Granular State Space Search

A Thesis

Submitted to the Faculty of Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy

in

Computer Science

University of Regina

By

Jigang Luo

Regina, Saskatchewan

August, 2013

UNIVERSITY OF REGINA

FACULTY OF GRADUATE STUDIES AND RESEARCH

SUPERVISORY AND EXAMINING COMMITTEE

Jigang Luo, candidate for the degree of Doctor of Philosophy in Computer Science**,** has presented a thesis titled, **Granular State Space Search*,*** in an oral examination held on June 20, 2013. The following committee members have found the thesis acceptable in form and content, and that the candidate demonstrated satisfactory knowledge of the subject material.

External Examiner:	*Dr. Witold Pedrycz, University of Alberta
Supervisor:	Dr. Yiyu Yao, Department of Computer Science
Committee Member:	Dr. Sandra Zilles, Department of Computer Science
Committee Member:	Dr. Jing Tao Yao, Department of Computer Science
Committee Member:	*Dr. Luigi Benedicenti, Faculty of Engineering & Applied Science
Chair of Defense:	Dr. Dongyan Blachford, Faculty of Graduate Studies and Research

*Participated via teleconference

Abstract

Granular computing is an emerging field of study that derives general principles and models from a wide range of disciplines and uses these principles and models to solve problems. One basic structure in granular computing is a granular structure, which is a hierarchical multi-level structure made of granules. The granular structure implies important ways to problem solving such as hierarchical problem solving and top-down progressive processing. These ways can be used in artificial intelligence. This thesis focuses on how to use granular computing to solve problems in state space search.

State space search is a classical approach to problem solving in artificial intelligence. A problem solving process is modeled by a state space and a search for a path from the start state to a goal state. One promising method for state space search is hierarchical state space search, which creates abstraction hierarchies as representations of state spaces and uses a procedure, called refinement procedure, to find solutions with the help of abstraction hierarchies. Good abstraction hierarchies and refinement procedures can speed up state space search while bad abstraction hierarchies and refinement procedures can slow down state space search. This thesis proposes granular computing based methods that can construct good abstraction hierarchies quickly and a new refinement procedure.

We first identify two main issues in hierarchical problem solving, namely, back-

tracking and incompleteness. Backtracking degrades the efficiency of state space search while incompleteness decreases the effectiveness and reliability. Based on the two issues, we propose a concept of completeness degree and give criteria for evaluating good abstraction hierarchies. We argue that a good abstraction hierarchy should have few backtrackings and a high completeness degree.

To apply granular computing in state space search, we propose two new concepts, called an attributed graph and a granulated attributed graph, respectively. An attributed graph can be mapped into a state space and a granulated attributed graph can be mapped into an abstraction. We analyze characteristics of attributed graphs and granulated attributed graphs and identify an important property of granulated attributed graphs known as inner connectivity. We argue that if a granulated attributed graph satisfies the inner connectivity property, the mapped abstraction hierarchy is free of backtrackings and incompleteness. Based on the attributed graphs and granulated attributed graphs, we propose a new genre of methods for state space search called granular state space search. Granular state space search can create good abstraction hierarchies that have fewer backtrackings and a higher completeness degree. Furthermore, we improve granular state space search by applying the machine learning techniques so that good abstraction hierarchies can be generated fast. Finally, we compare our granular state space search with existing hierarchical state space search methods and demonstrate the superiority of granular state space search by experimental results.

Acknowledgements

There are lots of people I would like to thank. First, and foremost, I must thank my supervisor, Dr. Yiyu Yao, for his huge support that I needed along the way. Without his help and encouragement I would not have finished this thesis.

I would like to thank Dr. Sandra Zilles for pointing out some most recent works on hierarchical problem solving and suggestions for experimental evaluation. I would like to thank Dr. JingTao Yao and Dr. Luigi Benedicenti to review my thesis and give me constructive comments.

I would like to thank the Faculty of Graduate Studies and Research and Dr. Yiyu Yao for supplying my funding.

Thank you to all of my friends and colleagues (Dr. Bing Zhou, Xiaofei Deng, Rong Fu, etc.) whose help and support were critical for the completion of this research.

Finally, I would like to thank my family: my wife, my parents, parents-in-law, and my daughter for their constant love, support, and encouragement throughout my graduate career.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Definitions

List of Theorems

List of Examples

Chapter 1

INTRODUCTION

Granular computing is a rapidly developing research area, and the results of granular computing research can be widely used in artificial intelligence. In this chapter, we briefly discuss the relations between granular computing and artificial intelligence, introduce a classical approach to problem solving in artificial intelligence, namely, state space search, and explain how to use granular computing to solve state space search problems.

1.1 Granular Computing and Artificial Intelligence

Granular computing is a field of study inspired by human problem solving [99]. It synthesizes principles and models from various disciplines and creates its own principles and models that can be used widely for problem solving. There are two

basic concepts in granular computing, i.e., granules and granulations. Generally speaking, a granule is a group of objects in a problem that are similar in some aspects, and a granulation is a set of all granules that gives an abstract view of the problem.

The framework of the granular computing theory is built upon the granular computing triangle proposed by Yao [96] in his triarchic theory. The essential part of this framework is a granular structure, a hierarchical structure made from granules and granulations. A problem can have different granulations, such that some are more abstract than others. Granulations of a problem can form a hierarchical structure. This structure is a granular structure. Two major tasks in granular computing are to research how to build granular structures and how to solve problems by using these granular structures. Based on granular structures, there are three basic perspectives in granular computing, forming the three vertexes of the granular computing triangle. They are the philosophical perspective, the methodological perspective, and the computational perspective [96]. These three perspectives provide three directions for granular computing research. We will discuss the granular computing triangle in detail in Section 2.1.

Many models and tools have been proposed to accomplish the two tasks of creating granular structures and solving problems by using granular structures. For creating granular structures, the popular tools are information tables and rough set theory [51]. For solving problems, the popular models are top-down progressive computing, bottom-up progressive computing and middle-out progressive computing [67, 101].

Artificial intelligence is a field of study that aims to design a system that perceives its environment and takes actions that maximize its chances of success [64]. The ways that a human processes and solves problems always give inspirations for the research of artificial intelligence. Granular computing, as being inspired by human problem solving, provides a new perspective on artificial intelligence. Ideas in granular computing are used in artificial intelligence, among which abstraction and hierarchical problem solving are the most widely used ones.

Abstraction is the most important idea implied by granular computing. An abstraction can be seen as a simplified version of a problem, which eliminates details of the problem and retains the vital elements. In granular computing, the multi-level granular structure can be seen as a structure built by abstractions. A higher level in the granular structure is always more abstract than a lower level. The procedure of making the granular structure is also the procedure of creating abstractions. An abstraction leads us to the most important part of a problem and lets us ignore the trivial and irrelevant details. Thus, an abstraction can help us understand the essence of a problem and finally solve the problem.

Hierarchical problem solving is also an implied idea in granular computing, which uses hierarchical structures to solve problems. Hierarchical structures are well known organizational structures for modeling large and complex systems. A large system is organized into hierarchical levels, every level is a kind of representation of the system. Hierarchical structures have been proven to be able to improve the efficiency of complex systems such as social systems, biological systems, etc. [68]. The granular structure is a general model of the hierarchical

structures existing in various fields. The multi-level granular structure naturally gives us a way of problem solving, that is, to solve a problem level by level. The processing can be from top level to bottom level, or from bottom level to top level, or even from middle level to other levels. The hierarchical problem solving usually refers to solving a problem from top level to bottom level.

Abstraction and hierarchical problem solving are widely used by state space search methods in artificial intelligence.

1.2 Hierarchical State Space Search

Search is a general problem solving method in artificial intelligence. Hierarchical state space search is a special case.

1.2.1 Search as a Means of Problem Solving

Many problems in artificial intelligence can be represented by a state space that is made of a set of states and a set of operators [64]. A state is a configuration of basic elements in a problem and an operator transforms a state into another state. For example, in the 8-puzzle problem [45], every configuration of the nine tiles is a state, every movement is an operator that can transform one configuration into another. Figure 1.1 illustrates the states and operators of the 8-puzzle problem. It shows three states s_1, s_2 and s_3, where s_1 is transformed into s_2 by the operator "*move blank tile up*", and s_2 is transformed into s_3 by the operator "*move blank tile right*".

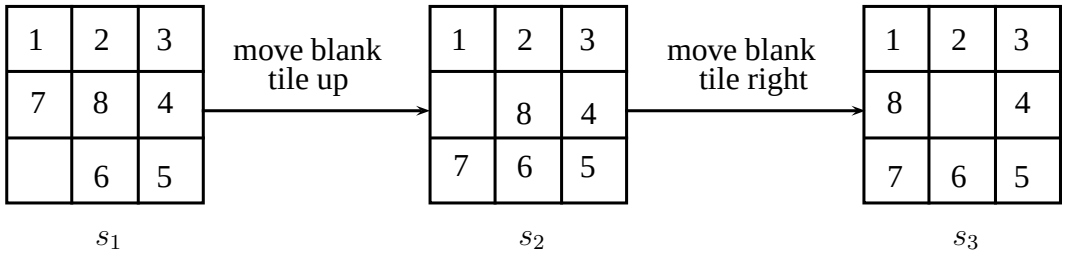


Figure 1.1: The states and operators of the 8-puzzle problem

A state space can be formally expressed as a pair of sets $SP = (S, O)$, where S is the set of states and O is the set of operators. A problem has a start state and a set of goal states. For example, in the 8-puzzle problem, the start state is the initial configuration of the nine tiles and the only one goal state is the configuration in which all tiles are on the specific positions as s_3 shows in Figure 1.1. A state space in fact represents a group of problems, where different start states and goal state sets mean different problems. For example, in the 8-puzzle problem, different initial configurations mean different problems that have the same state space, so all these problems belong to the same group of problems, called the group of 8-puzzle problems.

A graph of a state space (V, E) is a graphical representation of the state space, where $V = S$ and E is the set of edges between states. If a state can be transformed into another state by an operator, then there is an edge between these two states in the graph of the state space.

A solution to a problem is a sequence of operators that transforms the start state into a goal state. The process of finding a solution to a problem is called state space search. In the corresponding graph of a state space, state space search

is to find an edge path that connects the start state to a goal state. A solution can also be expressed by the sequence of states on this edge path. State space search is the most commonly used approach in artificial intelligence and plays a central role for solving many problems such as automatic planning [8] and automated theorem proving [6, 44, 72].

The straightforward method for state space search is exhaustive search, which explores edges one by one until a path is found between the start state and a goal state, or until all edges are explored and no path is found. An example of exhaustive search is breadth-first search, which starts from the start state and explores all edges connected to the start state, then explores all edges connected to the neighboring states, and so on. The process stops when a solution is found or when all reachable edges are explored and no solution is found.

Another exhaustive search method is depth-first search, which starts from the start state and explores an edge connecting the start state to a second state and then explores an edge connecting the second state to a third state, and so on until an already explored state is reached or a goal state is reached. When an already explored state is reached, the search backtracks to the second latest explored state, and explores an alternative edge by the same process. The process stops when a solution is found or when all reachable states are explored and no solution is found.

Straightforward methods can either find a solution or confirm that no solution exists. A straightforward method is at odds with efficiency because it cannot find a solution to a problem in a reasonable amount of time, especially when

the size of the problem's state space is large. As many problems in artificial intelligence have a large state space, straightforward methods are impracticable for many problems. Therefore, heuristic search [9, 54] has been proposed, which uses heuristics to speed up the search process.

A heuristic is additional knowledge that serves as an advice for searching a solution. Heuristic search uses heuristics to determine the order of states that will be explored or to even eliminate some states. Good heuristic search methods can search states in the order according to their likelihood of being in the solutions, or can eliminate states that are not in any solutions. A good heuristic search can find a solution quickly.

Heuristics can come from previous experience, domain-specific knowledge, or even instinct. There are many different heuristic search methods that choose heuristics from different sources and utilize heuristics in different ways. Among these heuristic search methods, hierarchical search is an important one which uses hierarchical structures [24, 34, 36, 38, 49, 79, 81, 88].

1.2.2 Hierarchical State Space Search

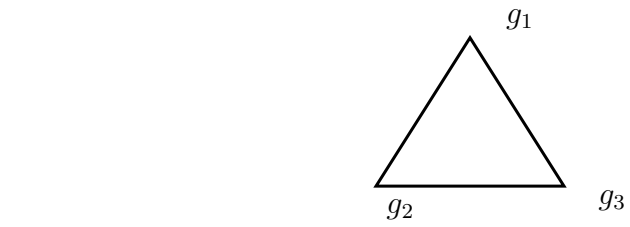
One application of the ideas of abstraction and hierarchical problem solving in artificial intelligence is hierarchical state space search which finds solutions by using hierarchical structures [2, 50, 87].

Hierarchical structures are made of abstract state spaces or abstractions, which are abstract versions of the original state space. An abstraction has abstract states and abstract operators. An abstract state is made by grouping

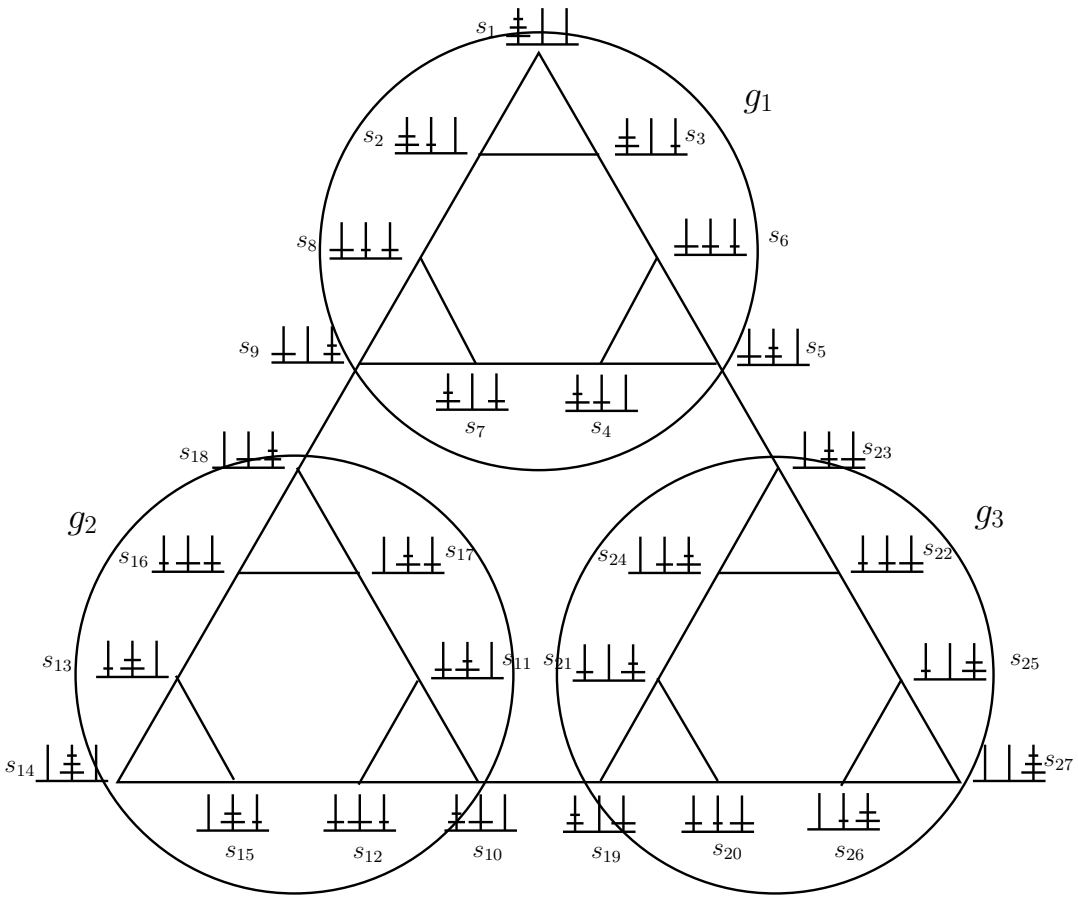
together a few original states and an abstract operator is derived from a few original operators. A hierarchical structure of a state space is an abstraction hierarchy, in which the ground level is the original state space and every other level is an abstraction of the lower level.

