing task from a selected execution point in another plan. The process of finding execution points has a relatively small computational cost and can be performed in parallel with the execution of the computation. Therefore, this approach does not necessarily increase the execution time of a computing task.

4.1 Problem Formulation

4.1.1 Assumptions

We make the follow assumptions before formulating the problem of plan switching.

1. A plan is a directed graph whose vertices are the atomic activities and directed arcs denote data and control flow dependencies. Concurrent activities are allowed, but iterative execution of activities is not allowed. Only

independent activities may be executed concurrently; no communication among concurrent activities is allowed.

2. A family of plans are created in advance. One of the members of the family, the one selected for execution, is called the *current plan*. All other plans serve as backups but still have a chance of execution when the current plan fails. These plans are called *alternative plans*.

3. Once an activity in a plan is executed, the success of its execution is guaranteed. If, however, an activity cannot be executed, we may either wait until it can be executed or switch the execution of the current plan to an alternative plan.

4.1.2 Definitions

We now provide several definitions necessary to formulate our plan switching problems. The term "snapshot" has been used to determine the progress of multiple processes running on distributed systems [110]. We use the same term to determine the progress of a plan execution. We next introduce the concept of "congruent snapshot," which is the basis of the plan switching approach.

Definition 7. A *single snapshot* is a partial description of the progress of plan execution. A single snapshot can be defined on either a pair of consecutive activities $\{a, b\}$ in a plan, denoting that activity a has finished execution while activity b is still pending, or between an activity and a dummy activity, if the activity has no precedent or subsequent activities.

We can annotate a plan by adding single snapshots in three cases: 1) between every two consecutive activities, 2) before all activities that have no precedent

activities, and 3) after all activities that have no subsequent activities. Figure 4.2 shows an annotated version for the plan in Figure 4.1.

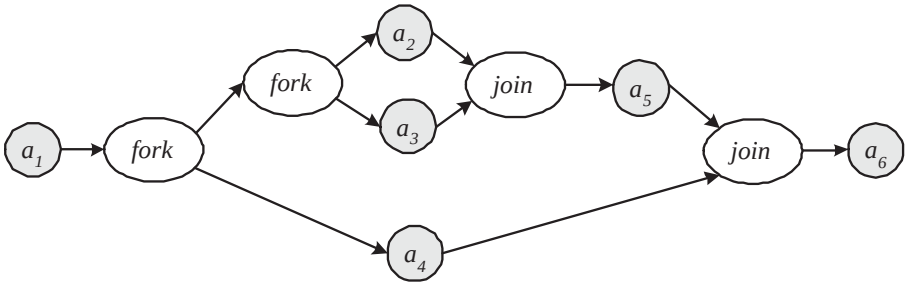


Figure 4.1: An example plan that contains six activities.

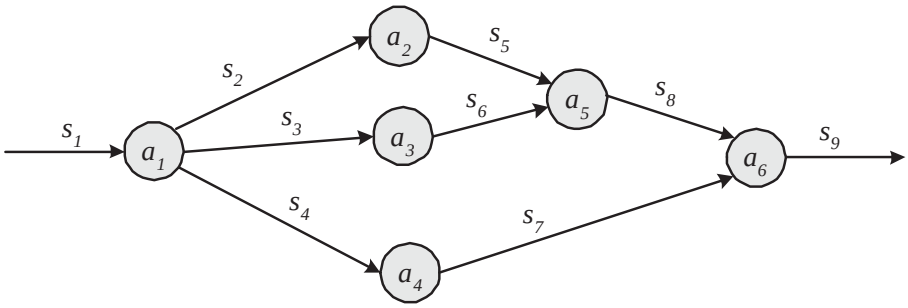


Figure 4.2: An annotated version of the plan shown in Figure 4.1. Nine single snapshots are added.

Definition 8. The *subsequent activities* of a single snapshot are the set of all activities that should be executed after the snapshot is reached. We use the function $subs(s)$ to denote the subsequent activities of a given single snapshot s. For instance, in Figure 4.2, $subs(s_1) = \{a_1, a_2, a_3, a_4, a_5, a_6\}$, $subs(s_5) = \{a_5, a_6\}$, and $subs(s_9) = \phi$. The *preceding activity* of a single snapshot is a set that contains the most recent executed activity, if it exists, before the snapshot is reached. We use the function $prec(s)$ to denote the preceding activity of a given

single snapshot s. For instance, in Figure 4.2, $prec(s_1) = \phi$, $prec(s_5) = \{a_2\}$, and $prec(s_9) = \{a_6\}$.

Definition 9. A pair of single snapshots, s_a and s_b, is independent if $prec(s_a) \cap subs(s_b) = \phi$ and $prec(s_b) \cap subs(s_a) = \phi$. For instance, snapshots s_2 and s_3 in Figure 4.2 are independent snapshots, while snapshots s_2 and s_5 are not. A set of single snapshots S is independent if every pair of single snapshots in S is independent. For instance, in Figure 4.2, $S = \{s_2, s_3, s_4\}$ is a set of independent snapshots.

Definition 10. A *composite snapshot* is a combination of single snapshots. We use a pair of square brackets "[" and "]" to denote the operation of combining single snapshots. For instance, $[s_2, s_3]$ denotes a composite snapshot that combines s_2 and s_3. A composite snapshot may contain single snapshots that are not independent of each other.

The above notions for a single snapshot can also be applied to composite snapshots. The *subsequent activities* of a composite snapshot are the union of the sets of subsequent activities of all single snapshots. The *preceding activities* of a composite snapshot are the union of the sets of preceding activities of all single snapshots. Two composite snapshots, s_a and s_b, are independent if $prec(s_a) \cap subs(s_b) = \phi$ and $prec(s_b) \cap subs(s_a) = \phi$. For instance, in Figure 4.2, $subs([s_2, s_3]) = \{a_2, a_5, a_6\} \cup \{a_3, a_5, a_6\} = \{a_2, a_3, a_5, a_6\}$, $prec([s_2, s_3]) = \{a_1\}$, and snapshots $[s_2, s_3]$ and s_4 are independent.

Definition 11. A snapshot is *consistent* if it is either a single snapshot or a composite snapshot of a set of independent snapshots. A consistent snapshot s in a plan P is a *global consistent snapshot* if there does not exist a single snapshot s' in P such that s' is not included in s and s' is independent of all snapshots in s. In contrast to a single snapshot, a global consistent snapshot gives a complete view

of the status of a plan execution. For instance, the composite snapshot $[s_2, s_3, s_4]$ in Figure 4.2 is a global consistent snapshot. This snapshot describes a status of plan execution in which activity a_1 has finished execution while activities a_2, a_3, and a_4 are pending execution followed by a_5 and a_6.

Definition 12. A global consistent snapshot s_1 in plan P_1 is *congruent* to a global consistent snapshot s_2 in plan P_2, if we are able to switch the execution from s_1 to s_2, execute the subsequent activities of s_2, and finish the computing task. We use the symbol "$\sim$" to denote the relation of congruence. If a snapshot s_1 is congruent to s_2, $s_1 \sim s_2$. In contrast to the notion of congruence in mathematics, the relation of congruence between a fair of snapshots is not reflexive, i.e., if $s_1 \sim s_2$, $s_2 \sim s_1$ may not hold. The relation of congruence is not transitive, either, i.e., if $s_1 \sim s_2$ and $s_2 \sim s_3$, $s_1 \sim s_3$ may not hold.

Definition 13. The *optimal congruent snapshot* for a given snapshot is the one whose subsequent activities incur minimal execution cost among all congruent snapshots.

4.1.3 Plan Switching between Congruent States

We formulate the problem of plan switching as follows: if the execution of the current plan P_{curr} cannot proceed from a global consistent snapshot s, find an optimal congruent snapshot s' of s from alternative plans and continue the execution from s' in the plan to which s' belongs. Figure 4.3 shows an example of switching between two plans, P_1 and P_2. Initially, P_1 is the current plan. When the execution of P_1 cannot continue in snapshot $[s_5, s_6, s_7]$, a congruent snapshot $[s_4', s_5']$ in Plan P_2 is found, and the plan execution is switched to P_2 from this

congruent snapshot. When the execution finishes, the complete set of activities having been executed is $\{a_1, a_2, a_3, a_4\}$ from P_1 and $\{a_4', a_5', a_6'\}$ from P_2.