In the search process, an abstract solution is first found on the top level abstraction, then the abstract solution on one level is used to guide the search for a solution at the next lower level, until a solution on the ground level is found. As the top level abstraction contains fewer abstract states, a straightforward search method can be used on this level; as there is a guidance on all levels below, the straightforward search method can also be used with the guidance to quickly find solutions on these levels.

Figure 1.2 illustrates how hierarchical state space search works. Figure 1.2b is the graph of the state space of the three-disk Hanoi tower problem (the description of this problem is in Section 3.1.1). In this state space s_1, s_2, $\cdots$, s_{27} are the states, where s_1 is the start state and s_{27} is the goal state. Figure 1.2a is the graph of an abstract state space, which is created by grouping s_1, $\cdots$, s_9 into g_1, grouping s_{10}, $\cdots$, s_{18} into g_2, and grouping s_{19}, $\cdots$, s_{27} into g_3. There is an edge between s_9 (which is grouped into g_1) and s_{18} (which is grouped into g_2) in the original state space, we can derive an abstract operator that transforms g_1 to g_2. Thus, there is an edge between g_1 and g_2 in the graph of the abstract state space. By the same reason there are abstract operators that can transform g_1 to g_3 and transform g_2 to g_3. In the abstract state space, the corresponding start state and goal state are g_1 and g_3. A solution in the abstract state space is found by a



(a) Top level abstraction



(b) Original state space

Figure 1.2: An abstraction hierarchy for state space search (Knoblock [35])

straightforward method, which is the abstract operator that transforms g_1 to g_3. When we search in the original state space with the guidance of the solution in the abstract one, we can cut off all the states that are grouped into g_2 along with edges between these states, because g_2 is not in the solution of the abstract state space. Thus, s_{10}, s_{11}, $\cdots$, s_{18} and the edges between them are cut off, we only need to search among the remaining 18 states, which makes the search process quicker.

From the above example we can see that an abstract state space corresponds to a partition of the original state set. If there is a partition of the original state set, we can take every block of the partition as an abstract state, and derive abstract operators from the original operator set so that if an operator can transform a state in one partition block into a state in another partition block, then there is an abstract operator that can transform an abstract state into another abstract state according to the two partition blocks. These abstract states and abstract operators form an abstraction. From one abstraction, we can again find a partition of the abstract state set, and get another abstraction that is more abstract than the first abstraction. In this way, we can create an abstraction hierarchy.

1.2.3 Earlier Results in Hierarchical State Space Search

Abstraction hierarchies were first used by Newell and Simon [50] in a problem solver called *General Planning System*. This system "uses abstraction to focus attention on the difficult parts of a problem, leaving the details or less critical

parts of a problem to be filled in later. This is usually done by first solving a problem in an abstract space and then using the abstract solution to guide the problem solving of the original more complex problem" [35]. After *General Planning System*, abstraction hierarchies were used in many problem solvers such as ABSTRIPS [65], NOAH [66], etc.

In these early problem solvers, abstraction hierarchies were manually constructed. There were no principles about how to construct abstraction hierarchies and no standards about what is a good abstraction hierarchy. Abstraction hierarchies in these early systems neither guaranteed the efficiency (to speed up state space search) nor guaranteed the effectiveness (to eventually find a solution). Therefore, research that followed focused on identifying properties of good abstraction hierarchies that guarantee the efficiency and effectiveness of hierarchical state space search.

Tenenberg [80] identified the upward and downward solution properties of abstraction hierarchies. The upward solution property states that "the existence of a ground-level solution implies the existence of an abstract-level solution" [80]. The upward solution property exists in all abstraction hierarchies, and therefore, it does not distinguish good abstraction hierarchies from bad abstraction hierarchies. The downward solution property states that "the existence of an abstract-level solution implies the existence of a ground-level solution" [80]. The downward solution property can guarantee the effectiveness, but this property is too restrictive to be useful because not many abstraction hierarchies have this property.

Yang and Tenenberg [89] identified a less restrictive property, the monotonicity property, which states "The existence of a ground-level solution implies the existence of an abstract-level solution that can be refined into a ground-level solution while leaving the literals established in the abstract plan unchanged" [89]. They also developed a problem solver by using abstraction hierarchies that had this property. The advantage of the monotonicity property is that it can guarantee effectiveness, but it cannot guarantee the efficiency.

Knoblock [35] identified the ordered monotonicity property that states "Every refinement of an abstract plan leaves all the literals that comprise the abstract space unchanged" [35]. The ordered monotonicity property can guarantee both effectiveness and efficiency. Knoblock designed an algorithm to automatically generate good abstraction hierarchies that satisfied the ordered monotonicity property [35]. One drawback of the ordered monotonicity property is that it is restrictive and cannot be used in many problems. We will thoroughly discuss Knoblock's work in Section 3.4.

In all the above work, abstraction hierarchies are generated implicitly, that is, an abstract state is represented by a set of *state variables* [22]. Abstraction hierarchies can be also generated explicitly, that is, an abstract state is represented by all the original states that can be mapped to the abstract state. There are many studies on how to generate good abstraction hierarchies explicitly. Holte et al. [31] pioneered in the work of generating abstraction hierarchies explicitly, they proposed the STAR algorithm, which groups original states within a predefined radius to form an abstract state. Clique-based abstraction, proposed by

Sturtevant and Buro [75], is an improvement of Holte et al.'s method, which "is to abstract cliques in each level of the abstraction. Every node in a clique will be no more than one edge away from every other node, which is a desirable property" [78]. Also, there is a line-based abstraction method proposed by Sturtevant and Jansen, which "finds sequences of nodes of length k, and abstracts them together" [78]. The advantage of explicit methods is that the abstraction hierarchies are effective, but as these methods need to explore all states in the original state space, the efficiency is not necessarily guaranteed.

1.2.4 Main Issues in Hierarchical State Space Search

Before using hierarchical state space search, two issues should be addressed, namely, backtracking and incompleteness. These two issues affect the efficiency and effectiveness of hierarchical state space search.

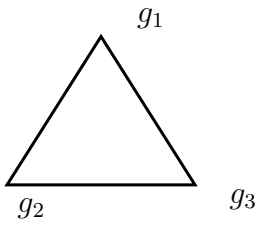
Backtracking

Backtracking means a solution at one level of the abstraction hierarchy cannot be found with the guidance of a higher level abstract solution, then the search process has to go back to the higher level to find another abstract solution. Backtracking abandons some partial result and affects the efficiency of the search process. For example, Figure 1.3 is another abstraction hierarchy for the same original state space as in Figure 1.2. In this abstraction, an abstract solution $g_1 g_3$ is found. The search process cuts off all states in g_2 and only considers the states in g_1 and g_3. Figure 1.4 is the remaining state space after the cut-off. We find that the

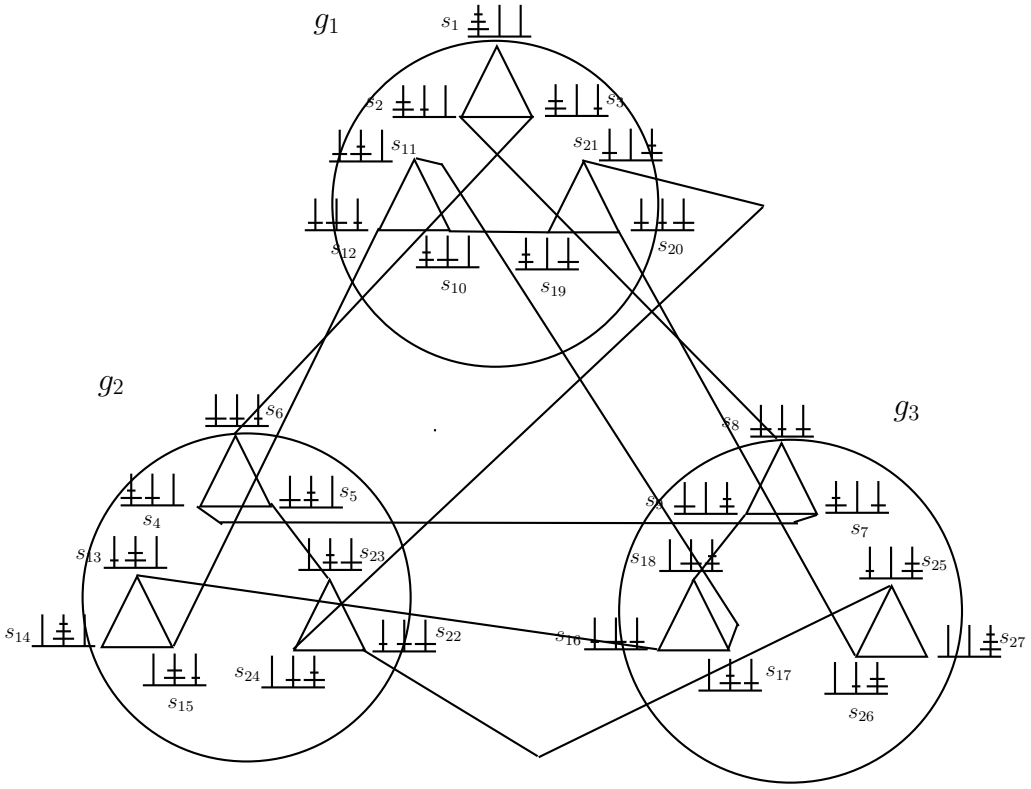
only way from s_1 to a state in g_3 is $s_1s_2s_8$ or $s_1s_3s_2s_8$, but we cannot find a path from s_8 to s_{27} within g_3. This means that we cannot find a final path with the guidance of the abstract solution. We have to abandon the abstract solution g_1g_3 and go back to the abstraction to find another abstract solution $g_1g_2g_3$, which can guide us to find a final solution. We can see that the backtracking wastes effort and slows down the search process.

Incompleteness

If there exists a solution in a state space for a problem, but the hierarchical state space search fails to find any solution by using an abstraction hierarchy, then the abstraction hierarchy is incomplete. Incompleteness means that the abstraction hierarchy is not fully effective. For example, Figure 1.5 is the third abstraction hierarchy for the same original state space in Figure 1.2. The only path in the abstraction that may be used to guide the search for the final solution $s_1s_3s_6s_5s_{23}s_{22}s_{25}s_{27}$ is $g_1g_3g_2g_1g_3$. But this path is not an abstract solution, because there is a cycle g_1-g_3-g_2-g_1 in it, and cycles are not allowed in a solution. If cycles are allowed, there would be infinitely many possible abstract solutions, such as $g_1g_2g_1$, $g_1g_2g_1g_2g_1$, $g_1g_2g_1g_2g_1g_2g_1$,..... If one cycle path fails to guide to find a final solution, we choose the second cycle path by backtracking. If the second cycle path fails again, we choose the third cycle path by backtracking, etc. Theoretically, we need to try all the cycle paths to either find a final solution or confirm that no final solution exists. As there are an infinite number of cycle paths, it is impossible for us to try all the cycle paths. When we cannot find



(a) Top level abstraction



(b) Original state space

Figure 1.3: Backtrackings in an abstraction hierarchy

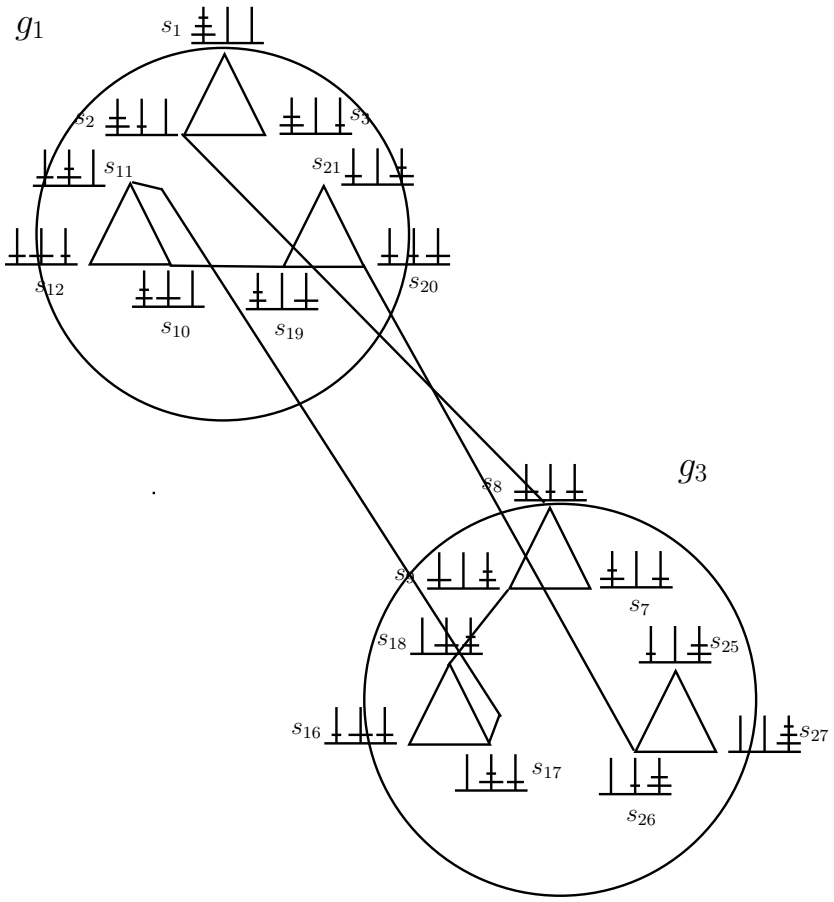


Figure 1.4: The remaining state space after a cut-off

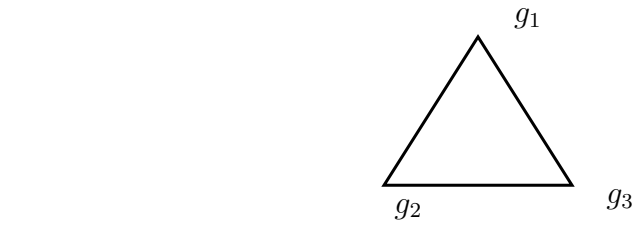
a final solution after trying many cycle paths, we do not know whether or not we should continue to try more cycle paths, because we do not know whether a final solution can be found by some cycle path or a final solution does not even exist. Therefore, cycles in solutions are not allowed in order to avoid infinitely many possible abstract solutions. In Figure 1.5, the only two abstract solutions are g_1g_3 and $g_1g_2g_3$, which cannot guide us to find a final solution. That is, the abstraction hierarchy in this example cannot find an existing solution, and this abstraction hierarchy is incomplete.