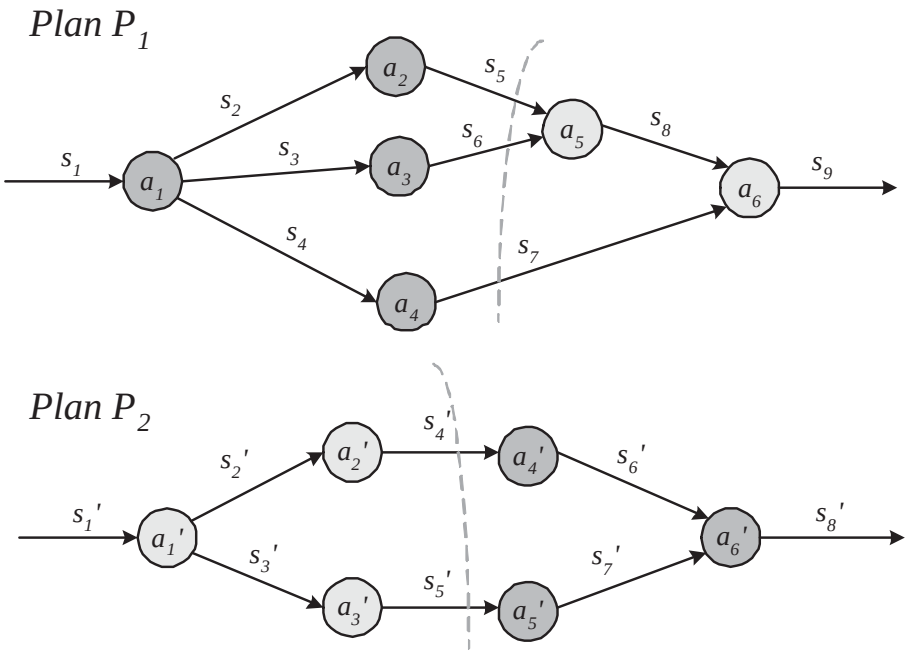


Figure 4.3: An example of execution switching between two plans. Snapshot $[s_4', s_5']$ in Plan P_2 is congruent to snapshot $[s_5, s_6, s_7]$ in Plan P_1.

4.2 Algorithm Design

The process of finding the optimal congruent snapshot for a given snapshot consists of three steps: 1) generate the set of global consistent snapshots for each plan, 2) identify congruent snapshots from the set of global consistent snapshots, and 3) choose the optimal congruent snapshot.

1. Finding Global Consistent Snapshots

To find the set of all global consistent snapshots for a plan, we first set S to be the set of all single snapshots in the plan. Then we repeat the steps in the following pseudo code to update the snapshots in S until S cannot be further updated. The final set S is the set of all global consistent snapshots of a plan.

```
Begin.
1. For each snapshot s in S, find all single snapshots that are
   independent of s.
2. If s has at least one independent snapshot, do
   a. Remove s from S.
   b. For each of its independent snapshot s', do
      (1) Combine s with s'.
      (2) If the combined snapshot is not in S, include it in S.
   c. End for.
3. End if.
End.
```

2. Locating Congruent Snapshots for a Given Snapshot

   Locating congruent snapshots from alternative plans allows a plan executor to easily switch task execution from the current plan to an alternative plan, when temporary or permanent failure occurs in current plan execution. During the execution of the current activity(ies), we try to find the optimal congruent snapshot for the snapshot after the execution of the current activity(ies) finishes. We show the pseudo code for locating congruent snapshots as follows.

```
Begin.
```

```
1. P-curr = the currently executing plan.
2. A-curr = the set of currently executing activities.
3. s-curr = the global consistent snapshot after all activities in
           A-curr finishes.
4. S = {}. /* initially, the set of congruent snapshots is empty */
5. For each alternative plan P, do
   a. For each global consistent snapshot s in P, do
      (1) If subs(s) can be executed from s-curr and the execution
          of subs(s) satisfies all goals for the computing task,
          include s in S.
   b. End for.
6. End for.
End.
```

3. Choosing the Optimal Congruent Snapshot

   Once we have identified all congruent snapshots in step two, we are able to estimate the computational cost of all subsequent activities for each congruent snapshot. If the computational costs of activities are not given or are difficult to estimate, a rough estimation which simply counts the number of subsequent activities for a congruent snapshot is applied. The optimal congruent snapshot is the one that incurs the lowest computational cost among all congruent snapshots.

What should we do if there does not exist a congruent snapshot when plan switching is requested? There are three options: 1) send out a request for replanning; 2) terminate the execution of the current plan completely and execute an alternative plan from the beginning; and 3) roll back the execution of the cur-

rent plan to the previous global consistent snapshot and try to find a congruent snapshot for that snapshot. We discuss the third option in detail.

Rollback is a process of backtracking the computation to a previous saved point when a failure occurs, so that the whole computation does not have to be resumed from the beginning [111]. We say an activity is *reversible* if we can roll back the execution of the activity completely to the snapshot right before it is executed. In order to roll back the execution of activities, we need to record every global consistent snapshot that has been reached and an ordered list of all activities that have been executed in the current plan. When a plan execution cannot proceed and there is not a congruent snapshot for the current snapshot, we try to roll back the execution of the last executed activity. If the activity is not reversible, we have to choose one of the first two options, either perform replanning or choose another plan to execute from the beginning. Otherwise, we roll back the execution of the activity, regress the computation to the previous snapshot, and attempt to find a congruent snapshot for this snapshot. If a congruent snapshot exists, we switch the execution of the computation to another plan. If, however, a congruent snapshot is still unavailable, we repeat the preceding steps and roll back the execution of previous activities, until we have successfully reached a snapshot that has a congruent snapshot, or the execution of the computation cannot be further rolled back.

4.3 Simulation Study

4.3.1 Environment Design

We perform a simulation study to evaluate the effectiveness of plan execution. The simulation environment consists of a number of randomly generated plans. Both the sizes of plans (denoted by the number of activities) and the number of subsequent activities of each activity in a plan may be different but follow Gaussian distributions. A maximum ten plan switches can occur during the computation of an entire task. If this limit is reached, we terminate the process and mark the plan execution as a failure. We also assume that all activities have the same computational costs.

We test different cases by varying the success rate of activity execution, the probability of a global consistent snapshot being a congruent snapshot, and the number of available plans to a computing task. We test each case ten times and evaluate the performance with two criteria: the success rate of plan execution (i.e., the number of runs out of ten runs that a computation task can successfully finish) and, for the successful runs, the average number of plan switchings performed. Table 4.1 lists the parameter settings for the experiment.

4.3.2 Simulation Results

We first test the case in which there are three available plans and the probability of congruent snapshots is fixed at 0.1. We vary the success rate of a computing activity between 0.4 and 0.9. No activity is allowed to roll back its execution.

Table 4.1: Parameter settings for the experiment.

Parameter	Value
Number of Runs	10
Number of Plans	3, 6
Avg. Number of Activities	10
Maximum Number of Plan Switches	10
Avg. Number of Subsequent Activities	2
Prob. of an Successful Activity	0.4, 0.5, ..., 0.9
Prob. of a Congruent Snapshot	0.01, 0.05, and 0.1

Figure 4.4 shows the number of successful runs in each case and the minimum, average, and maximum number of plan switchings in the successful runs. The results indicate that a lower success rate of computing activities increases the possibility and occurrences of plan switchings. As a congruent snapshot cannot always be found when plan switching is requested, a lower success rate of computing activities results in a higher probability of failure in plan execution.

Next, we evaluate the impact of the probability of congruent snapshots on the success of plan switching. We test the cases in which only 1% and 5% of the global consistent snapshots are congruent snapshots. Again, the success rate of a computing activity is set between 0.4 and 0.9, and no activity is allowed to roll back its execution. The simulation results, shown in Figures 4.5 and 4.6, indicate that the probability of congruent snapshots has profound effect on the success of plan switching. A lower probability leads to a lower possibility of finding a congruent snapshot, and thus reduces the success rate of plan execution. We noticed that when the probability of congruent snapshots is reduced to 0.01, the failure of plan execution, in most cases, is due to inability to find congruent snapshots

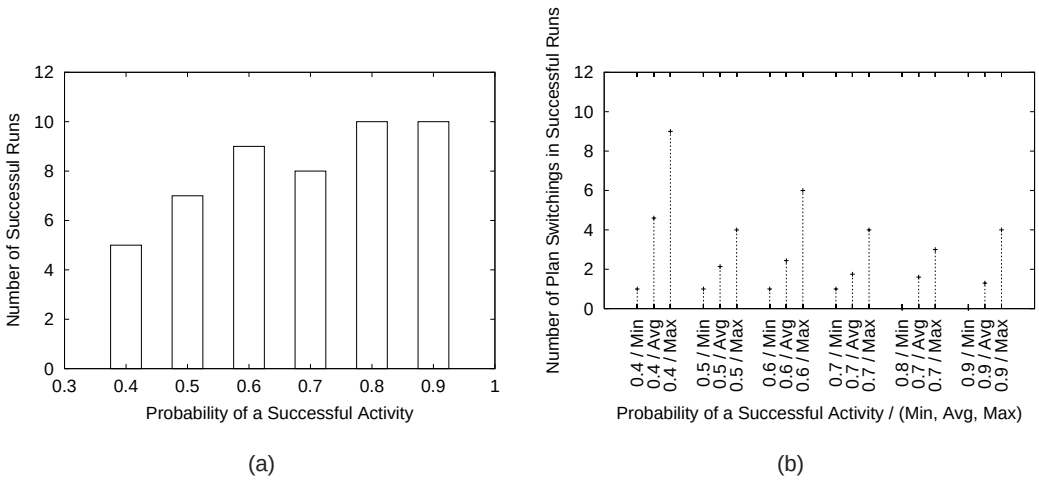


Figure 4.4: The simulation results on the effect of the success rate of a computing activity to the success of plan switching. (a) The number of successful runs out of ten runs. (b) The average, minimum, and maximum number of plan switches in successful runs.

rather than overreaching the maximum allowed number of plan switches. This result indicates that allowing rollback of plan execution might be an effective cure to plan execution when the probability of congruent snapshots is low.

Figure 4.7 shows the results in cases when rollback of plan execution is enabled and all activities in a plan are reversible. The probability of congruent snapshots is kept as low as 0.01. Obviously, allowing rollback of activity execution gives more opportunities for finding congruent snapshots and thus increases the probability of a successful plan switch. Comparing Figures 4.6(a) and 4.7(a), only for some of the different probabilities of a successful activity does rollback improve the number of successful runs. However, Figure 4.8 indicates that in those failed runs, allowing rollback offers more chances for plan switching among alternative plans. Some runs fail solely because a maximum number of plan switches have already been attempted.

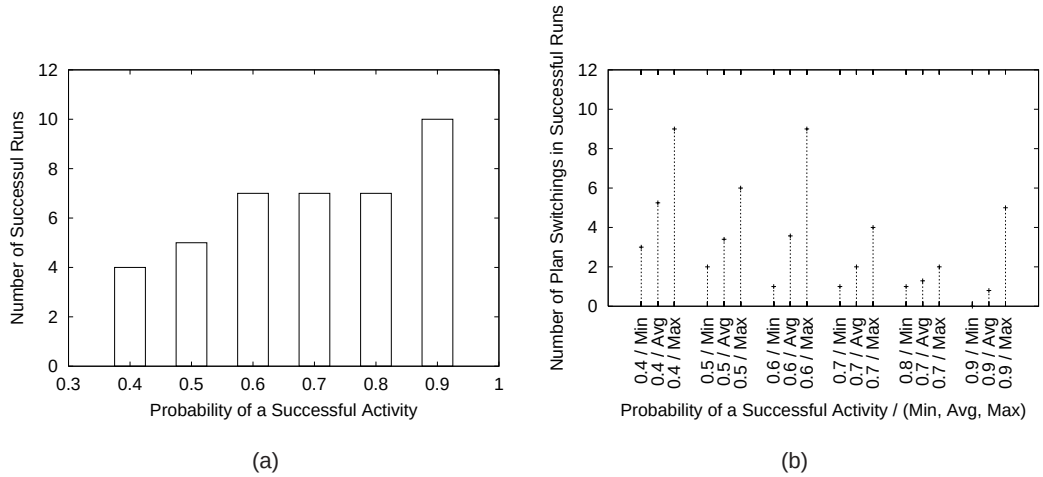


Figure 4.5: The simulation results when 5% of the global consistent snapshots are congruent snapshots. (a) The number of successful runs out of ten runs. (b) The average, minimum, and maximum number of plan switches in successful runs.

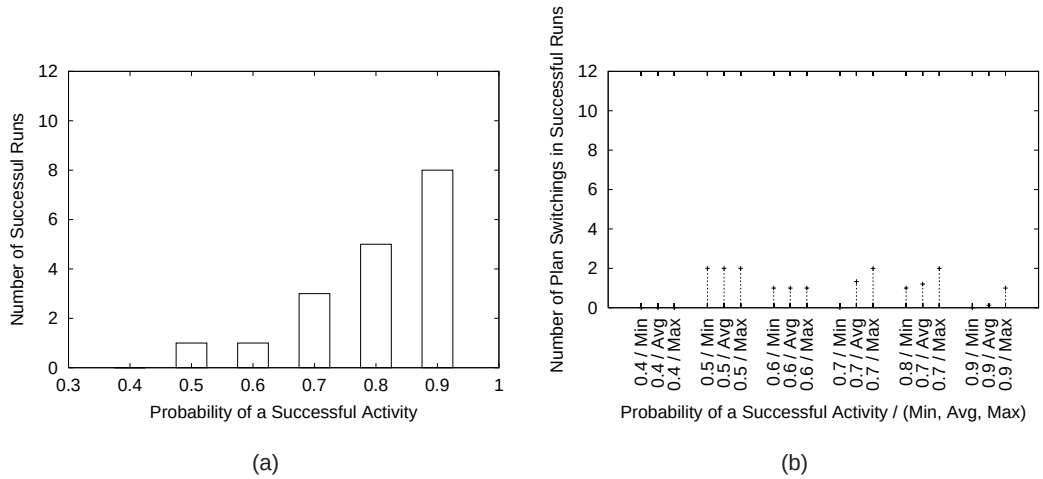
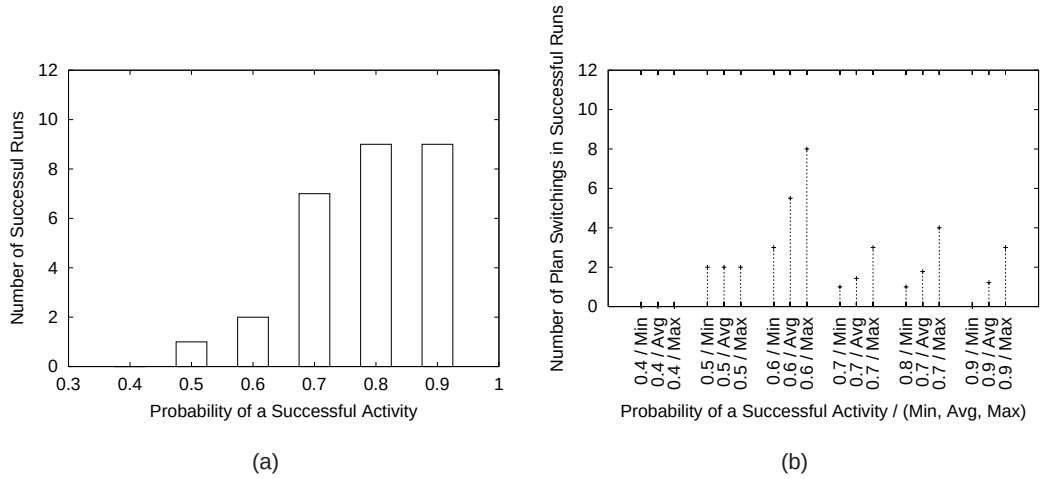
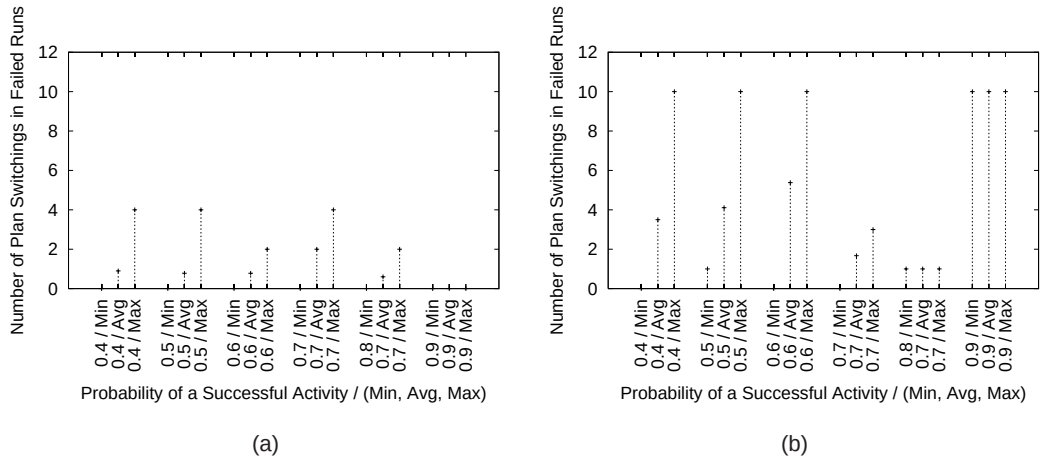