1.2.5 Existing Methods for Tackling the Two Issues

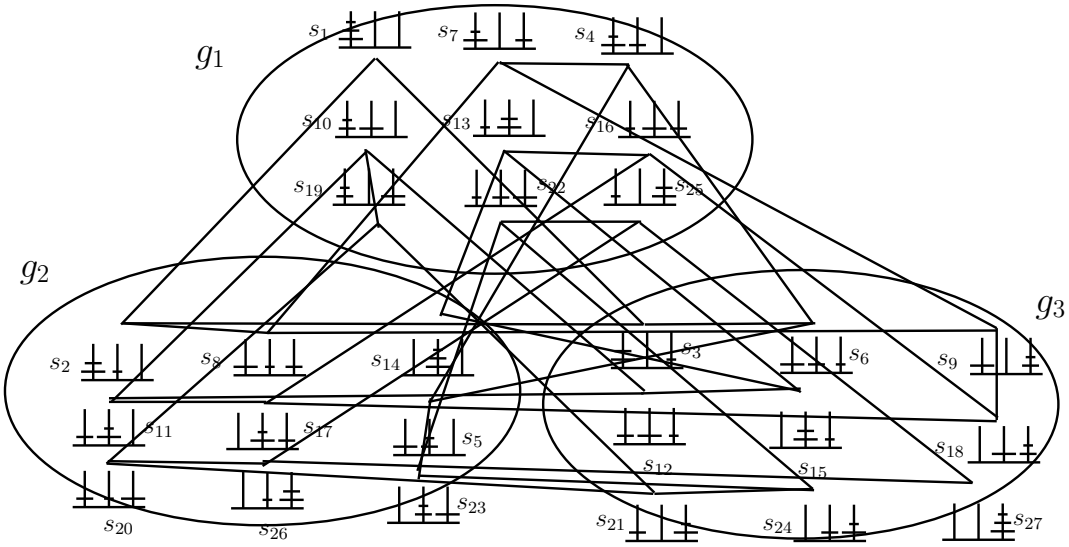
In order to use hierarchical state space search efficiently and effectively, we need to create abstraction hierarchies with as few backtrackings as possible and not incomplete. Many methods have been proposed to create such good abstraction hierarchies. The most popular ones are Knoblock's method [35] and Holte et al.'s method [31].

Knoblock's method for generating an abstraction hierarchy

In Knoblock's method [35], the states and operators are described by a first-order language. A state is a set of literals in the first-order language, an operator has two parts, which are the precondition part and the effect part, each part is a set of literals in the first-order language. Every literal is assigned a level value, so that the abstract states and abstract operators in abstractions are derived from the original states and operators by dropping all literals whose level values are



(a) Top level abstraction



(b) Original state space

Figure 1.5: Another bad abstraction hierarchy

less than a particular threshold. The higher the threshold, the more abstract the created abstraction. Thus, by setting different thresholds, this method can create varying abstractions that form an abstraction hierarchy.

Knoblock [35] defines the ordered monotonicity property, which, generally speaking, means that if a final solution exists, then for any abstract solution P at one level, there is a solution at the next lower level that can be found with the guidance of P. As long as an abstraction hierarchy has the ordered monotonicity property, the incompleteness can be avoided. Knoblock gives two requirements that are sufficient to generate abstraction hierarchies that have the ordered monotonicity property:

1. For any operator, all literals in the effect part have the same level value.

2. For any operator, the level value of any literal in the effect part is no less than the level value of any literal in the precondition part.

Knoblock further gives an algorithm to automatically check all operators and to assign right level values to literals to satisfy the above two requirements. Consequently, an abstraction hierarchy without incompleteness can be created.

The abstraction hierarchies created by Knoblock's method are immune to incompleteness. However, the two requirements in this method are so restrictive that no level value assignment can be found in many problems. Therefore, this method has limited applications.

Holte et al.'s method for generating an abstraction hierarchy

Another popular method is the one proposed by Holte et al. [31]. This method randomly partitions the state set such that the radius of every partition block is no more than r (which is a pre-decided value), and every two states in the same partition block are connected by an edge or by a path within the partition block. This partition is then used to induce an abstraction. Once all the abstractions in a hierarchy are created in this way, this abstraction hierarchy can avoid any backtracking and incompleteness.

Based on Holte et al.'s method, there are other methods such as the clique-based abstraction method, which ensures that all the original states being mapped into a same abstract state form a clique [75], and the line-based abstraction method, which ensures that all the original states being mapped into a same abstract state form a line [78]. All these methods have the same characteristic that they have to explore all states in the original state space and all abstract states in the abstractions before creating an abstraction hierarchy. They are not suitable for solving problems with large state spaces. Furthermore, as the abstractions are constructed by randomly generated partition blocks, they may not have semantic meanings.

1.3 Granular State Space Search

To overcome the issues in hierarchical state space search, we propose a new kind of state space search methods called granular state space search, which uses gran-

ular computing methods to create good abstraction hierarchies. The difference between hierarchical state space search and granular state space search is that hierarchical state space search uses the ideas in granular computing such as abstraction and hierarchical problem solving, while granular state space search uses not only the ideas but also models and tools in granular computing. Granular state space search is a specific application of granular computing models and belongs to the computational perspective of granular computing. Granular state space search processes the search from the granular computing perspective and utilizes the latest research results in granular computing, which brings a new horizon to the study of state space search and better methods for improving search efficiency and effectivity.

1.3.1 Attributed Graph

For using granular computing methods to create abstraction hierarchies, we propose a new concept called attributed graph to represent a state space. An attributed graph is a combination of two parts: an information table and a graph. The information table describes the states. Every row in the information table is a description of a state. The graph in an attributed graph is the same as the graph of a state space. For example, Table 1.1 is the information table for the state space of the three-disk Hanoi tower problem. There are three attributes A, B, C in this information table, which describe the positions of disk A, B, C, respectively. The first row describes state s_1 in which $A = 1$, $B = 1$ and $C = 1$, that is, disk A, disk B and disk C are all on *peg* 1. The graph of this state space

is shown in Figure 1.2.

state	C	B	A
s_1	1	1	1
s_2	1	1	2
s_3	1	1	3
s_4	1	2	1
s_5	1	2	2
s_6	1	2	3
.....			

Table 1.1: The information table for a state space

The advantage of an attributed graph over other granular computing tools is that it describes both the semantics of objects (by attributes) and the relations between objects (by a graph). Granulations created by attributed graphs satisfy both semantic requirements and graphical requirements.

The traditional granular computing method of creating a granulation is done by selecting a subset of the attribute set in the information table, so that all the objects in a granule share the same values on this selected subset [91]. For example, if we select $\{C, B\}$ as a subset of the attribute set in Table 1.1, we can put s_1, s_2, s_3 in the same granule because they have the same values on C and B, also we can put s_4, s_5, s_6 in the same granule. In this way we can group together all the states into granules, and these granules form a granulation.

After combining an information table with a graph, we can go further when

creating a granulation. That is, all the objects in a granule not only share the same values on some attribute subset, but also satisfy some graphical requirements. For example, in the three-disk Hanoi tower problem, if the graphical requirement is that every two objects in a granule should be connected by a path within this granule, then not all subsets of the attribute set are suitable for creating granulations. For example, Figure 1.2 is a granulation created by selecting $\{C\}$ as the subset, Figure 1.3 is a granulation created by selecting $\{B\}$ as the subset and Figure 1.5 is a granulation created by selecting $\{A\}$ as the subset. Among these three granulations, only the granulation created by selecting $\{C\}$ satisfies the graphical requirement.

Because granulations created by attributed graphs satisfy both semantic requirements and graphical requirements, attributed graphs are suitable for creating abstractions for state spaces. The semantic requirements give us a way to create abstractions quickly by selecting attribute subsets, and the graphical requirements can guarantee the desirable properties of the created abstractions.

1.3.2 New Abstraction Hierarchy Generating Methods

The first method for creating abstraction hierarchies we propose is an exhaustion style method. In this method, the graphical requirement is that every partition block satisfies the inner connectivity property. Inner connectivity of a partition block means that this partition block is a strongly connected component. This property guarantees the prevention of backtracking and incompleteness. In this method, we exhaustively examine all the subsets of the attribute set and find all

subsets that can create abstractions that satisfy the inner connectivity property. This method is like Holte et al.'s method in the sense that every partition block is inner connective. The difference is that our method partitions the state space according to the semantics (attributes) of states instead of randomly. All states in the same partition block are semantically similar, which makes it easy to explain granulations or abstractions. For example, we only need to use an attribute subset to express a granulation or abstraction instead of using all states like Holte et al.'s method does.

The exhaustion style method is the first trial to create abstractions by granular computing, but it is in fact not efficient because it requires us to explore all states just like Holte et al.'s method does. The second learning based method is proposed. We first randomly generate a set of sample paths in a state space. We then use these sample paths to learn the optimal attribute subsets. This method is based on the same principle as the exhaustion style method, which considers both semantics and external relations between states. The difference is that learning based method uses the machine learning techniques to quickly find proper partitions without exploring all states. The learning based method is more efficient and can be widely used in many problems. Furthermore, as the learning based method uses attributed graphs, every abstract state in every abstraction has semantic meanings and the abstractions are easy to understand.

1.4 Contributions of the Thesis

The contributions of this thesis are in both granular computing and artificial intelligence. We propose new models and representation tools that enrich the tool kit and extend the application scope of granular computing. The application of these tools in state space search provides an efficient and effective method for generating good abstraction hierarchies.

First, this thesis suggests a granular computing model – the top-down progressive computing model. This model bridges the gap between the triarchic theory and concrete applications of granular computing, and provides a method for solving problems by granular computing.

Second, this thesis proposes three new tools in granular computing, which are attributed graphs, granulated attributed graphs and granulated attributed graph hierarchies. These tools can be used to represent problems, granulations and granulation hierarchies. These three new tools can describe both the inner properties of granules and relations between granules, which extends the ability of granular computing and gives granular computing a potential to be used in many applications.

Third, this thesis proposes methods for creating granulated attributed graphs and granulated attributed graph hierarchies from an attributed graph. It explores properties of granulated attributed graph hierarchies and proves some useful theorems about granulated attributed graph hierarchies. These theorems can be used to design methods for creating good abstraction hierarchies.

Fourth, this thesis uses granulated attributed graph hierarchies to represent abstraction hierarchies and gives a new way to create abstraction hierarchies through attributed graphs. It proposes a new method for creating good abstraction hierarchies. Abstraction hierarchies created by this new method have meaningful structures and are semantically clear, thus they can be easily understood and further processed if needed. Furthermore, the performance of the new method is good. It is more efficient than Holte et al.'s method in the term of explored states and is almost as efficient as other advanced explicit methods. Comparing with Knoblock's method, the new method has wider applications and has the same efficiency.

Fifth, this thesis proposes a new refinement procedure called $PSRB$ refinement, which can reuse the partial result as much as possible when backtracking happens. The $PSRB$ refinement can further speed up state space search.

Sixth, this thesis provides an implementation and empirical demonstration of the new abstraction hierarchy generating method and refinement procedure. The empirical results show that the new abstraction hierarchy generating method and refinement procedure are effective and efficient, which also demonstrates the power and usefulness of attributed graphs.

1.5 Outline of the Thesis

The rest of this thesis is divided into six chapters, which describe granular computing, hierarchical state space search, granulated attributed graphs, granular

state space search, empirical results and summary. The chapters are organized as follows.

Chapter 2 introduces basic theories and concepts in granular computing, and proposes a top-down progressive computing model in granular computing. Furthermore, an explanation is given for why granular computing can be used to solve artificial intelligence problems.

Chapter 3 introduces state space search, abstraction hierarchies and refinement procedure. It explains how an abstraction hierarchy and refinement procedure are used to speed up state space search. It further identifies characteristics of good abstraction hierarchies and discusses existing methods for generating good abstraction hierarchies.

Chapter 4 introduces the notion of a granulated attributed graph, and examines its features. Useful theorems about granulated attributed graphs are proved.

Chapter 5 reports granular computing based methods for generating good abstraction hierarchies. These granular computing based methods form a new genre of state space search methods called granular state space search. This chapter presents an exhaustive method to generate good abstraction hierarchies through attributed graphs. As the exhaustive method is not efficient, a more efficient learning based method is proposed. The learning based method uses the machine learning techniques to generate good abstraction hierarchies quickly. Finally, a new refinement approach is proposed, which can further speed up state space search.

Chapter 6 gives empirical results that demonstrate the superiority of the gran-

ular state space search methods.

Chapter 7 gives the summary of the thesis and points out some future research directions.

Chapter 2

AN INTRODUCTION TO GRANULAR COMPUTING

In this chapter, we present an overview of the theoretical foundations and methodologies of granular computing and propose a top-down progressive computing model. This chapter contains four parts. The first part introduces the triarchic theory of granular computing, which is the foundation for granular computing as a field of study. The second part discusses the relations between granular computing and artificial intelligence and explains ideas of granular computing that are used in artificial intelligence. The third part introduces basic concepts and notions in granular computing and proves some useful theorems. In the fourth part, we propose a granular computing model named top-down progressive computing [101].

2.1 Triarchic Theory of Granular Computing

The triarchic theory of granular computing is proposed by Yao [96], serving as a foundation of granular computing research.

2.1.1 Overview

Granular computing is an interdisciplinary study of human-inspired computing and problem solving [95]. The development of granular computing is based on the research results of a wide range of areas such as rough sets, machine learning, algorithms, etc. The two major purposes of granular computing are to provide a better understanding of the underlying principles of human problem solving and to design machines and systems based on the same principles [100]. To achieve these purposes, granular computing analyzes the general principles in various areas, such as the principles of divide-and-conquer, from-general-to-specific, etc., and incorporates the view of granularity into these principles to derive structures, models and algorithms. The derived structures, models and algorithms can in turn help improve the research in other areas.

To make granular computing a mature and wholesome field of study independent from other research areas, a unified framework for granular computing is necessary so that all the research of granular computing can be carried out in this framework. Proposed by Yao [92, 93, 96], the triarchic theory of granular computing is a conceptual model characterized by the *granular computing triangle* as shown in Figure 2.1. The central part of the granular computing triangle is *gran-*

ular structures, which are a family of complementary hierarchies working together for a complete and comprehensive understanding and representation of a problem or a system from different views and at different levels. Surrounded around the granular structures are three basic perspectives on granular computing: the philosophical perspective, the methodological perspective, and the computational perspective [96].

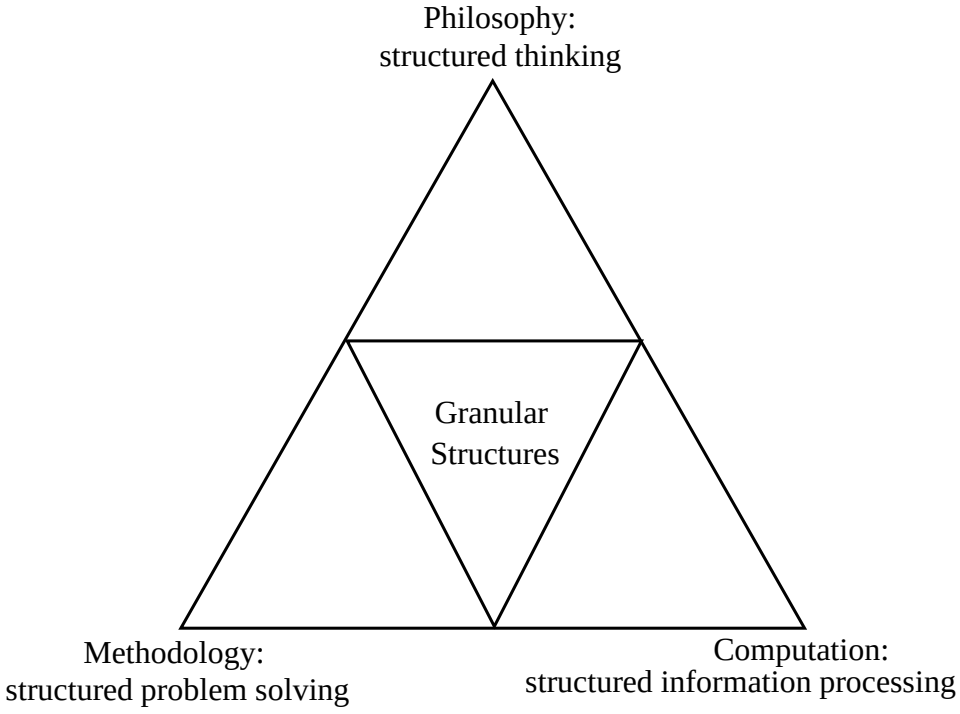


Figure 2.1: The granular computing triangle (Yao [96])

The triarchic theory and granular computing triangle have laid the foundation for granular computing. They provide a guidance for the development of granular computing and point out three major research directions for granular computing. Based on the triarchic theory and granular computing triangle, granular computing can be seen as an independent field of study.

2.1.2 Granular Structures

Any scientific research field needs information or data, which comes in different formats for different fields of study. In granular computing, the basic format of data is the granular structure. By extracting common features of various structures in many fields of science and technology, the notion of granular structures is both a kind of organization of information and a guidance for information processing. Ideas created to granular structures are multi-level and multi-view [18], that is, a granular structure is a hierarchy of different levels, many hierarchies form different views of the same problem.

The basic ingredient of granular structure is level. Level is used to describe, organize and interpret things for the purposes of simplicity and clarity [97]. The concept of level has been extensively used in almost all branches of science. Granular computing, as a cross-disciplinary study, takes this concept as its fundamental ingredient. As defined by Conger [20], a level is a class of structures or processes which are distinguishable from others as being either higher or lower. The difference of higher or lower levels implies an order between levels, thus a sequence of levels can be made by this order and can form a multi-level structure which is the skeleton of a granular structure.

The constructing units of a granular structure are granules and granulations. A granule can be thought of as a set of information extracted from a problem based on a specific granularity, and thus, different granules may have different granularities. A granulation is a group of related granules. A granular structure is a hierarchy which is made of levels of granulations. One such hierarchy provides

a complete view of a problem, and many hierarchies can provide multi views of a problem.

2.1.3 Three Perspectives of Granular Computing

The granular structure is the base of the triarchic theory, around which the three perspectives are developed. These three perspectives are tied closely together so that any one perspective is an indispensable part of granular computing.

The philosophical perspective is more about specifying ways of thinking in granular computing than dealing with concrete problems. It is summarized from various problem solving methods across a wide range of disciplines. The philosophical perspective is a guidance for granular computing, and gives general philosophies that granular computing follows. Any concrete granular computing models and algorithms should follow the philosophical perspective to get best results. The philosophical perspective is a dynamically growing group and includes new philosophies as the research of granular computing moves on. At present, the most frequently used philosophies in granular computing are structured thinking, and hierarchical thinking, which are embodied by most granular computing models and algorithms.

The methodological perspective is more specific than the philosophical perspective, it specifies general models and methods in granular computing. These models and methods follow the philosophies in the philosophical perspective and provide a methodological guidance for solving concrete problems in granular computing. Like the philosophical perspective, the methodological perspective is also

a dynamically growing group, more and more models and methods are added into this group as the research of granular computing advances. Developing these models and methods is one of the major research focuses in granular computing.

The computational perspective focuses on concrete algorithms to solve specific problems. These algorithms are based on models in the methodological perspective and also guided by philosophies in the philosophical perspective. The computational perspective can also be seen as the application of granular computing, it provides solutions for various problems.

The three perspectives are integrated together into granular computing and are dependent on each other. For example, the development of the methodological perspective follows the guidance of the philosophical perspective and on the other hand provides ideas for the development of new philosophies. The creations of new algorithms in the computational perspective follow the models in the methodological perspective and in turn may give hint for creating new models. The three perspectives indicate three directions for the research of granular computing. The advance of every perspective may benefit from and be beneficial to the other two perspectives. In this thesis, our research lies in the methodological perspective and the computational perspective.

2.2 Artificial Intelligence Perspectives on Granular Computing

As granular computing has synthesized the general principles of various disciplines, it can be used in a variety of areas. Artificial intelligence is one area that witnesses more and more usages of granular computing. The philosophies in granular computing provide directions for artificial intelligence, and the models in granular computing can be used to develop methods in artificial intelligence.

In this section we explain the relations between granular computing and artificial intelligence and discuss how the ideas in granular computing are used in artificial intelligence.

2.2.1 Granular Computing and Artificial Intelligence

As pointed out by Simon [73], artificial intelligence can have two purposes. One is to use the power of computers to augment human thinking, just as we use motors to augment human or horse power. The other is to use a computer's artificial intelligence to understand how humans think. Similarly, as proposed by Yao [99], there are two goals for granular computing, one is to understand the nature, the underlying principles and mechanisms of particular ways of human problem solving; the other is to apply them in the design and implementation of human-inspired machines and systems [100]. The difference between artificial intelligence and granular computing is that artificial intelligence focuses on specific problems such as automated planning, machine learning, etc., and aims to get best solutions

for these problems, while granular computing focuses more on abstract models and generic principles and how to use them in specific applications. Therefore, the results of granular computing can be applied in artificial intelligence.

One challenge in artificial intelligence is the dichotomy between machines and humans regarding the complexity or easiness in solving different problems [48]. Some tasks are easy to achieve by a human but hard by a machine, such as human perception. To solve this dichotomy, we need to understand better about human problem solving. Granular computing is relevant to the task of reverse-engineering the mechanisms of human problem solving and, hence, may have a significant impact on artificial intelligence [100].

2.2.2 Ideas of Granular Computing in Artificial Intelligence

The characteristics of multi-level and multi-view of granular structure imply two important ideas in granular computing, which are abstraction and hierarchical problem solving. These two ideas are also widely used in artificial intelligence. As the ideas of abstraction and hierarchical problem solving in granular computing are summarized from various areas, they are more general and powerful than those in artificial intelligence. The research of granular computing can help us understand these two ideas better and use them more effectively in artificial intelligence.

Abstraction is a kind of information processing, it extracts vital information from a set of data to form a new set of data which is more abstract than the

original data. In a multi-level structure, the information in a higher level is formed by extracting common features from the information in a lower level. The higher level eliminates some details and is more abstract than the lower level. The procedure of making a granular structure is the procedure of making more and more abstract levels. The same idea of abstraction is essential in artificial intelligence [19, 37, 82]. For example, in state space search, abstract states are made by eliminating details of states and summarizing commonalities of states. Although abstractions have already been used in artificial intelligence [29, 114], there is not a general model of how to create good abstractions and how to use abstractions to solve problems. The granulation in granular computing can be seen as an abstraction in artificial intelligence [21, 32, 108]. Therefore, the principles, models and algorithms of constructing granulations and processing granulations in granular computing can be used in artificial intelligence. For example, the models in granular computing for generating good granulations can be used in artificial intelligence to create good abstractions and improve the efficiency of problem solving.

In granular computing, hierarchical problem solving is a way of processing information and solving problems by using granular structures. As every level of a granular structure provides a description of the same problem, we can solve the problem at different levels. Furthermore, as every level is made by the information from the same problem and a higher level is more abstract than a lower level, there are relations between every two consecutive levels. These inter-level relations imply a relation between the problem solving of two levels, and give

us a hint that a higher level problem solving along with the inter-level relations may help improve the efficiency of problem solving at a lower level. There is also hierarchical problem solving in artificial intelligence, such as hierarchical state space search [35]. However, hierarchical problem solving in artificial intelligence always aims at solving specific problems and does not have general models. By introducing the hierarchical problem solving models in granular computing (such as top-down progressive computing proposed in Section 2.4) into artificial intelligence, we can understand more about the hierarchical problem solving in artificial intelligence and design better methods and algorithms in artificial intelligence.

2.3 Basic Concepts and Notions of Granular Computing

The idea of granular computing comes from the way a human processes information and solves problems [90,107]. A human can perceive a problem from different angles and levels, and solve the problem level by level. This idea has already been embodied in many methods and algorithms for many fields [99, 112]. Granular computing, as a field of study, attempts to extract the commonalities from existing fields to establish a set of generally applicable principles [92]. Previous work on granular computing has developed some basic concepts and theories, which paves the way for future research in granular computing.

2.3.1 Granules and Granulations

The first concept in granular computing is granule. A granule is a set of basic elements of a problem, which provides a partial view of the problem. The second concept in granular computing is granulation. A granulation is a group of granules, and the union of these granules contains all elements in a problem at a particular level. A granulation provides an overall view of a problem at this particular level. Basically, from the methodological perspective and the computational perspective, granular computing is to research how to create granules and granulations, and how to solve a problem by using these granules and granulations. The formal definitions of these concepts are given as follows, based on the set-theoretic fundament of granular computing proposed by Yao et al. [104]

Definition 2.1. (Granules and granulations) *Suppose U is a finite nonempty set of all the basic elements of a problem, then any non-empty subset $g \in 2^U$ is a* granule *of this problem, where 2^U is the power set of U. If G is such a group of granules that any two granules in G are disjoint and the union of all granules in G is U, then G is a granulation of this problem.*

Let us take the three-disk Hanoi tower problem as an example to illustrate the main ideas (the description of this problem is in Section 3.1.1). We can take a state in the state space as a basic element. Then U is the state set, any non-empty subset of the state set is a granule. We can take $\{s_1, s_2, \cdots, s_9\}$ as the first granule g_1, $\{s_{10}, s_{11}, \cdots, s_{18}\}$ as the second granule g_2, and $\{s_{19}, s_{20}, \cdots, s_{27}\}$ as the third granule g_3, then $\{g_1, g_2, g_3\}$ is a granulation of this problem.

Note that in this definition of granulation, any two granules are disjoint. In fact, this disjoint condition is not mandatory in granular computing, other definitions of granulation may exist in the literature [105,111]. In this thesis, we only investigate granulations satisfying the disjoint condition, that is, granulations corresponding to partitions of U [16, 41].

A granule may only contain one basic element, this granule is called *trivial granule* or *basic granule*. A granulation may only have trivial granules, this granulation is called *trivial granulation* or *basic granulation*. The identification of basic elements depends on specific problems. There may be different granulations for a problem. For example, for a time series data mining problem, a basic element may be the statistic information in one hour. A granule may be the statistic information of all hours in a day, and the granulation made of these kind of granules gives an overall view of the problem at the level of days. A granule can also be the statistic information of all hours in a week, and the granulation made of these kind of granules gives an overall view of the problem at the level of weeks.

2.3.2 Granulation Relations

There are some basic relations between granules, derived from set relations [39, 85, 104].

Definition 2.2. (Refinement-coarseness and equivalence relations for granules)

Refinement-coarseness relation: Given two granules $g_1, g_2 \in 2^U$*, if* $g_1 \subset g_2$*, then* g_1 *is a refined granule of* g_2*, or* g_2 *is a coarse granule of* g_1*, denoted by* $g_1 \prec g_2$*.*

Equivalence relation: If $g_1 = g_2$*, then* g_1 *is equivalent to* g_2*.*

We can use $g_1 \preceq g_2$ *to denote that either* g_1 *is a refined granule of* g_2 *or* $g_1 = g_2$*.*

From the basic relations between granules, we also have basic relations between granulations.

Definition 2.3. (Refinement-coarseness and equivalence relations for granulations)

Refinement-coarseness relation: Given two granulations G_1 *and* G_2*, if* $\forall g_1 \in G_1 \exists g_2 \in G_2(g_1 \preceq g_2)$ *and* $\exists g_1 \in G_1 \exists g_2 \in G_2(g_1 \prec g_2)$*, then* G_1 *is a* refined granulation *of* G_2 *or* G_2 *is a* coarse granulation *of* G_1*, denoted by* $G_1 \ll G_2$*.*

Equivalence relation: If $\forall g_1 \in G_1 \exists g_2 \in G_2(g_1 = g_2)$ *and* $\forall g_2 \in G_2 \exists g_1 \in G_1(g_2 = g_1)$*, then* G_1 *is equivalent to* G_2*, denoted by* $G_1 = G_2$*.*

The refinement-coarseness and equivalence relations are important relations in granular computing. As they link individual granules and granulations together, they are indispensable ingredients in many models like the top-down progressive computing model, which will be discussed later.

2.3.3 Information Tables and DL-Language

For a problem with a basic element set U, we can have $2^{|U|} - 1$ granules, and many more granulations. Not all these granules and granulations are useful for solving the problem. In order to create useful granules and granulations, to discover relations between granules and granulations, and to solve a problem by granules and granulations, we need representation tools to describe granules and granulations. Many tools have been proposed in the literature [23,40,55,56,58,70, 109], among which two important tools are information tables [51, 106] and DL-Language [51] that describe the semantic meaning of granules and granulations.

An information table is a widely used representation tool to describe the semantic meanings of basic elements, granules and granulations in a clear, simple way [52]. The formal definition of an information table is given as follows [51,106].

Definition 2.4. (Information table) *An information table for a universe of a domain is a tuple* $(U, At, \{V_a | a \in At\}, \{I_a | a \in At\})$*, where*

U is a finite nonempty set of all the objects in the universe of the domain,

At is a finite nonempty set of attributes,

V_a is a finite nonempty set of values for $a \in At$, and

I_a: $U \longrightarrow V_a$ is an information function for $a \in At$.

Each information function I_a is a total function that maps an object of U to exactly one value in V_a. The contents of the four elements U, At, $\{V_a | a \in At\}$,

$\{I_a | a \in At\}$ depend on specific problems. Before creating an information table, we should analyze the problem that we want to solve, then we can create an information table according to the problem. For example, if the problem is to diagnose patients, U could be the set of patients such as $John$, $Mike$; At could be the set of symptoms such as blood pressure, body temperature, etc.; V defines values for every symptoms, for example, $V_{body\ temperature} = \{35, 36, 37, 38, 39\}$ means that the values for body temperature could be 35, 36, 37, 38, 39; I gives values to symptoms for every patient, for example, $I_{body\ temperatur}(John) = 37$ means that $John$'s body temperature is 37 degree. An information table can be conveniently presented in a table form. Table 2.1 shows an information table. There are nine objects and four attributes. Every object has a value on every attribute. For example, object o_1 has value *short* on attribute $Height$, value *blond* on attribute $Hair$, value *brown* on attribute $Eyes$ and value *heavy* on attribute $Weight$.

An equivalence relation between two objects can be defined in terms of a subset of attributes [106].

Definition 2.5. (Equivalence relation induced by a set of attributes) *For a subset of attributes $A \subseteq At$, the equivalence relation induced by A is defined as*

$$xE_Ay \Longleftrightarrow (\forall a \in A) I_a(x) = I_a(y). \tag{2.1}$$

That is, two objects are equivalent in terms of A if and only if they have the same value for every attribute in A.

In order to make the representation and inference for granules and granula-

$Object$	$Height$	$Hair$	$Eyes$	$Weight$
o_1	$short$	$blond$	$brown$	$heavy$
o_2	$short$	$blond$	$brown$	$light$
o_3	$tall$	red	$blue$	$median$
o_4	$tall$	$dark$	$brown$	$light$
o_5	$tall$	$dark$	$brown$	$heavy$
o_6	$tall$	red	$blue$	$heavy$
o_7	$tall$	$dark$	$brown$	$median$
o_8	$short$	$blond$	$brown$	$median$
o_9	$short$	$blond$	$brown$	$median$

Table 2.1: An information table

tions easier, we can introduce certain logic languages defined on an information table. Pawlak [51] proposed a decision logic language (DL-language) on an information table.

Definition 2.6. (DL-Language) *The formulas in DL-language on* $(U, At, \{V_a | a \in At\}, \{I_a | a \in At\})$ *are recursively defined as follows:*

- (a, v) *is an atomic formula in DL-language, where* $a \in At$, $v \in V_a$.
- *If* ϕ *and* ψ *are formulas in DL-language, so are* $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$ *and* $\neg\phi$.
- *Nothing else is a formula in DL-language.*

A DL-language on $(U, At, \{V_a | a \in At\}, \{I_a | a \in At\})$ can be shortly called DL-language when the context is clear.

The satisfiability of a formula ϕ by an object x, written $x \models \phi$, is interpreted as follows [94].