Figure 4.6: The simulation results when 1% of the global consistent snapshots are congruent snapshots. (a) The number of successful runs out of ten runs. (b) The average, minimum, and maximum number of plan switches in successful runs.



(a) (b)

Figure 4.7: The simulation results showing the effectiveness of allowing rollback in plan execution when all activities are reversible and 1% of the global consistent snapshots are congruent snapshots. (a) The number of successful runs out of ten runs. (b) The average, minimum, and maximum number of plan switches in successful runs.



(a) (b)

Figure 4.8: The minimum, average, and maximum number of plan switches before the plan execution fails. (a) Rollback of execution is not allowed. (b) Rollback of execution is allowed.

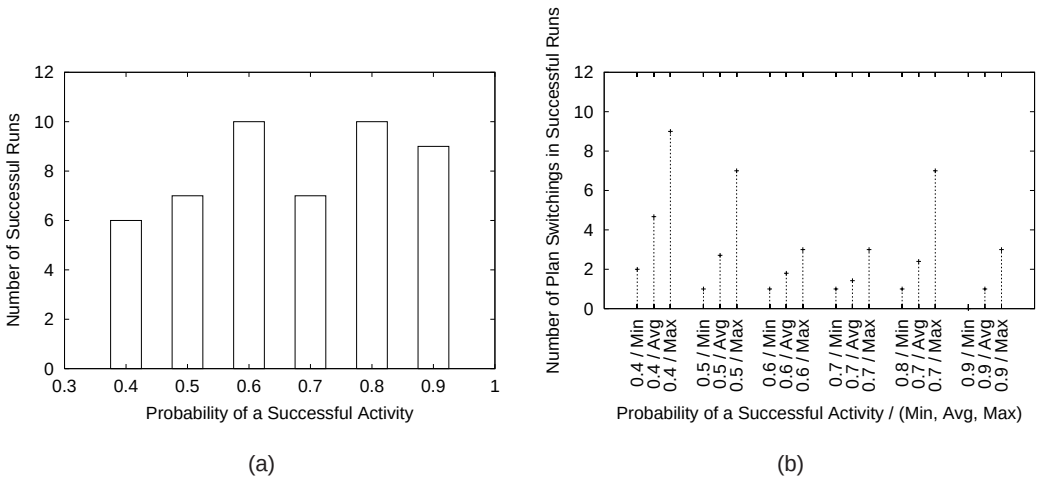


Figure 4.9: The simulation results for cases in which six plans are available for execution and 5% of the global consistent snapshots are congruent snapshots. (a) The number of successful runs out of ten runs. (b) The average, minimum, and maximum number of plan switches in successful runs.

In the third case, we study whether the number of alternative plans affects the success of plan switching. We repeat the above tests by setting the probability of congruent snapshots to 0.05 but increase the number of plans to six. Activities are not allowed to roll back their execution. The simulation results, shown in Figure 4.9, demonstrate that having more alternative plans definitely improves the performance of plan switching. When there are six available plans, the success of plan switching is less likely to rely on either the success rate of activity execution or the probability of congruent snapshots, as more global consistent snapshots (hence more congruent snapshots) are available from alternative plans.

4.4 Concluding Remarks

The success of a plan execution cannot be guaranteed due to the dynamics of a large-scale distributed system. Plan switching is a strategy to continue the execution of a computation when failure occurs during the execution of a plan. Given a family of plans available to perform a computing task, the function of plan switching is to switch the execution of a computing task from one plan to another plan. When a plan switching is successfully performed, we reschedule the execution on the new plan and continue the execution of the computing task.

We formulate the problem of plan switching in a large-scale distributed system and present an approach to the problem. This approach introduces the concept of congruent snapshots that allow transition from the execution of one plan to another. The main idea of the approach is to find congruent snapshots from alternative plans during the execution of the current plan. When the execution of the current plan fails, we continue the execution of the task from an optimal congruent snapshot in another plan.

A simulation study on this approach indicates that a high probability of congruent snapshots, a high success rate of computing activities, and more alternative plans can improve the performance of plan switching. In addition, allowing rollback of activity execution offers additional opportunities to find congruent snapshots, thus is also benefiting plan switching.

CHAPTER 5

SUMMARY OF WORK AND CONCLUSIONS

5.1 Summary of Work

Planning and scheduling are among the essential functions for executing a complex computing task in large-scale distributed systems. The execution of a complex computing task typically requires a large amount of computing resources and has a long execution time. There are cases in which the execution of a complex computing task cannot be supported by a single computing service available in a large-scale distributed system. Instead, we need to execute multiple computing services to produce the expected results of a computing task. The function of planning is to produce a plan that specifies the computing services to be executed, the order of their execution, and the data dependencies among them. After a plan is available, we need scheduling to assign the execution of each computing service in a plan to a computing node in the system so that the total execution time of a computing task is minimized. A large-scale distributed system is dynamic and the conditions of a system, such as the availability of computing resource needed to execute a computing task, may change during the course of planning and scheduling. As a result, the success of a computing task cannot be guaranteed. We introduce a method called plan switching to improve the success of a computing task. This method allows for the switching of the execution of a

computing task from one plan to another plan so that the execution may continue without the need to create a new plan. After plan switching is performed, we need another round of scheduling and then continue the execution of the new plan until a computing task is successfully completed, or a failure cannot be avoided.

In this dissertation, we address the problems of planning and scheduling for large-scale distributed systems. Planning and scheduling for large-scale distributed systems are complicated as both problems require the search in a large solution space and in a changing problem environment due to the complexity of computing tasks and the scale and dynamics of the system. Deterministic approaches have been applied to planning and scheduling problems. These approaches typically use knowledge extracted from the problems to reduce the search space, and thus are efficient search approaches. However, knowledge from one problem may not be applied to other problems. As a result, these approaches may not be applicable to a variety of planning and scheduling problems. In addition, the knowledge used for the search may not be applicable as the problem conditions change. We investigate a GA approach to planning and scheduling for large-scale distributed systems. GAs have been applied to solve difficult search and optimization problems. GAs also exhibit quick adaptation to changing problem environments. Therefore, GAs can be a good candidate approach to the addressed problems.