(1) $x \models (a, v)$ iff $I_a(x) = v$,

(2) $x \models \neg\phi$ iff *not* $x \models \phi$,

(3) $x \models \phi \wedge \psi$ iff $x \models \phi$ *and* $x \models \psi$,

(4) $x \models \phi \vee \psi$ iff $x \models \phi$ *or* $x \models \psi$,

(5) $x \models \phi \rightarrow \psi$ iff $x \models \neg\phi$ *or* $x \models \psi$,

(6) $x \models \phi \equiv \psi$ iff $x \models \phi \rightarrow \psi$ *and* $x \models \psi \rightarrow \phi$.

The interpretation of a formula is the set of all the objects that satisfy the formula [94].

Definition 2.7. (Meaning of formula) *The meaning of a formula* ϕ *is* $m(\phi) = \{x \in U | \ x \models \phi\}$.

According to Definition 2.7 and satisfiability of formulas, we can easily get the

following properties [102]:

$$m(a, v) = \{x \in U | I_{a(x)} = v\},$$

$$m(\neg\phi) = U - m(\phi),$$

$$m(\phi \wedge \psi) = m(\phi) \cap m(\psi),$$

$$m(\phi \vee \psi) = m(\phi) \cup m(\psi),$$

$$m(\phi \rightarrow \psi) = (U - m(\phi)) \cup m(\psi),$$

$$m(\phi \equiv \psi) = (m(\phi) \cap m(\psi)) \cup ((U - m(\phi)) \cap (U - m(\psi)))$$

That is, logical connectives are characterized by set-theoretic operators.

2.3.4 Definable Granules and Granulations by DL-Language

By the meaning sets of formulas, we can divide all granules on U into two categories, namely, definable and non-definable granules [86,103]. A definable granule means that this granule can be expressed by a formula, a non-definable granule means that this granule cannot be expressed by any formula.

Definition 2.8. (Definable and non-definable granules) *Suppose g is a granule. If there is a formula ϕ such that $g = m(\phi)$, then g is definable, otherwise g is non-definable.*

An example of a non-definable granule is the trivial granule $\{o_8\}$ in Table 2.1. Because $o_8 E_{At} o_9$, o_8 and o_9 have the same value for every attribute in At, any formula that can be satisfied by o_8 can also be satisfied by o_9, so there does not exist a formula that has only $\{o_8\}$ as its meaning.

There is a special kind of granule that can be expressed by a conjunction of atomic formulas [53].

Definition 2.9. (Conjunctively definable granules) *Suppose g is a granule. If there is a formula ϕ such that $\phi = \bigwedge_{1 \leq i \leq n} P_i$, where $1 \leq n$, P_i is an atomic formula and $g = m(\phi)$, then g is conjunctively definable.*

A conjunctively definable granule has the semantics that all objects in the granule have the same values on a set of some attributes. Objects in a conjunctively definable granule share the same characteristics. For example, in Table 2.1, the granule $\{o_1, o_2, o_8, o_9\}$ is a conjunctively definable granule, because there is a formula $(Height, short) \bigwedge (Hair, blond)$ whose meaning is the granule. All objects in this granule share a characteristic, that is, they have the same values on $Height$ and $Hair$.

For the purpose of our research of granular state space search, we define two new concepts, which are conjunctively definable granulation and attribute reduct.

Definition 2.10. (Conjunctively definable granulations) *Suppose G is a granulation, At is a set of attributes, $Q \subseteq At$. If for every $g \in G$, $g = m(\bigwedge_{a \in Q} (a, v_a))$, where $v_a \in V_a$, then G is a conjunctively definable granulation. We say Q is an attribute set of G, or G is defined by Q.*

A conjunctively definable granulation can be presented by an information table $(U', At', \{V'_a | a \in At'\}, \{I'_a | a \in At'\})$, where U' is the set of granules, At' is the attribute set of this granulation, $V'_a = V_a$, and $I'_a(g') = I_a(o)$, where

$Object$	$Height$	$Hair$
g_1	$short$	$blond$
g_2	$tall$	red
g_3	$tall$	$dark$

Table 2.2: The information table for a granulation

$a \in At'\, g' \in U', o \in g'$. Table 2.2 is an information table of a granulation defined by $\{Height, Hair\}$ in Table 2.1. Granule g_1 is $\{o_1, o_2, o_8, o_9\}$, granule g_2 is $\{o_3, o_6\}$ and granule g_3 is $\{o_4, o_5, o_7\}$.

A granulation may be defined by different sets of attributes. For example, in Table 2.1, there are three sets of attributes that define the granulation $G_1 = \{\{o_1, o_2, o_8, o_9\}, \{o_3, o_6\}, \{o_4, o_5, o_7\}\}$, which are $Q_1 = \{Height, Hair, Eyes\}, Q_2 = \{Height, Hair\}$ and $Q_3 = \{Height, Eyes\}$. In the set Q_1, either the attribute $Hair$ or the attribute $Eyes$ is redundant, because after removing either $Hair$ or $Eyes$ from Q_1, the remaining set can still define the granulation G_1. We can define a kind of sets of attributes that have no redundant attributes.

Definition 2.11. (Attribute reduct) *Suppose Q is an attribute set of G. If for any $a \in Q$, $Q - \{a\}$ is not an attribute set of G, then Q is an attribute reduct of G.*

Note that an attribute reduct is not unique for a granulation. In the above example, Q_2 and Q_3 are both attribute reducts of G_1.

2.3.5 Theorems

If two attribute sets have a refinement-coarseness relation, there is also a relation between the conjunctively definable granulations defined by these two attribute sets. We propose and prove some theorems about the relations of attribute sets and conjunctively definable granulations. These theorems will be used later for granular state space search.

Theorem 2.1. *If $At_1 \subset At_0 \subseteq At$, G_0 is a conjunctively definable granulation defined by At_0 and G_1 is a conjunctively definable granulation defined by At_1, then $G_0 \ll G_1$ or $G_0 = G_1$.*

Proof: Let $Q = At_0 - At_1$ and g_0 be an arbitrary granule in G_0. Then,

$$\begin{aligned} g_0 &= m(\bigwedge_{a \in At_0}(a, v_a)) \\ &= m(\bigwedge_{a \in At_1}(a, v_a) \wedge \bigwedge_{b \in Q}(b, v_b)) \\ &= m(\bigwedge_{a \in At_1}(a, v_a)) \cap m(\bigwedge_{b \in Q}(b, v_b)) \quad (by\ m(\phi \wedge \psi) = m(\phi) \cap m(\psi)) \\ &\subseteq m(\bigwedge_{a \in At_1}(a, v_a)) \end{aligned}$$

According to Definition 2.10 $m(\bigwedge_{a \in At_1}(a, v_a))$ is a granule in G_1, so $g_0 \prec m(\bigwedge_{a \in At_1}(a, v_a))$ or $g_0 = m(\bigwedge_{a \in At_1}(a, v_a))$. This means that for any granule g_0 in G_0 there is a granule in G_1 that is equivalent to g_0 or is a coarse granule of g_0, so $G_0 \ll G_1$ or $G_0 = G_1$. The statement is proved. ■

Theorem 2.2. *Let G_0 and G_1 be two conjunctively definable granulations. If $G_0 \ll G_1$, then there exist two attribute sets Q_0 and Q_1 such that $Q_1 \subset Q_0$, Q_0 is an attribute set of G_0 and Q_1 is an attribute set of G_1.*

Proof: According to Definition 2.10, there is an attribute set At_0 of G_0 and an attribute set At_1 of G_1. Now we will prove that $At_0 \cup At_1$ is also an attribute set of G_0. Assuming g_0 is an arbitrary granule in G_0, according to Definition 2.3, there is a granule $g_1 \in G_1$ such that $g_0 \prec g_1$ or $g_0 = g_1$, then $g_0 = g_0 \cap g_1$. As $g_0 = m(\bigwedge_{a \in At_0}(a, v_a))$ and $g_1 = m(\bigwedge_{b \in At_1}(b, v_b))$, according to $m(\phi \wedge \psi) = m(\phi) \cap m(\psi)$, $g_0 = g_0 \cap g_1 = m(\bigwedge_{a \in At_0}(a, v_a)) \cap m(\bigwedge_{b \in At_1}(b, v_b)) = m(\bigwedge_{a \in At_0}(a, v_a) \wedge \bigwedge_{b \in At_1}(b, v_b)) = m(\bigwedge_{a \in At_0 \cup At_1}(a, v_a))$. Thus, $At_0 \cup At_1$ is an attribute set of G_0.

Note that At_0 is not equivalent to At_1, otherwise they define the same granulation. At_0 is not a subset of At_1, otherwise according to Theorem 2.1, G_0 cannot be a refined granulation of G_1. So $At_0 \cup At_1 \supset At_1$. When we take $At_0 \cup At_1$ as Q_0 and At_1 as Q_1, the statement is proved. ■

Theorem 2.1 and Theorem 2.2 can be extended to the situation of more than two granulations.

Theorem 2.3. *If $At_n \subset At_{n-1} \subset \cdots \subset At_1 \subset At_0 \subseteq At$, G_i is defined by At_i for $0 \leq i \leq n$, then $G_i \ll G_{i+1}$ or $G_i = G_{i+1}$ for $0 \leq i < n$.*

Proof: This statement is an easy extension of Theorem 2.1. ■

Theorem 2.4. *If $G_0, G_1, \cdots, G_n$ is a sequence of conjunctively definable granulations such that $G_i \ll G_{i+1}$ for $0 \leq i < n$, then there exists a sequence of sets of attributes $At_0 \supset At_1 \supset \cdots \supset At_n$ such that At_i is an attribute set of G_i for $0 \leq i \leq n$.*

Proof: We prove this statement by induction. From Theorem 2.2 we know that this statement holds for two granulations. Assume that this statement holds

for n granulations. For $n+1$ granulations $G_0, G_1, \cdots, G_n$, there is a sequence of sets of attributes $At_1 \supset \cdots \supset At_n$ such that At_i is an attribute set of G_i for $1 \leq i \leq n$. There is an attribute set Q of G_0. As $G_0 \ll G_1$, by the same reasoning in the proof of Theorem 2.2, we know that $Q \cup At_1$ is an attribute set of G_0 and $Q \cup At_1 \supset At_1$. We take $Q \cup At_1$ as At_0, then $At_0 \supset At_1 \supset \cdots \supset At_n$ is a sequence of attribute sets of $G_0 \ll G_1 \ll \cdots \ll G_n$. The statement is proved. ■

The Theorem 2.3 implies that we can find a granulation hierarchy by finding an ordered sequence of subsets of attributes.

2.4 Top-Down Progressive Computing

The triarchic theory describes granular computing at a much abstract, or coarse-grain, level of granularity. To make the conceptual model practically useful, we must further develop and refine the theory at more concrete and fine-grain levels. More specifically, we must develop and formulate concrete methodologies and computational algorithms of granular computing.

A problem may have different granulations that provide different views of the problem. These different granulations may be organized into a hierarchical granular structure that provides views of a problem from different levels [59]. This hierarchical granular structure in granular computing suggests three possible models of computation, namely, top-down, bottom-up and middle-out approaches. Based on the top-down approach, we suggest a concrete computing model for granular computing, which is the *top-down progressive computing model* [101].

2.4.1 Granulation Hierarchy

A granulation hierarchy [17, 18, 71, 97] is a multilevel granular structure, which is a fundamental structure in granular computing. A granulation hierarchy has different levels, every level is populated by granules of the same granularity or similar nature. Levels are partially ordered by their granularity. A level is a representation of a problem and may serve a particular purpose. A granulation hierarchy gives multiple representations of the same problem.

The basic ideas of granulation hierarchy can be illustrated by maps with different granularity. A world map can be seen from the level of countries. Every country is treated as a basic unit and a world map is represented by the countries and their connections. At the next level, there is a map for each country, where every city is a basic unit. A map is represented by cities and their connections. At yet another level, each city has its own map where different regions are basic units. In this way, more detailed maps can be further developed. Many versions of maps thus provide a granulation hierarchy.

According to Yao [101], a granulation hierarchy has some desirable properties:

- Granules in a particular level are relatively independent or loosely related. Each granule provides a local, partial description and all granules in the level collectively provide a global, complete description.

- A granulation hierarchy offers multiple representations and descriptions of the same problem. Representations at different levels must be consistent with each other. A problem in a lower level must have a corresponding

problem in a higher level; a solution to a higher level problem can serve as a guide for searching for a solution to the same problem at a lower level.

- Although different levels in a granulation hierarchy represent the same problem, different vocabularies and languages may be used to define and interpret granules at different levels and different methods may be used for processing at different levels.

- Levels are partially or totally ordered by their granularity. A lower level granulation has more detailed information than a higher level granulation. That is, a lower level granulation is a refinement of a higher level granulation, and a higher level is an abstraction of a lower level. Granules in a particular level are used to explain larger granules in the next higher level and, at the same time, are explained by smaller granules in the next lower level.

- The number of levels is not fixed a priori. It is possible to combine several levels into one or to split a level into more levels in the construction of a granular structure.

- Interactions between granules are mostly restricted to three levels, namely, the current level, the next lower level, and the next higher level. This restriction would greatly reduce the complexity of problem solving.

- There are bidirectional influences between two adjacent levels. A higher level typically controls and specifies its next lower level. Conversely, a lower

level may also causes restructuring of its next higher level.

- There are at least two basic operations for level transformation. A refinement or zoom-in operation that transforms a representation or description in a higher level into one in the next lower level. An important feature of this operation is the addition or fill-in of details. A coarsening, abstraction, or zoom-out operation transforms a representation or description in a lower level into one in the next higher level. An important feature is the omission of details, so that only crucial information is preserved.

A granulation hierarchy therefore provides structured descriptions of a problem with multiple levels of abstraction and details. This useful structure serves as a basis of structured approaches of granular computing.

Formally, we can define granulation hierarchy based on the refinement-coarseness relation between granulations.

Definition 2.12. (Granulation hierarchy) *Suppose* G_0 *and* G_1 *are two granulations. If* $G_0 \ll G_1$*, then the sequence of* $\{G_0, G_1\}$ *forms a two-level granulation hierarchy. Generally, if there is a sequence of granulations* $\{G_0, G_1, \cdots, G_{n-2}, G_{n-1}\}$ *such that* $G_i \ll G_{i+1}$ *for* $0 \leq i < n-1$*, then this sequence is an* n*-level granulation hierarchy.*

In a granulation hierarchy, G_0 is usually called the original or ground level granulation and the last granulation is usually called the top level granulation.

2.4.2 A Basic Progressive Computing Algorithm

One way of solving problems by granulation hierarchies is to process granulations one by one from the coarsest granulation to the finest granulation, which is called *top-down progressive computing*.

The top-down progressive computing model involves two basic tasks, namely, to build a granulation hierarchy and to compute with this granulation hierarchy. We can build a granulation hierarchy from top to bottom in a progressive manner. Once a granulation hierarchy is constructed, the same top-down progressive computing model is used to find a solution to a problem. We find an inaccurate solution in a coarser granulation and use the solution as a guide for finding a more accurate solution in a finer granulation. This process is repeated to refine progressively a solution until a satisfactory solution is found. The two tasks can be performed simultaneously in a top-down progressive way.

The top-down progressive computing model may speed up the problem solving process. The number of granules in a higher level is usually smaller than that in a lower level. Finding a solution in a higher level is relatively faster. Once a solution in a higher level is found, it can serve as a guidance to find a solution in a lower level, which may avoid unnecessary work in a lower level.

In specifying a top-level, generic progressive computing algorithm, we consider the following basic ingredients:

- Multiple representations of a problem. This is the basis of progressive computing model. In fact, progressive computing model is a sequence of refine-

ments from coarse descriptions at higher levels to refined descriptions at lower levels.

- Refinement operation on granulations. A granulation refinement operation is needed so that one can transform a coarser description into a finer description. It is important that a refinement operation preserves certain consistent properties of different descriptions.

- Refinement operation on solutions. A solution refinement operation revises an inaccurate, approximate solution in a higher level into a more accurate solution in a lower level.

- Evaluation function. An evaluation function measures the fitness of a solution obtained in a level. Once a satisfactory solution is obtained, one can stop the progressive computing process.

By utilizing these ingredients, a basic progressive computing algorithm (BPC) is given in Algorithm 2.1. In BPC, *GranulationRefinement* and *SolutionRefinement* are granulation refinement and solution refinement operations, respectively. G_0 and P_0 are some initial values for granulation (G) and solution (P). In the very first step, G_1=*GranulationRefinement(*G_0*)* produces the coarsest granulation G_1 in the granulation sequence G_1,G_2,G_3,$\cdots$ and P_1=*SolutionRefinement*(P_0, G_1) is the very first solution in the solution sequence $P_1, P_2, \cdots$. BPC uses a function $Fitness$ to evaluate a solution, and the process terminates once a satisfactory solution is found. It should be noted that sometimes the refined granulation itself is a solution. In this case, the *SolutionRefinement* step may be simply

removed by using G_k as a solution.

Algorithm 2.1 BPC: Basic progressive computing algorithm

1: $k \leftarrow 0$

2: set G_0 and P_0

3: **repeat**

4: $\quad k \leftarrow k + 1$

5: $\quad G_k = GranulationRefinement(G_{k-1})$

6: $\quad P_k = SolutionRefinement(P_{k-1}, G_k)$

7: $\quad F_k = Fitness(P_k)$

8: **until** F_k satisfies a certain condition

2.5 Conclusion

In this chapter, we review the triarchic theory for casting granular computer as a field of study. We study the relations between granular computing and artificial intelligence, and point out that the models and principles in granular computing can be used in artificial intelligence. We introduce the basic concepts and notions of granules, granulations, information tables and DL-language, and explain how to represent granules and granulations in an information table by using DL-language. We also suggest a top-down progressive computing model for granular computing. This model will be used later to solve state space search problems in artificial intelligence.

Based on the results of this chapter, in the following chapters, we will propose new concepts and concrete granular computing methods, and explain how to use

the top-down progressive computing model to model state space search problems.

Chapter 3

HIERARCHICAL STATE SPACE SEARCH

In artificial intelligence (AI), state space search is a classical approach for solving many problems, such as automatic planning and scheduling [8, 47], automated theorem proving [25], etc. To speed up state space search, many methods have been proposed. Hierarchical state space search is a widely used class of state space search methods. The two major issues about hierarchical state space search are the definition of a good abstraction hierarchy and the construction of good abstraction hierarchies. This chapter addresses the first issue and introduces the existing work on the second issue.

This chapter first introduces state space and abstraction hierarchy, then explains some refinement procedures that use abstraction hierarchies to search for solutions, and points out that only good abstraction hierarchies can be used by the ordered refinement procedure. Finally, this chapter identifies the characteris-

tics of good abstraction hierarchies and reviews existing methods for constructing good abstraction hierarchies.

3.1 Non-hierarchical State Space

3.1.1 State Space and State Space Search

A *state space* for a problem is a pair of set $SP = (S, O)$, where S is a finite non-empty set of *states* and O is a finite non-empty set of *operators*. A *state* is a configuration of basic elements in the problem, and an *operator* is a transformation rule in the problem, which can transform one state into another state. If an operator o transforms s_1 to s_2, we denote it by $s_1 \xrightarrow{o} s_2$. A problem has one *start state* and one or more *goal states*. Any two states can form a problem if we take one as the start state and the other as a goal state.

A *solution* to a problem is a sequence of operators that transform the start state into a goal state. The operators in a solution transform the start state into a sequence of *intermediate states* before a goal state is reached. Thus, a solution can also be expressed as a sequence of the start state, intermediate states and a goal state. The expression of a solution by a sequence of operators is called *operator expression*, while the expression of a solution by a sequence of states is called *state expression*. A *non-cyclic solution* is a solution in which no two states are identical. In this thesis, all the solutions we talk about are non-cyclic solutions.

Different solutions have different operator expressions but may have the same

state expression. This is because two operators may transform a state into the same state. For example, if both o_1 and o_2 transform s_{start} to s_1, and o_3 transforms s_1 to s_{goal}, then both o_1o_3 and o_2o_3 are solutions, and they have the same state expression $s_{start}s_1s_{goal}$.

A solution to a problem in a state space is usually found by an approach called *state space search*, which is widely used in artificial intelligence such as automatic planning [10, 42, 62]. The performance of a state space search approach is measured by efficiency and effectiveness. Efficiency is determined by the average cost for finding a solution, and effectiveness is determined by whether or not a search approach can find a solution. The cost for finding a solution is usually measured by the number of states explored by a search approach. The fewer the explored states, the more efficient the search approach.

Example 3.1. (The example of the three-disk Hanoi tower problem) *We use the three-disk Hanoi tower problem as an example to illustrate concepts about state space. The three-disk Hanoi tower problem requires to move three disks with different sizes from one peg to another peg by using an intermediate peg. The constraints in this problem are that a large disk can never be placed on a small disk, only one disk can be moved from one peg to another at a time, and only the top disk on a peg can be moved. The pegs are peg* 1*, peg* 2*, peg* 3 *and the disks are disk A, which is the smallest one, disk B, which is the medium one, and disk C, which is the largest one. A state is a legal configuration of pegs and disks that satisfies the constraints. Thus, there are totally* 27 *states, which are shown*

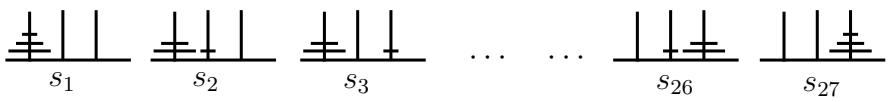


Figure 3.1: The states of the three-disk Hanoi tower problem

in Figure 3.1. The start state is that all disks are on peg 1 *(s_1) and the only one goal state is that all disks are on peg* 3 *(s_{27}). An operator is any movement of disks as long as this movement satisfies the constraints.*

3.1.2 Graphical Representation of a State Space

A labeled graph is a triple (V, E, L), where V is a set of vertexes, E is a set of edges and L is a set of labels. An edge is an ordered pair (v_1, v_2), where $v_1, v_2 \in V$, which means that this edge is from vertex v_1 to v_2, v_1 is the start vertex of the edge and v_2 is the end vertex of the edge. If for every edge $(v_1, v_2) \in E$, (v_2, v_1) is also an edge, this graph is a symmetric graph. In this thesis, every graph we discuss is a symmetric graph if we do not specify otherwise. Every edge can have a label. An *edge path* between two vertexes v_a and v_b is a sequence of edges $(v_1, v_1')(v_2, v_2') \cdots (v_{n-1}, v_{n-1}')(v_n, v_n')$ such that $v_a = v_1$, $v_b = v_n'$, $v_i' = v_{i+1}$ for $1 \leq i < n$. A *vertex path* between two vertexes v_a and v_b is a sequence of vertexes $v_1 v_2 \cdots v_{n-1} v_n$ such that $v_a = v_1$, $v_b = v_n$, $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$. A *non-cyclic vertex path* is a vertex path in which there are no identical vertexes.

A state space and a graph have natural relations. A state space can be represented by a graph, which is called *the graph of the state space*. The idea of graphical perspective of a state space has already been used by many re-

searchers, for example, Berge [5], Carre [15], Joslin [33], Manjari [26], Holte [31] and Helmert [27].

The graph representation of a state space (S, O) can be created as follows:

1. Every vertex v in the graph corresponds to a state s in the state space.

2. If v_1 corresponds to s_1, v_2 corresponds to s_2 and s_1 can be transformed into s_2 by one operator then $(v_1, v_2) \in E$. The label of edge (v_1, v_2) is the set of names of the operators that can transform s_1 to s_2.

A solution in a state space corresponds to an edge path or a vertex path from the start state to a goal state in the graph of the state space. From the graphical perspective, state space search approach is to search for a vertex path or edge path from the start state to a goal state in the graph of the state space.

Figure 3.2 shows the graph of the three-disk Hanoi tower problem's state space. In this graph, every edge in fact indicates two edges because every two states can be transformed from one to another and vice verse in this problem. We add two labels on each edge in the figure. A vertex path from s_1 to s_{27} in the graph corresponds to one solution in the state space.

3.1.3 Logic Language Representation of a State Space

Every state and operator have semantic meanings that can be described by languages. The language descriptions of all the states and operators form the language representation of a state space. For example, by using the natural language, the state s_1 in the three-disk Hanoi tower problem can be described as "*disk A*

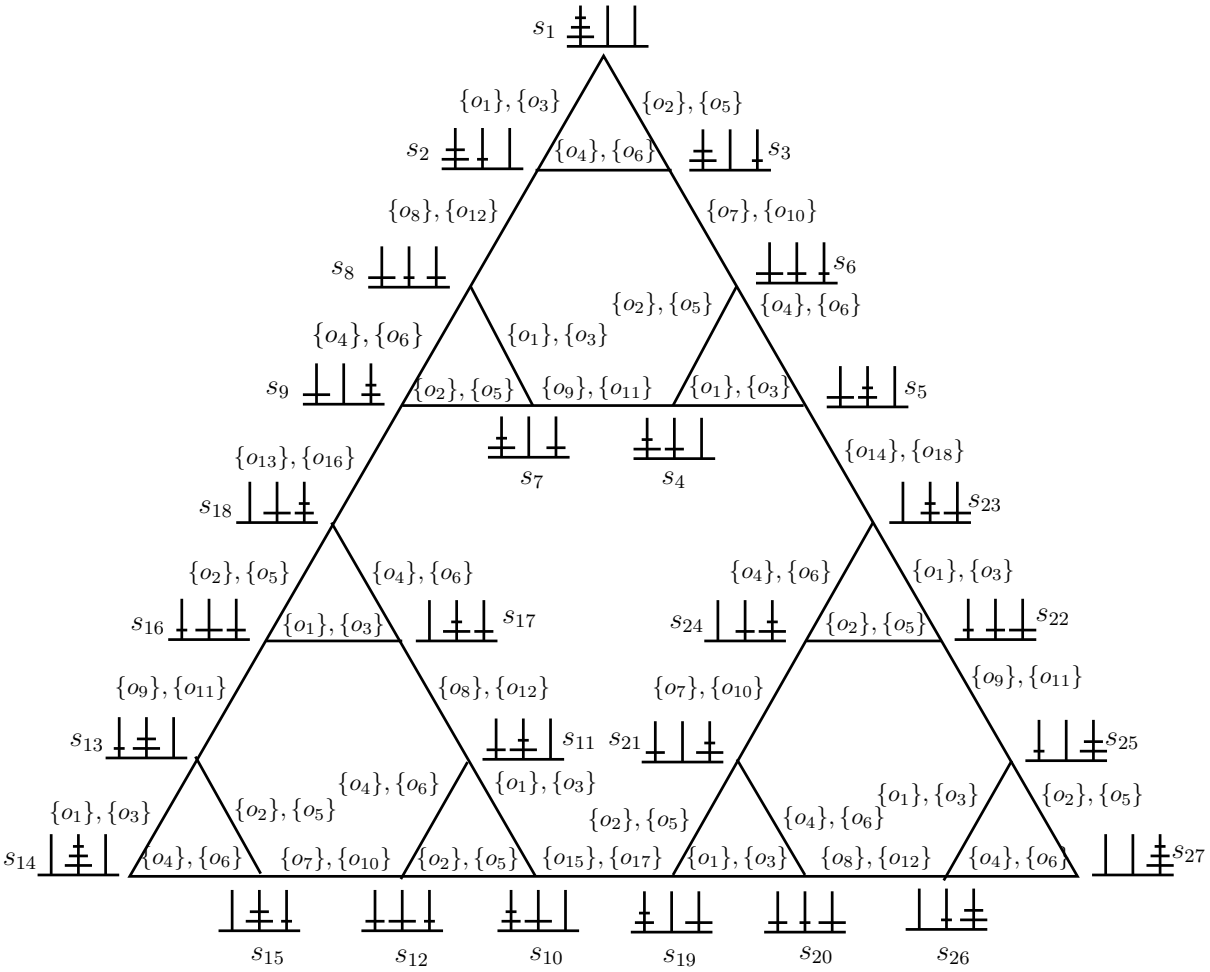


Figure 3.2: The graph for the three-disk Hanoi tower problem's state space (Knoblock [35])

is on *peg* 1, *disk* B is on *peg* 1 and *disk* C is on *peg* 1", the operator o_1 can be described as "if *disk* A is on *peg* 1 then move *disk* A to *peg* 2".

As the natural language is not suitable for logical analysis, formal languages are usually used to describe the state space, among which the first-order language is a widely used one [35]. By using the first-order language, a state is described as a non-empty set of literals; an operator is described as a triple (*Precondition*, *Add*, *Del*), where *Precondition*, *Add*, *Del* are sets of literals and *Add* and *Del* cannot be both empty. The triple of an operator means that if a state contains all the literals in *Precondition* then all the literals in *Add* are added into the state and all the literals in *Del* are deleted from the state, which results in the state being transformed into another state. In the three-disk Hanoi tower problem, there are nine literals $On(A,\ 1)$, $On(A,\ 2)$, $On(A,\ 3)$, $On(B,\ 1)$, $On(B,\ 2)$, $On(B,\ 3)$, $On(C,\ 1)$, $On(C,\ 2)$, $On(C,\ 3)$. $On(K,\ P)$ means that *disk* K is on *peg* P, where $K \in (A,\ B,\ C)$ and $P \in (1,\ 2,\ 3)$. The state s_1 can be described as $\{On(A,\ 1), On(B,\ 1), On(C,\ 1)\}$, the operator o_1 can be described as $(Precondition = \{On(A,\ 1)\}, Add = \{On(A,\ 2)\}, Del = \{On(A,\ 1)\})$. As the state s_1 contains the *Precondition* of o_1, o_1 can be applied to s_1 by adding $On(A,\ 2)$ into s_1 and deleting $On(A,\ 1)$ from s_1, which results in the state $\{On(A,\ 2), On(B,\ 1), On(C,\ 1)\}$.

3.2 Abstraction Hierarchy

3.2.1 Search by Abstraction Hierarchies

State space search is the most important approach for solving problems. The straightforward method for state space search is brute-force search or exhaustive search such as breadth-first search and depth-first search, which examines the edges or states one by one until a solution is found or until all reachable edges or all reachable states are examined and no solution is found. For example, in Figure 3.2, a breadth-first search starts by examining the edges (s_1, s_2), (s_1, s_3), (s_2, s_3), (s_2, s_8), (s_3, s_6), $\cdots$, until it examines the edge (s_{25}, s_{27}) and finds the solution $s_1 s_3 s_6 s_5 s_{23} s_{22} s_{25} s_{27}$.

Due to the phenomenon of combinatorial explosion, the brute-force search is costly, especially when the state space is quite large [14]. Therefore, heuristics are usually used to speed up the search process [3, 4, 113]. A heuristic serves as an advice to guide the search process. A popular heuristic based method is hierarchical state space search [50, 81, 95], which uses abstraction hierarchies as heuristics to guide the search and improve the search efficiency [30].

Hierarchical state space search embodies the idea of hierarchical problem solving in granular computing. It has two steps, which are the counterparts of the two basic tasks in top-down progressive computing (to build a granulation hierarchy and to compute with this granulation hierarchy). The first step is to construct an abstraction hierarchy and the second step is to solve problems by using this abstraction hierarchy top-down wisely [69]. In the first step, different abstract

versions of a state space, called abstractions or abstract state spaces, are created and organized hierarchically into an abstraction hierarchy according to the degree of their abstraction. An abstraction is a combination of abstract states and abstract operators. Abstract states and abstract operators are created by extracting the common characteristics from states and operators of the original state space. For example, in the three-disk Hanoi tower problem, the states s_1, s_2 and s_3 have the same characteristics that both *disk B* and *disk C* are on *peg* 1. We can create an abstract state s'_1 to represent the situation that both *disk B* and *disk C* are on *peg* 1.