We make the following five aspects of contributions from our work: a) applying genetic algorithms to planning for large-scale distributed systems; b) applying genetic algorithms to scheduling for large-scale distributed systems; c) introducing the method of plan switching; d) studying the variable length representation and the incremental search strategies in evolutionary computation; e) designing

an intelligent agent-based middleware for large-scale distributed systems. We summarize the work for each aspect as follows.

a) *Applying genetic algorithms to planning for large-scale distributed systems*

We first attempt to apply GA to traditional AI planning problems. Our algorithm is extended from the traditional GAs in two aspects. First, we use an indirect encoding method to remove invalid operators in a plan. Second, we use an incremental search strategy that divides the search for solutions into multiple individual phases. Experiments on two domains, the Towers of Hanoi and the Sliding-tile puzzles, show that this approach can solve the 6-disk Towers of Hanoi problem and the 3×3 Sliding-tile puzzles. As the problem size increases, the search performance of this GA degrades very quickly. GA cannot find a solution to the 7-disk Towers of Hanoi problem in any of the 50 runs in our experiment, and GA only finds solutions to the 4×4 sliding tile puzzles in 5 out of 50 runs. We believe the search performance of this approach can be improved if we use a more accurate fitness function for solution evaluation. Therefore, more knowledge extracted from the planning problem is needed for an accurate estimation of the distance between the current state and the goal specifications. This result indicates that domain knowledge is still very important to improve the performance of this GA-based planning algorithm.

We also introduce a search strategy, called recursive subgoals, to the Sliding-tile puzzles. This strategy divides the goal of the problem into a group of subgoals and specifies the order of these subgoals such that reaching one subgoal can reduce a problem to the same problem at a smaller size. We show that this strategy can be incorporated easily in our planning algorithm as we can assign a specific subgoal for each individual phase. After the subgoal is reached in one phase, we assign the next subgoal for the subsequent phases. With the recursive subgoal

strategy, we can consistently find valid solutions to the 7×7 Sliding-tile puzzles, a significant improvement compared to the results without using this strategy. The applicability of the recursive subgoal strategy, however, is restricted to a limited set of problems. It is also very difficult to determine the applicability of this strategy to a given planning problem.

We modify the GA-based algorithm to planning for large-scale distributed systems. We classify the deterministic and non-deterministic planning problems. A planning problem is non-deterministic if we do not have the complete knowledge regarding the state of the system at some stages of plan execution. For non-deterministic planning problems, we need a non-linear structure for plans to allow for the conditional and iterative execution of activities so that the execution of a plan can still satisfy the goals of a computing task despite the uncertainties of the system state. The modified algorithm can evolve a non-linear structure for plans. We build a simulation environment to evaluate the scalability of this algorithm. This simulation environment allows us to easily configure the scale of a system by varying the number of processing nodes and the number of end-user services. We also simulate complex computing tasks by randomly generating task graphs for a given size (i.e., the number of computing activities for performing a computing task). Our results show that GA needs a larger population and more generations to maintain its search performance as the problem size increases. As a result, the execution time of GAs increases very quickly as well.

b) *Applying genetic algorithms to scheduling for large-scale distributed systems*

We first study the problem of multi-processor scheduling and apply a GA-based algorithm to this problem. Comparing to the previous GA-based scheduling approaches, our GA uses a more flexible representation method that gives GA a complete exploration of the search space. We also use an incremental fit-

ness function that encourages the formation of partial solutions found during the search. We conclude that, given sufficient computing resources, this algorithm is able to find comparable and sometimes better schedules (i.e., with shorter execution times) than deterministic algorithms such as ISH, DSH, and CPFD. The GA-based algorithm, however, is much less efficient than the deterministic algorithms. We perform additional experiments to evaluate the performance of GA on a system with a larger number of processing nodes. The results show that, given the same population size and number of generations, GA finds worse solutions to most tested problems when the number of processing nodes increases, because finding a schedule for a large system requires the search in a larger solution space. The largest system in our experiment contains 50 processing nodes.

The experiments on heterogeneous and dynamic computing environments show the strength of this GA: the algorithm can be applied easily to other scheduling problems and it adapts quickly to the changes in the computing environment. In the experiment on a heterogeneous computing system, GA finds considerably better solutions than the deterministic algorithms. This study demonstrates that the GA-based algorithm can adapt quickly in a large-scale distributed system, a natural dynamic computing environment.

We extend the above algorithm to scheduling for large-scale distributed systems. This algorithm can schedule non-deterministic activity graphs that contain activities whose execution can only be determined by the results of preceding activities. We schedule the execution of non-deterministic activity graphs in multiple steps. In each step, we only schedule the activities whose execution can be determined at the current step. After the scheduled activities finish, we retrieve the computing results, determine the subsequent activities to be executed, and perform another round of scheduling. We may schedule the same activity graph

multiple times during the execution of a computing task. Reusing the existing schedules can save the computation time of the scheduling algorithm. We present a semi-static approach to scheduling for large-scale distributed systems. This approach stores the schedules that have been evolved during the execution of a computing task. When a schedule on the same activity graph is requested, we evaluate all corresponding schedules and select the best one as the solution. As a large-scale distributed system is a dynamic system, a best schedule may not always be the best in the later scheduling steps. The solutions evolved by GA provide diverse resources which allow this approach to adapt to the changes of the computing system. Our study shows that a diverse population of a GA is beneficial to this approach as it contains solutions to more problem conditions than a converged population. A higher mutation rate typically produces a more diverse population. In our experiment, a mutation rate of 0.02, which is higher than the one used in other experiments, exhibits the best performance for this semi-static approach.

c) *Introducing the method of plan switching*

We introduce plan switching as a method for recovering from failures during the execution of a computing task. This method assumes that there are a family of plans available to perform a computing task. Only one plan is selected to execute each time, and the other plans serve as backup plans. When failure occurs, we can switch the execution of a computing task from one plan to a selected backup plan. If plan switching is successful, we do not need replanning.

We introduce the concept of "congruent snapshot" in our approach to plan switching. The main idea of the approach is to find the optimal congruent snapshot from backup plans in parallel with the execution of the current plan. When plan switching is needed, we can directly switch the execution to a new plan

and continue to execute that plan from the optimal congruent snapshot. The simulation study indicates that plan switching can improve the success of a computing task (with higher probability of successfully finishing a computing task), and the effectiveness of this approach depends on the number and availability of congruent snapshots from the backup plans: more congruent snapshots result in a higher probability of successful plan switching.

d) *Studying the variable length representation and the incremental search strategies in evolutionary computation*

We use the variable length representation to encode solutions in both the planning and scheduling algorithms for the following two reasons. First, it is very difficult to estimate the size of the optimal solutions to both problems. As a variable length representation allows a GA to evolve various sizes of solutions in a population, it turns out to be a more suitable representation method than the traditional fixed length representation. Second, both planning and scheduling require the search in a changing computing environment. As a result, the size of the optimal solutions may also change during the course of planning and scheduling. A variable length representation enables a GA to adjust the size of solutions to the changes of the problem environment, so it adapts better than fixed length representation to problem changes.

In addition, we study the effectiveness of using incremental search strategies in a GA. In the planning algorithm, we use an incremental method by dividing the search into multiple independent phases and the final solution is the concatenation of solutions found in each phase. The experiments on the Towers of Hanoi show that a multi-phase GA outperforms the traditional single-phase GA. A multi-phase GA finds solutions in all 50 runs to the 6-disk problem, while the single-phase GA does not. Although the multi-phase GA cannot find a valid solution

to the 7-disk problem, it reaches higher goal fitness than the single-phase GA. In the scheduling algorithm, we use an incremental fitness function to reward partial solutions found during the search. We evaluate the effectiveness of this fitness function by testing the algorithm in which the contributions of partial solutions are ignored in the fitness function. In this test, GA is unable to consistently find valid solutions over multiple runs. We believe that the incremental fitness function is an important factor to the success of the scheduling algorithm.

e) *Designing an intelligent agent-based middleware for large-scale distributed systems*

We design an intelligent middleware for large-scale distributed systems. "Intelligence" means more system automation and less user intervention. Two features in our design contribute to the intelligence of this middleware: a multi-agent framework and ontology-based knowledge sharing among agents.

The function of a middleware is supported by a group of services. We classify two classes of services: core services and end-user services. Core services are the system-wide services that provide coordinated and transparent access to computing resources in a large-scale distributed system. Core services are the essential component of a middleware and must be persistent and reliable. Planning, scheduling, and plan switching are among the core services included in the middleware. End-user services, on the other hand, are specialized services offered by autonomous service providers that perform the actual computing service for users. End-user services are transient and can be removed by the service providers at any time.