In the second step, hierarchical state space search uses abstraction hierarchies to solve problems. It first finds an abstract solution in the top level abstraction, and then uses this abstract solution to guide the search for another abstract solution in the next lower abstraction. This process is repeated in a way that the solution searching is always guided by the already found abstract solution in the immediate higher abstraction until a solution in the original state space is found.

The philosophy behind hierarchical state space search is the principle of hierarchical problem solving implied by granular computing, that is, an abstract version of a state space presents the most important parts and central issues of a problem which should be solved first, and the trivial details and peripheral issues of a problem contained in the less abstract abstraction or the original state space could be considered later [35]. Theoretically, hierarchical state space search can find a solution faster than the exhaustive methods. This is because an abstraction has fewer states and operators than the original state space, therefore, the search

process in an abstraction is faster. Once a solution in an abstraction is found, it can be used to guide and expedite the search process in lower abstractions or the original state space. The whole search process in abstractions and the original state space is faster than directly searching in the original state space [35].

3.2.2 Definitions of Abstraction and Abstraction Hierarchy

The basic ingredients of hierarchical state space search are abstractions and abstraction hierarchies. The formal definition of an abstraction is given as follows.

Definition 3.1. (Abstraction) *Suppose $SP = (S, O)$ and $SP' = (S', O')$ are two state spaces, SP' is an* abstract state space *or* abstraction *of SP if and only if the following conditions are satisfied:*

1. *$|S'| < |S|$, $|O'| \leq |O|$, where $|S'|$ and $|S|$ are the numbers of states in S' and S, $|O'|$ and $|O|$ are the numbers of operators in O' and O.*

2. *There is a function pair (f_s, f_o), where $f_s : S \rightarrow S'$ is a function that maps S to S' and $f_o : O \rightarrow O'$ is a function that maps O to O'.*

3. *$\forall s' \in S' \exists s \in S(f_s(s) = s')$ and $\forall o' \in O' \exists o \in O(f_o(o) = o')$.*

4. *$s'_1 \xrightarrow{o'} s'_2$, if and only if $\exists s_1, s_2 \in S \exists o \in O(f_s(s_1) = s'_1 \wedge f_s(s_2) = s'_2 \wedge f_o(o) = o' \wedge s_1 \xrightarrow{o} s_2)$.*

If $f_s(s) = s'$ then s' is an *abstract state* of s and s is a *pre-image* of s'. All the pre-images of an abstract state form a *pre-image set* of this abstract state. If $f_o(o) = o'$ then o' is an *abstract operator* of o and o is a *pre-image* of o'. All the pre-images of an abstract operator form a *pre-image set* of this abstract operator. The abstract state of the start state in the original state space is called *abstract start state* and the abstract states of the goal states in the original state space are called *abstract goal states*. The abstract start state and abstract goal states form a problem in the abstraction, which is an abstract problem of the problem in the original state space. As a problem in the original state space may have more than one goal state, the abstract problem may also have more than one abstract goal state. If there is a solution in an abstraction which can transform the abstract start state into an abstract goal state, such solution is called *abstract solution*.

From the definition of abstraction we can derive a relation between two state spaces named *abstract relation*. If SP_b is an abstraction of SP_a, we can say that there is an abstract relation between SP_a and SP_b, denoted by $SP_a \ll SP_b$. The abstract relation is a transitive relation. If $SP_a \ll SP_b$ and $SP_b \ll SP_c$ then $SP_a \ll SP_c$.

Based on the abstract relation between abstractions, we can get the definition of abstraction hierarchy as follows.

Definition 3.2. (Abstraction hierarchy) *Suppose* (SP_0, SP_1, $\cdots$, SP_{n-2}, SP_{n-1}) *is a sequence of state spaces, where* $2 \leq n$. *If* $SP_i \ll SP_{i+1}$ *for* $0 \leq i < n-1$, *then this sequence is an* n-level abstraction hierarchy *of the* SP_0, *where*

SP_0 is the original state space or group level abstraction, and SP_i is the ith-level abstraction *for* $0 < i$. *The bigger* i, *the higher the level.*

Abstraction hierarchies are the basic structures for hierarchical state space search, we will introduce how to use abstraction hierarchies to solve state space search problems in the next section.

3.3 Refinement Procedure

Abstraction hierarchies are used by hierarchical state space search to search for solutions in a state space. One important procedure in hierarchical state space search is the *refinement procedure*, which is to refine an abstract solution at one level into a solution at the next lower level.

3.3.1 Basic Refinement Procedure

The principle of the basic refinement procedure is the idea of divide-and-conquer [63], which is dividing the next lower level problem into sub-problems according to the abstract solution in the higher level, and then solving these sub-problems one by one. After all the sub-problems have been solved, the problem on this level will be solved.

A basic refinement procedure has two parts, one is *operator refinement* and the other is *sub-problem solving*. The operator refinement chooses pre-images of abstract operators in the abstract solution, and determines the sub-problems. The sub-problem solving solves these sub-problems. For example, in Figure 3.3, there

is a 2-level abstraction hierarchy, the start state and goal state in the original state space are s_{start} and s_{goal}, the abstract start state and abstract goal state in this abstraction are s'_{start} and s'_{goal}. An abstract solution $o'_1 o'_2 \cdots o'_n$ is searched in the abstraction, the state expression of this abstract solution is $s'_{start} s'_1 \cdots s'_{n-1} s'_{goal}$. This abstract solution can be found by an exhaustive method such as breadth-first method or depth-first method. In the operator refinement part of the basic refinement procedure, a pre-image of o'_1 is first chosen that is o_1. The first sub-problem is identified, whose start state is s_{start} and goal states are all the states that can be transformed by o_1 to a pre-image of s'_1. In the sub-problem solving part, a sub-solution sub_1 is found which can transform s_{start} into one goal state s_1. The operator o_1 is applied to s_1 to transform it into s_1^* that is a pre-image of s'_1. The procedure goes to the operator refinement part again, where a pre-image of o'_2 is chosen that is o_2 and the second sub-problem is identified whose start state is s_1^* and goal states are all the states that can be transformed by o_2 to a pre-image of s'_2. The sub-problem solving part is used to find a sub-solution sub_2. The operator refinement part and sub-problem solving part are alternately used repeatedly until the final sub-problem is identified and solved. We can concatenate sub_1, o_1, sub_2, o_2, $\cdots$, sub_n, o_n, sub_{n+1} to get a solution to the original problem.

3.3.2 Restrained Refinement Procedure

The basic refinement procedure mentioned in the previous section uses an abstract solution as a control mechanism to search for a solution at the next lower

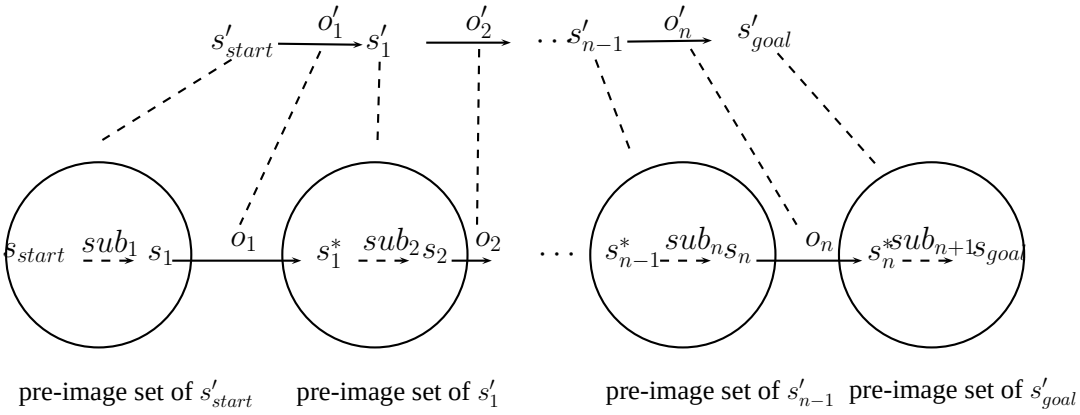


Figure 3.3: The basic refinement procedure

level. This control mechanism is not strict in the sense that a sub-solution to a sub-problem may go outside the pre-image set of the corresponding abstract state. Figure 3.4 shows such a situation. The abstract state s'_k and s'_{k+1} are two consecutive states in the abstract solution, and o'_{k+1} is an operator in the abstract solution. A pre-image of o'_{k+1} is chosen which is o_{k+1}, and a sub-problem is identified whose start state is s^*_k and goal states are all the states that can be transformed by o_{k+1} to a pre-image of s'_{k+1}. A sub-solution to this sub-problem is found which is $o_1o_2o_3$. The intermediate states in this sub-solution are s_q and s_p which are not in the pre-image set of s'_k. The go-outside phenomenon means that the abstract solution loses the functionality of guiding the solution search for sub-problems, which affects the efficiency of an abstraction hierarchy. In the worst situation, a solution to a sub-problem may go through almost all the states outside the pre-image set of the corresponding abstract state. In order to prevent the go-outside phenomenon, we need to add restrictions to the refinement procedure. One refinement procedure with additional restrictions is the *ordered*

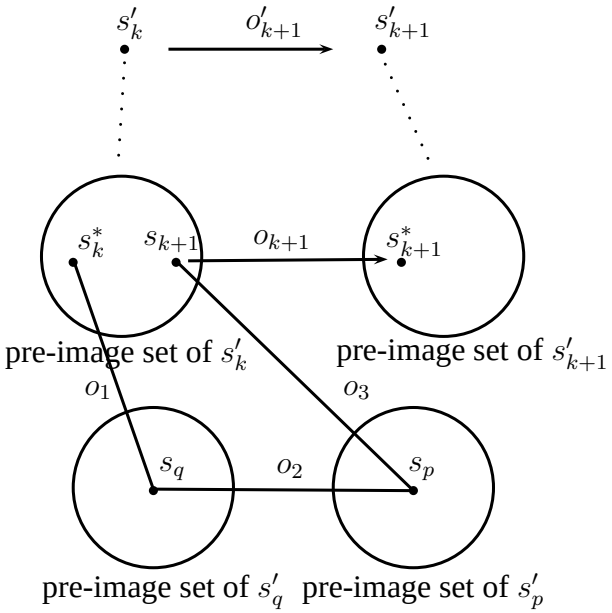


Figure 3.4: A solution going outside the pre-image set

refinement procedure.

Definition 3.3. (Ordered refinement procedure) *In a refinement procedure, if all states in the sub-solution to every sub-problem are in the pre-image set of the corresponding abstract state, this refinement procedure is an* ordered refinement procedure.

In an ordered refinement procedure, we can explore only the states that are within the corresponding pre-image set. For example, in Figure 3.5, when exploring possible succeeding operators for s_1, we can ignore the operators o_1, o_3, o_5 and o_6 because they will reach states outside the pre-image set of s'. Thus, by applying the ordered refinement procedure, the number of available operators in sub-problems are reduced and the search efficiency is improved.

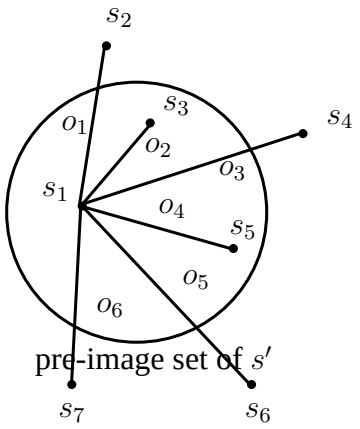


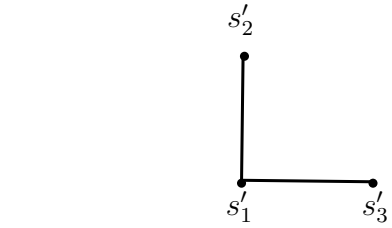
Figure 3.5: The ordered refinement procedure

3.3.3 An Issue of the Ordered Refinement Procedure

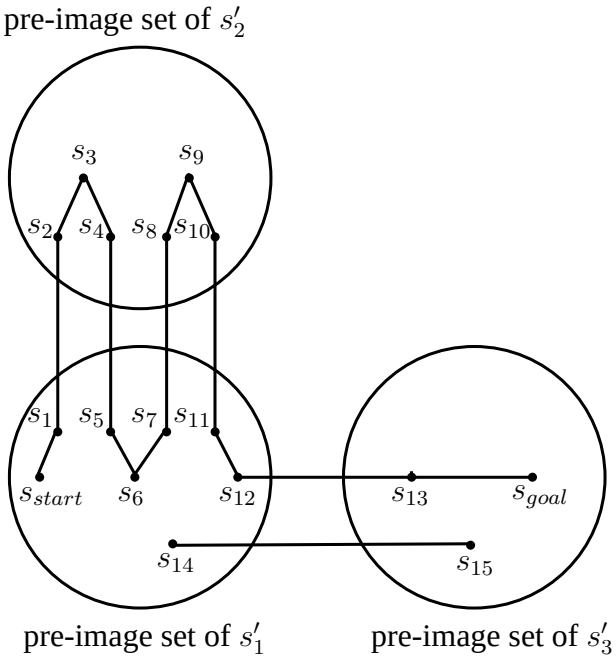
One remaining issue of the ordered refinement procedure is that it may not be complete. That is, it may fail to find any solution to a solvable problem.

Let us consider an example shown in Figure 3.6. This is a 2-level abstraction hierarchy, where the states s_{start}, s_1, s_5, s_6, s_7, s_{11} and s_{12} are in the same pre-image set of an abstract state s'_1, the states s_2, s_3, s_4, s_8, s_9, s_{10} are in the same pre-image set of an abstract state s'_2, and the states s_{13}, s_{goal} are in the same pre-image set of an abstract state s'_3.

We consider a solvable problem with s_{start} and s_{goal} as the start and goal state. By using this abstraction hierarchy, the only abstract solution is $s'_1 s'_3$, but this abstract solution cannot be refined by the ordered refinement procedure to a solution in the original state space. When mapping the solution in the original state space ($s_{start} s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{goal}$) into a path in the abstraction, it should be $s'_1 s'_2 s'_1 s'_2 s'_1 s'_3$. This means that only the path $s'_1 s'_2 s'_1 s'_2 s'_1 s'_3$



(a) Abstraction



(b) State space

Figure 3.6: An issue of the ordered refinement procedure

can be refined by the ordered refinement procedure into the solution in the original state space. But this path is not an abstract solution, because it contains cycles. Cycles are not allowed in any abstract solutions, otherwise there would be an infinite number of abstract solutions in an abstraction, which may cause the backtracking process to go endlessly. For example, if an abstract solution fails to find a solution in the original state space, the backtracking process will try all possible abstract solutions until a final solution is found or until all abstract solutions are tried and no final solution is found. If there are an infinite number of abstract solutions, the backtracking process may not stop. So the ordered refinement procedure cannot solve the solvable problem in Figure 3.6.

However, if we choose another abstraction hierarchy of the same original state space, we may use the ordered refinement procedure to find a final solution. For example, Figure 3.7 is another abstraction in which s_{start} and s_1 are in the same pre-image set of an abstract state s''_1; s_2, s_3 and s_4 are in the same pre-image set of s''_2; s_5, s_6 and s_7 are in the same pre-image set of s''_3; s_8, s_9 and s_{10} are in the same pre-image set of s''_4; s_{11} and s_{12} are in the same pre-image set of s''_5; s_{13} and s_{goal} are in the same pre-image set of s''_6; s_{14} and s_{15} are in the same pre-image set of s''_7. When solving the problem with s_{start} and s_{goal} as the start and goal states, the abstract solution is $s'_1 s'_2 s'_3 s'_4 s'_5 s'_6$, which can be refined to a final solution in the original state space. This abstraction can be used by the ordered refinement procedure to solve problems.