The execution of each service in the middleware is supported by an autonomously running software agent. We assign each agent to perform the role of a pre-specified service. Each agent stores the essential knowledge to perform its

designated roles. All agents must work coherently to achieve the overall function of a middleware. In order for these agents to cooperate and share knowledge with each other, they must share the same ontology that defines the structure of knowledge. We develop the ontology for the middleware. The ontology consists of a group of classes, and each class specifies one entity of knowledge for a large-scale distributed system. We define a set of basic classes for the ontology. This ontology is also extensible to include additional knowledge for specific computations supported by a system.

5.2 Conclusions

The goal of this dissertation is to investigate the application of GA approaches to planning and scheduling for large-scale distributed systems. Our study shows that a GA can be applied to the addressed problems but has its restrictions. A GA has the following two strengths that are the desired attributes for the addressed problems.

First, the search for a solution in GA typically uses less knowledge extracted from computing domains than the deterministic approaches. In a changing computing environment, the knowledge that can be applied to the current problem conditions may not be applicable as the conditions change. Our design of the algorithm uses more general knowledge that can be applied to a variety of problem conditions. For instance, the GA-based scheduling algorithm, although designed for homogeneous computing environments, can also be applied to heterogeneous computing environments. The traditional list scheduling algorithms, such as ISH,

DSH, and CPFD, only work well in a homogeneous computing environment in which all processors are fully connected and have the same processing ability.

Second, the diverse solutions produced by GAs offer important resources for planning and scheduling to deal with the dynamics in the computing environment. A GA is a parallel search method and produces a population of candidate solutions. Although only one solution (typically the best one) is selected, the other solutions cannot simply be discarded: they provide resources for planning and scheduling to react to changes in a computing environment. Plans that are not selected may serve as candidate solutions for plan switching when failure occurs on the selected plan; while schedules that are not selected may turn out to be a desirable schedule in a later stage of the semi-static approach. Without GA, we have to rerun the algorithm for each request of planning or scheduling.

The disadvantage of using GA approaches is the high computational cost. As a GA uses less domain-specific knowledge in search for a solution, it typically requires a more complete exploration of the search space than deterministic approaches and thus is less efficient. Our experiments consistently show that the execution time of a GA increases very quickly as the problem size increases, which is unfavorable as both planning and scheduling for large-scale distributed systems involve a search in a very large solution space. Some heuristic approaches, such as the recursive subgoal strategy, can significantly improve the search results and reduce the computational time, but they can only be applied to a limited set of domains.

Our study on planning and scheduling approaches for large-scale distributed systems gives us an insightful view on the design of search algorithms, especially on the issue of how to balance the two criteria for the performance of an algorithm: the efficiency of the search and the quality of solutions. Our GA

algorithms put more focus on the quality of results, but they suffer from much higher computational costs than the deterministic approaches. Although we use several strategies, such as the multi-phase search strategy and the indirect encoding method, to improve the efficiency of the search, much other work is needed to significantly reduce the search space and the execution time. We believe that our GA algorithms can be further improved by incorporating additional methods for narrowing down the search space while not affecting the search results.

In addition, our work extends the previous study in the field of evolutionary computation in two aspects. First, we improve our understanding of the variable length representation by employing this encoding method in the addressed problems. We believe that a variable length representation is ideal to problems in which the size of the optimal solutions cannot be easily determined. Variable length representation also has the advantage over the fixed length GA with the flexibility of dynamically adjusting the solution sizes during the search. This flexibility is important to our GA-based algorithms as the size of solutions to planning and scheduling may vary due to the changes in the computing environment. Second, we study the incremental search strategies in a GA. We apply the incremental search strategy in different ways in our planning and scheduling algorithms. In the planning algorithm, we build a plan incrementally by dividing the search for a solution into multiple phases. Each phase is considered an individual GA run, and the final solution is the concatenation of partial solutions evolved in all phases. In the scheduling algorithm, we use an incremental, dynamic fitness function to encourage the formation and recombination of partial solutions. The success of these strategies indicates that the methods for preserving partial solutions during the search can improve the performance of GAs. We believe that the above attempts on the incremental search strategies can also be applied to other non-deterministic search and optimization approaches. For instance, both

the multi-phase search strategy and the incremental evaluation function can be easily embedded in the simulated annealing approach.

Four areas of research extensions from the current work are on top of the list of items I plan to pursue in the future.

First, we intend to explore a dynamic approach to integrate the function of planning and scheduling for large-scale distributed systems. In this dissertation, we study two approaches, the static and semi-static approaches, to integrate the function of planning and scheduling. In the *static approach*, we schedule a computing task right after a plan is produced. In the *semi-static approach*, we use the plan switching method to switch the execution of a computing task among a family of plans. As a new schedule is required after each successful plan switching, we may schedule the execution of a computing task multiple times. A *dynamic approach* is different from the above approaches as we do not schedule all computing activities in a plan at one time. Instead, we deter the scheduling of an activity until it is dispatched for execution. In this case, scheduling becomes a trivial problem as only one activity is scheduled at each time. The dynamic approach, however, may not work in a large-scale distributed system in which resource-intensive tasks compete frequently for computing nodes. The delay of scheduling an activity may increase the possibility of failure for an activity execution due to the unavailability of resources. It is interesting to study the feasibility of the dynamic approach and compare its performance with the static and semi-static approaches in large-scale distributed systems.

Second, the study of the GA-based planning and scheduling algorithms encourages us to develop new methods to improve the efficiency of the GA. Domain-related knowledge, which can effectively reduce the search space while not sacrificing the quality of solutions, will be studied and incorporated in the search.

On the other hand, we will continue to study new encoding methods and other incremental search strategies for GA to improve its search efficiency.

Third, we attempt to improve the plan switching approach by using multiple criteria in selecting the optimal congruent snapshot. In our current approach, the optimal congruent snapshot is always the one that has the lowest cost of subsequent activities among all candidates. While this is a simple approach, it may result in switching to a plan that contains inexecutable activities so that another request of plan switching may be inevitable. Other heuristics can be embedded into this process to balance both the execution cost of a computing task and the success rate of plan execution.

Fourth, we intend to study the effectiveness of using GA algorithm for providing candidate solutions for plan switching. We plan to evaluate the effectiveness by testing the GA-based algorithm in a number of a real-world computations. We are interested to study whether the candidate solutions contain useful resources for plan switching (i.e., whether congruent snapshots can be located in candidate solutions) and whether a diverse population is beneficial to improving the success of plan switching.

List of References

[1] Foster, I., Kesselman, C.: Blueprint for a New Computer Infrastructure. W. H. Freeman, San Francisco (1999)

[2] Marinescu, D.C.: Internet-Based Workflow Management: Towards a Semantic Web. Wiley, New York, NY (2002)

[3] Marinescu, D.C., Marinescu, G.M., Ji, Y.: The complexity of scheduling and coordination on computational grids. In Marinescu, D.C., Lee, C., eds.: Process Coordination and Ubiquitous Computing, CRC Press (2002) 119–132

[4] Marinescu, D., Ji, Y.: A computational framework for the 3D structure determination of viruses with unknown symmetry. Journal of Parallel and Distributed Computing **63** (2003) 738–758

[5] Fikes, R., Nilsson, N.: STRIPS: A new approach to the application of theorem proving to problem solving. Journal of Artificial Intelligence **2** (1971) 189–208

[6] Erol, K., Nau, D.S., Subrahmanian, V.S.: Complexity, decidability and undecidability results for domain-independent planning. Journal of Artificial Intelligence **76** (1995) 75–88

[7] Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Journal of Artificial Intelligence **90** (1997) 281–300

[8] Jonsson, P., Haslum, P., Backstrom, C.: Towards efficient universal planning, a randomized approach. Journal of Artificial Intelligence **117** (2000) 1–29

[9] Nebel, B., Koehler, J.: Plan reuse versus plan generation: A theoretical and empirical analysis. Journal of Artificial Intelligence **76** (1995) 427–454

[10] Bonet, B., Geffner, H.: Planning as heuristic search. Journal of Artificial Intelligence **129** (2001) 5–33

[11] Korf, R.E., Taylor, L.A.: Finding optimal solutions to the twenty-four puzzle. In: Proc. of the International Conference on Artificial Intelligence (AAAI 96), Portland, OR (1996) 1202–1207

[12] Korf, R.E., Felner, A.: Disjoint pattern database heuristics. Journal of Artificial Intelligence **134** (2002) 9–22

[13] Koza, J.R.: Genetic Programming. MIT Press, Cambridge, MA (1992)

[14] Spector, L.: Genetic programming and ai planning systems. In: Proc. of the 12th National Conference of Artificial Intelligence. (1994) 1329–1334

[15] Muslea, I.: SINERGY: A linear planner based on genetic programming. In: Proc. of the 4th European Conference on Planning, Springer (1997) 312–324

[16] Penberthy, J.S., Weld, D.: UCPOP: A sound, complete, partial-order planner for ADL. In: Proc. of the 3rd International Conference on Knowledge Representation and Reasoning (KR-92), Cambridge, MA (1992) 103–114

[17] Westerberg, C.H., Levine, J.: GenPlan: Combining genetic programming and planning. In: Proc. of the 19th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2000), Open University, Milton Keynes, UK (2000) 255–265

[18] Westerberg, C.H., Levine, J.: Investigation of different seeding strategies in a genetic planner. In: Proc. of the 2nd European Workshop on Scheduling and Timetabling (EvoSTIM 2001), Springer (2001) 505–514

[19] Smith, S.F.: A learning system based on genetic adaptive algorithms. In: PhD thesis, Dept. Computer Science, University of Pittsburgh. (1980)

[20] Goldberg, D.E., Korb, B., Deb, K.: Messy genetic algorithms: Motivation, analysis, and first results. Complex Systems **3** (1989) 493–530

[21] Harvey, I.: Species adaptation genetic algorithms: A basis for a continuing saga. In: Proceedings of the First European Conference on Artificial Life. (1992) 346–354

[22] Wu, A.S., Lindsay, R.K.: Empirical studies of the genetic algorithm with non-coding segments. Evolutionary Computation **3** (1995) 121–147

[23] Wu, A.S., Garibay, I.: The proportional genetic algorithm: Gene expression in a genetic algorithm. Genetic Programming and Evolvable Machines **3** (2002) 157–192

[24] Yu, H., Wu, A.S., Lin, K., Schiavone, G.: Adatptation of length in a nonstationary environment. In: Proc. of Genetic and Evolutionary Computation Conference (GECCO). (2003) 1541–1553

[25] http://www.cut-the knot.com/recurrence/hanoi.shtml: Tower of Hanoi (2005)

[26] Luger, G.F., Stubblefield, W.A.: Artificial Intelligence: Structures and Strategies for Complex Problem Solving (Fifth Edition). Addison-Wesley, Boston, MA (2004)

[27] Russell, S.J., Norvig, P.: Artifricial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River, NJ (1995)

[28] Johnson, W.W., Story, W.E.: Notes on the '15' puzzle. American Journal of Mathematics **2** (1879) 397–404

[29] Wu, A.S., Schultz, A.C., Agah, A.: Evolving control for distributed micro air vehicles. In: Proc. of the IEEE International Symposium on Computational Intelligence in Robotics and Automation. (1999) 174–179

[30] Newell, A., Shaw, J.C., Simon, H.A.: Report on a general problem solving program. In: Proc. of the International Conference on Information Processing. (1960) 256–264

[31] Newell, A., Simon, H.A.: Human Problem Solving. Prentice Hall, Englewood Cliffs, NJ (1972)

[32] Korf, R.E.: Planning as search: A quantitative approach. Journal of Artificial Intelligence **33** (1987) 65–88

[33] Barrett, A., Weld, D.S.: Characterizing subgoal interactions for planning. In: Proc. of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93), Chambery, France (1993) 1388–1393

[34] Barrett, A., Weld, D.S.: Partial-order planning: evaluating possible efficiency gains. Journal of Artificial Intelligence **67** (1994) 71–112

[35] Cheng, J., Irani, K.B.: Ordering problem subgoals. In: Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89), Detroit, USA (1989) 931–936

[36] Etzioni, O.: Acquiring search-control knowledge via static analysis. Journal of Artificial Intelligence **62** (1993) 255–301

[37] Koehler, J., Hoffmann, J.: Planning with goal agendas. Technical Report 110, Institute for Computer Science, Albert Ludwigs University, Freiburg, Germany (1998)

[38] Lin, F.: An ordering on subgoals for planning. Annals of Mathematics and Artificial Intelligence **21** (1997) 321–342

[39] Drummond, M., Currie, K.: Goal ordering in partially ordered plans. In: Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89), Detroit, USA (1989) 960–965

[40] Hertzberg, J., Horz, A.: Towards a theory of conflict detection and resolution in nonlinear plans. In: Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89), Detroit, USA (1989) 937–942

[41] Manna, Z., Waldinger, R.: How to clear a block: A theory of plans. Journal of Automated Reasoning **3** (1987) 343–377

[42] Sacerdoti, E.D.: A Structure for Plans and Behavior. Elsevier-North Holland, New York (1977)

[43] Steel, S.W.D.: An iterative construct for non-linear precedence planners. In: Proc. of the 7th Biennial Conference of the Canadian Society for the Computational Study of Intelligence. (1988) 227–233

[44] Cresswell, S., Smaill, A., Richardson, J.: Deductive synthesis of recursive plans in linear logic. In: Proc. of the 5th European Conference on Planning (ECP-99), Durham, England (1999) 252–264

[45] Ghassem-Sani, R., Steel, S.: Recursive plans. In: Proc. of the European Workshop on Planning, St. Augustin, Germany (1991) 53–63

[46] Smith, D.E., Weld, D.S.: Conformant graphplan. In: Proc. of AAAI-98. (1998) 889–896

[47] Weld, D.S., Anderson, C.R., Smith, D.E.: Extending graphplan to handle uncertainty and sensing actions. In: Proc. of AAAI-98. (1998) 897–904

[48] Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Weak, strong, and strong cyclic planning via symbolic model checking. Journal of Artificial Intelligence **147** (2003) 35–84

[49] Langdon, W.B., Poli, R.: Fitness causes bloat. Soft Computing in Engineering Design and Manufacturing (1997) 13–22

[50] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Morgan Kaufmann, New York (1979)

[51] El-Rewini, H., Lewis, T.G., Ali, H.H.: Task scheduling in parallel and distributed systems. Prentice Hall (1994)

[52] Hou, E.S., Ansari, N., Ren, H.: A genetic algorithm for multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems **5** (1994) 113–120

[53] Kwok, Y., Ahmad, I.: Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. Journal of Parallel and Distributed Computing **47** (1997) 58–77

[54] Tsuchiya, T., Osada, T., Kikuno, T.: Genetic-based multiprocessor scheduling using task duplication. Microprocessors and Microsystems **22** (1998) 197–207

[55] Bollinger, S.W., Midkiff, S.F.: Processor and link assignment in multicomputers using simulated annealing. In: Proc. of the International Conference on Parallel Processing. (1988) 1–7

[56] Hwang, K., Xu, J.: Mapping partitioned program modules onto multicomputer nodes using simulated annealing. In: Proc. of the International Conference on Parallel Processing (ICPP). (1990) 292–293

[57] Nanda, A.K., DeGroot, D., Stenger, D.: Scheduling directed task graphs on multiprocessors using simulated annealing algorithms. In: Proc. of the 12th International Conference on Distributed Computing Systems. (1992)

[58] Porto, S.C.S., Ribeiro, C.C.: A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. International Journal of High-Speed Computing **7** (1995)

[59] Ahmad, I., Dhodhi, M.K.: Multiprocessor scheduling in a genetic paradigm. Parallel Computing **22** (1996) 395–406

[60] Ali, S., Sait, S.M., Benten, M.S.T.: GSA: Scheduling and allocation using genetic algorithm. In: Proc. of EURO-DAC'94. (1994) 84–89

[61] Correa, R.C., Ferreira, A., Rebreyend, P.: Scheduling multiprocessor tasks with genetic algorithms. IEEE Transactions on Parallel and Distributed Systems **10** (1999) 825–837

[62] Dhodhi, M.K., Ahmad, I., Ahmad, I.: A multiprocessor scheduling scheme using problem-space genetic algorithms. In: Proc. of IEEE International Conference on Evolutionary Computing. (1995) 214–219

[63] Wang, L., Siegel, H.J., Roychowdhury, V.P., Maciejewski, A.A.: Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. Journal of Parallel and Distributed Computing **47** (1997) 8–22

[64] Zomaya, A.Y., Ward, C., Macey, B.: Genetic scheduling for parallel processor systems: comparative studies and performance issues. IEEE Transactions on Parallel and Distributed Systems **10** (1999) 795–812

[65] Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press (1975)

[66] Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley (1989)

[67] Lohn, J.D., Haith, G.L., Columbano, S.P., Stassinopoulos, D.: A comparison of dynamic fitness schedules for evolutionary design of amplifiers. In: Proc. of the 1st NASA/DoD Workshop of Evolvable Hardware. (1999) 87–92

[68] Frantz, D.R.: Non-linearities in genetic adaptive search. PhD thesis, University of Michigan (1972)

[69] Eshelman, L.J., Caruana, R.A., Schaffer, J.D.: Biases in the crossover landscape. In: Proc. of the 3rd International Conference on Genetic Algorithms (ICGA). (1989) 10–19

[70] Bagley, J.D.: The behavior of adaptive systems which employ genetic and correlation algorithms. PhD thesis, University of Michigan (1967)

[71] Burke, D.S., De Jong, K.A., Grefenstette, J.J., Ramsey, C.L., Wu, A.S.: Putting more genetics into genetic algorithms. Evolutionary Computation **6** (1998) 387–410

[72] Franceschini, R.W., Wu, A.S., Mukherjee, A.: Computational strategies for disaggregation. In: Proc. of the 9th Conference on Computer Generated Forces and Behavioral Representation. (2000)

[73] Harik, G.R.: Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms. PhD thesis, University of Michigan (1997)

[74] Lobo, F.G., Deb, K., Goldberg, D.E., Harik, G., Wang, L.: Compressed introns in a linkage learning genetic algorithm. In: Proc. of the 3rd Conference on Genetic Programming. (1998) 551–558

[75] Paredis, J.: The symbiotic evolution of solutions and their representations. In: Proc. of the 6th International Conference on Genetic Algorithms (ICGA). (1995) 359–365

[76] Wu, A.S., Lindsay, R.K.: A comparison of the fixed and floating building block representation in the genetic algorithm. Evolutionary Computation **4** (1996) 169–193

[77] Soule, T., Ball, A.E.: A genetic algorithm with multiple reading frames. In Spector, L., Goodman, E.D., Wu, A.S., Langdon, W.B., Voigt, H.M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M.H., Burke, E., eds.: Proc. of Genetic and Evolutionary Computation Conference (GECCO). (2001) 615–622

[78] Wu, A.S.: Non-coding segments and floating building blocks for the genetic algorithm. PhD thesis, University of Michigan (1995)

[79] Forrest, S., Mitchell, M.: Relative building-block fitness and the building-block hypothesis. In: Foundations of Genetic Algorithms 2. (1992) 109–126

[80] Levenick, J.R.: Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In: Proc. of the 4th International Conference on Genetic Algorithms (ICGA). (1991) 123–127

[81] Mayer, H.A.: ptGAs: Genetic algorithms using promoter/terminator sequences. PhD thesis, University of Salzburg (1997)

[82] Soule, T., Foster, J.A., Dickinson, J.: Code growth in genetic programming. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996, Cambridge, MA, MIT Press (1996) 215–233

[83] Kruatrachue, B., Lewis, T.G.: Duplication Scheduling Heuristic, a new precedence task scheduler for parallel systems. Technical Report 87-60-3, Oregon State University (1987)

[84] Ahmad, I., Kwok, Y.: On exploiting task duplication in parallel program scheduling. IEEE Transactions on Parallel and Distributed Systems **9** (1998) 872–892

[85] Macey, B.S., Zomaya, A.Y.: A performance evaluation of CP list scheduling heuristics for communication intensive task graphs. In: Proc. of Joint 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Programming. (1998) 538–541

[86] Wu, M.Y., Gajski, D.D.: Hypertool: A programming aid for message-passing systems. IEEE Transactions on Parallel and Distributed Systems **1** (1990) 330–343

[87] Kruatrachue, B., Lewis, T.G.: Grain size determination for parallel processing. IEEE Software **5** (1988) 23–32

[88] Al-Mouhamed, M.A.: Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. IEEE Transactions on Software Engineering **16** (1990) 1390–1401

[89] Branke, J.: Memory enhanced evolutionary algorithms for changing optimization problems. In: Proc. of Congress on Evolutionary Computation (CEC). (1999) 1875–1882

[90] Cobb, H.G.: An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments. Technical Report AIC-90-001, Naval Research Laboratory, Washington, D. C. (1990)

[91] Deb, K., Goldberg, D.E.: An investigation of niche and species formation in genetic function optimization. In: Proc. of International Conference on Genetic Algorithms (ICGA). (1989) 42–50

[92] De Jong, K.A.: An analysis of the behavior of a class of genetic adaptive systems. PhD thesis, University of Michigan (1975)

[93] Goldberg, D.E., Smith, R.E.: Nonstationary function optimization using genetic algorithms with dominance and diploidy. In: Proc. of International Conference on Genetic Algorithms (ICGA). (1987) 59–68

[94] Grefenstette, J.J.: Genetic algorihtms for changing environments. In: Proc. of Parallel Problem Solving from Nature (PPSN). Volume 2. (1992) 137–144

[95] Grefenstette, J.J.: Evolvability in dynamic fitness lanscapes: a genetic algorithm approach. In: Proc. of Congress on Evolutionary Computation (CEC). (1999) 2031–2038

[96] Kita, H., Yabumoto, Y., Mori, N., Nishikawa, Y.: Multi-objective optimization by means of the thermodynamical genetic algorithm. In: Proc. of Parallel Problem Solving from Nature (PPSN). (1996) 504–512

[97] Liles, W., De Jong, K.: The usefulness of tag bits in changing envrionments. In: Proc. of Congress on Evolutionary Computation (CEC). (1999) 2054–2060

[98] Mori, N., Kita, H., Nishikawa, Y.: Adaptation to a changing environment by means of the thermodynamical genetic algorithm. In: Proc. of Parallel Problem Solving from Nature (PPSN). (1996) 513–522

[99] Smith, R.E.: Diploidy genetic algorithms for search in time varying environments. In: Annual Southeast Regional Conference of the ACM. (1987) 175–179

[100] Chou, T., Abraham, J.: Load balancing in distributed systems. IEEE Transactions on Software Engineering **SE-8** (1981) 401–412

[101] Towsley, D.: Allocating programs containing branches and loops within a multiple processor system. IEEE Transactions on Software Engineering **SE-12** (1986) 1018–1024

[102] El-Rewini, H., Ali, H.H.: Static scheduling of conditional branches in parallel programs. Journal of Parallel and Distributed Computing **24** (1995) 41–54

[103] Parhi, K.K., Messerschmitt, D.G.: Static rate-optimal scheduling of iterative dataflow programs via optimum unfolding. IEEE Transactions on Computers **40** (1991) 178–195

[104] Yang, T., Fu, C.: Heuristic algorithms for scheduling iterative task computations on distributed memory machines. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 608–622

[105] Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Computing Surveys **27** (1995) 367–432

[106] Jain, S.: Circular scheduling: a new technique to perform software pipelining. In: Proc. of ACM Conference on Programming Language Design and Implementation. (1991) 219–228

[107] Lam, M.: Software pipelining: an effective scheduling technique for VLIW machines. In: Proc. of ACM Conference on Programming Language Design and Implementation. (1988) 318–328

[108] Rau, B.R.: Iterative modulo scheduling: An algorithm for software pipelining loops. In: Proc. of the 27th Annual International Symposium on Microarchitecture. (1994) 63–74

[109] Van Dongen, V.H., Gao, G.R., Ning, Q.: A polynomial time method for optimal software pipeling. In: Proc. of the 2nd Joint International Conference on Vector and Parallel Processing. (1992) 613–624

[110] Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems **3** (1985) 63–75

[111] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)