If we use the ordered refinement procedure to solve problems, we should choose appropriate abstraction hierarchies. If we do not choose good ones, we may not be

able to solve the problems. Concerned questions are: What are characteristics of good abstraction hierarchies that can solve problems with the ordered refinement procedure? How to construct such good abstraction hierarchies? We will answer these questions in the following sections.

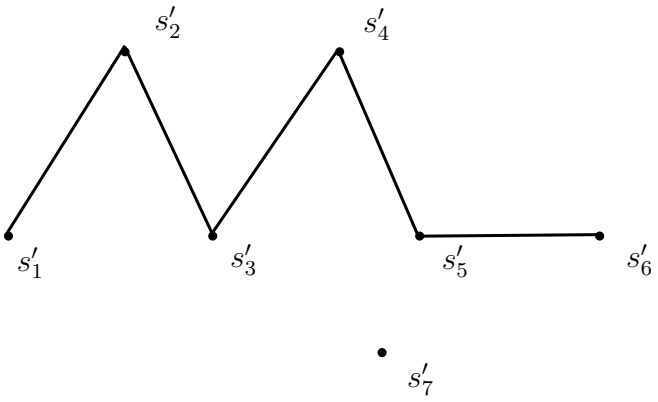
3.3.4 Characteristics of Good Abstraction Hierarchies

In this section, we discuss the characteristics of good abstraction hierarchies. We consider two criteria that a good abstraction hierarchy should satisfy, one is effectiveness and the other is efficiency. Effectiveness means that the abstraction hierarchy can find solutions to problems, and efficiency means that it finds the solutions quickly. There are two additional concepts, completeness degree and backtracking, which are related to the effectiveness and efficiency.

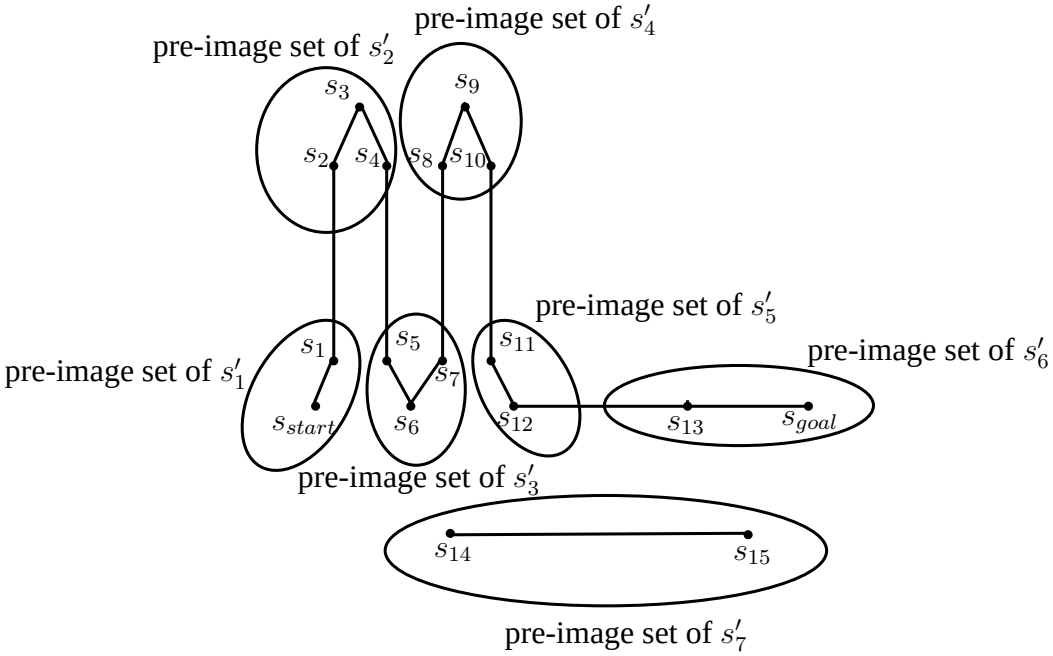
Completeness Degree

For the same abstraction hierarchy, some solvable problems can be solved by the ordered refinement procedure, while other solvable problems cannot be solved by the ordered refinement procedure. For example, in Figure 3.6, the problem (s_{start}, s_{goal}) can be solved while the problem (s_{start}, s_{15}) cannot be solved by the ordered refinement procedure. According to how many solvable problems an abstraction hierarchy can solve by the ordered refinement procedure, we can define a concept, named *completeness degree*, to measure the goodness of an abstraction hierarchy.

Definition 3.4. (Completeness degree and complete abstraction hierar-



(a) Abstraction



(b) State space

Figure 3.7: An example of another abstraction

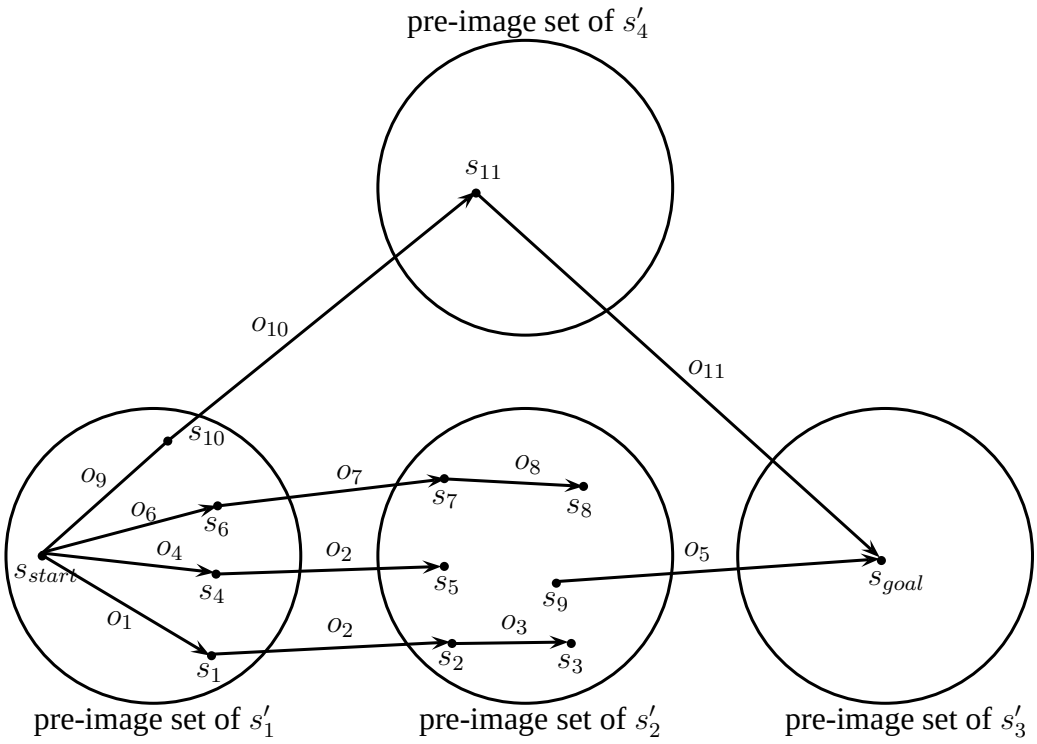
chy) *Suppose H is an abstraction hierarchy, n is the number of all the solvable problems in the original state space, and m is the number of all the solvable problems that can be solved by H with the ordered refinement procedure. Then $\frac{m}{n}$ is the* completeness degree *of H. If an abstraction hierarchy has a completeness degree of* 1*, then this abstraction hierarchy is called a* complete abstraction hierarchy.

The completeness degree measures the reliability of an abstraction hierarchy. The higher the completeness degree, the more solvable problems the abstraction hierarchy can solve with the ordered refinement procedure. If an abstraction hierarchy has a completeness degree of 1, it can solve all solvable problems with the ordered refinement procedure in the original state space.

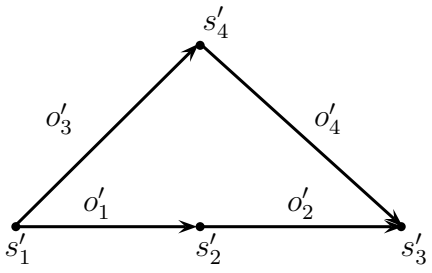
Backtracking

Abstraction hierarchies along with the ordered refinement procedure cannot always speed up the search process due to an immanent issue of the abstraction hierarchies, which is backtracking. Backtracking is a process of changing a previous partial solution when the refinement procedure fails to find a sub-solution to a sub-problem. Depending on the changed partial solution, there are three types of backtracking. *Goal state backtracking* is the process of changing the goal states for the previous sub-problem. *Operator pre-image backtracking* is the process of changing the pre-images of an abstract operator. *Abstract solution backtracking* is the process of changing the high level abstract solution.

Figure 3.8 shows an example of backtracking. In the refinement procedure, an abstract solution $s'_1 s'_2 s'_3$ or $o'_1 o'_2$ is found. For the first sub-problem, one of



(a) Graph of a state space



(b) Abstraction

Figure 3.8: Backtrackings

o'_1's pre-image o_2 is chosen. As two states s_1, s_4 in the pre-image set of s'_1 can be transformed by o_2 to a pre-image of s'_2, s_1 and s_4 are two goal states of the first sub-problem. A solution $s_{start}s_1$ is found for the first sub-problem, and the refinement procedure goes to the second sub-problem which has start state s_2. No solution to the second sub-problem can be found. The refinement procedure must take a goal state backtracking by revoking the solution to the first sub-problem and choosing a different goal state for the first sub-problem that is s_4. Another solution $s_{start}s_4$ for the first sub-problem is found. The refinement procedure still cannot find a solution to the second sub-problem. As all alternative goal states of the first sub-problem have been tried, the refinement procedure must take an operator pre-image backtracking by revoking the solution to the first sub-problem and choosing a different pre-image of abstract operator o'_1, which is o_7. A solution $s_{start}s_6$ to the first sub-problem is found. Still, no solution can be found for the second sub-problem. As all alternative pre-images of o'_1 have been tried, the refinement procedure must take an abstract solution backtracking by finding a different abstract solution, which is $s'_1s'_4s'_3$ or $o'_3o'_4$. From this abstract solution, a final solution $s_{start}s_{10}s_{11}s_{goal}$ can be found.

We know that backtracking reduces the efficiency of hierarchical state space search. Once a backtracking happens, a part of previous work is wasted.

Characteristics of Good Abstraction Hierarchies

Now we can argue that the two characteristics of good abstraction hierarchies are high completeness degree and few occurrences of backtracking. When construct-

ing abstraction hierarchies, we should make sure these abstraction hierarchies have completeness degree as high as possible and have occurrences of backtracking as few as possible. A perfect abstraction hierarchy is the one with completeness degree of 1 and without any backtracking. In the following part of this thesis, we will investigate how to create good abstraction hierarchies.

3.4 Existing Methods for Generating Good Abstraction Hierarchies

Recall that there are two types of methods for generating good abstraction hierarchies. The first type is to generate abstraction hierarchies implicitly by representing state spaces as a set of *state variables* [22] and analyzing these state variables. The second type is to generate abstraction hierarchies explicitly by exploring the explicit graphs of state spaces. A typical method for the first type is Knoblock's method, and a typical method for the second type is Holte et al.'s method. Therefore, in this thesis we choose and focus on the two methods proposed by Knoblock and Holte et al. This section reviews Knoblock's method, Holte et al.'s method and other explicit methods.

3.4.1 Knoblock's Method

The first work on how to generate good abstraction hierarchies was done by Knoblock [35]. Knoblock introduces a concept named a justified solution. Generally speaking, a justified solution is a solution in which every operator is indis-

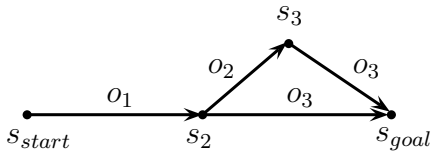


Figure 3.9: A non-cyclic but not justified solution

pensable. The idea of a justified solution is similar to that of a non-cyclic solution. But they are different in that a justified solution is defined from the perspective of operators and a non-cyclic solution is defined from the perspective of states. A justified solution is a non-cyclic solution, but a non-cyclic solution may not be a justified solution. In Figure 3.9, both solutions o_1o_3 and $o_1o_2o_3$ are non-cyclic solutions, but $o_1o_2o_3$ is not a justified solution because o_2 is dispensable in the sense that after o_2 is removed from this solution, the remaining sequence o_1o_3 is still a solution.

Based on justified solutions, Knoblock defines a kind of abstraction hierarchies, named *Ordered Monotonic Abstraction Hierarchy*. Generally speaking, it is an abstraction hierarchy that if there is a justified solution in the original state space, then there is a justified solution in every abstraction that can be refined to this justified solution in the original state space by the ordered refinement procedure. The idea behind the ordered monotonic abstraction hierarchy is like the idea of complete abstraction hierarchy. If an abstraction hierarchy is an ordered monotonic abstraction hierarchy, it is also a complete abstraction hierarchy.

Knoblock's method uses the first-order language to describe states and operators, and analyzes possible interactions between the literals in the operators. Ordered monotonic abstraction hierarchies are created to avoid the interactions

between the literals in the operators. In Knoblock's method, every literal is assigned a level value, which means that this literal can only appear in the abstractions lower than or equal to the level value. An abstraction is created by dropping all literals whose level values are less than a specific value. Knoblock gives a restriction that for every operator, the level values of literals in *Add* and *Del* should be the same and should be no smaller than those of the literals in *Precondition*. Knoblock proves that if an abstraction hierarchy satisfies this condition, then the abstraction hierarchy is an ordered monotonic abstraction hierarchy. He also gives an algorithm to analyze the operators and assign a level value to every literal.

We use the three-disk Hanoi tower problem to demonstrate Knoblock's method. There are 9 different literals and 18 operators in this problem as shown in Table 3.1. By analyzing possible interactions between the literals in these operators, we can assign level value 3 to $On(C,1)$, $On(C,2)$ and $On(C,3)$, assign level value 2 to $On(B,1)$, $On(B,2)$ and $On(B,3)$ and assign level value 1 to $On(A,1)$, $On(A,2)$ and $On(A,3)$. We can know that this assignment satisfies Knoblock's restriction. We can create an abstraction by setting a threshold and dropping off the literals whose level values are less than this threshold. If we set the threshold as 2, we then drop $On(A,1)$, $On(A,2)$ and $On(A,3)$ in all operators to create abstract operators in the 1st-level abstraction. For example, we drop off all the occurrences of $On(A,1)$, $On(A,2)$ and $On(A,3)$ in the operator 7, the result is an abstract operator $(\{On(B,1)\},\{On(B,2)\},\{On(B,1)\})$. If both *Add* and *Del* become empty after the drop-off in an operator, then no abstract operator is cre-

ated for that operator. For example, we drop off $On(A,1)$, $On(A,2)$ and $On(A,3)$ in operator 1 and result in an empty operator, which means that no abstract operator can be created from operator 1. We can also drop off $On(A,1)$, $On(A,2)$ and $On(A,3)$ in all states to create abstract states. These abstract states and the abstract operators are the 1st-level abstraction. Again, we can set the threshold as 3 and drop off $On(A,1)$, $On(A,2)$, $On(A,3)$, $On(B,1)$, $On(B,2)$, $On(B,3)$ to create abstract states and operators to form the 2nd-level abstraction.

The advantage of Knoblock's method is that the abstraction hierarchies created by this method has a completeness degree of 1. Thus, every solvable problem can be solved by the abstraction hierarchies created by this method.

One shortcoming of Knoblock's method is that this method is not widely applicable because the restriction for operators is too strong and a state space may not have abstraction hierarchies satisfying this restriction. For example, in the three-disk Hanoi tower problem, a new special rule may be added, which is *if disk C is on peg* 1*, disk B and disk A are on peg* 2*, then move disk B and disk A on peg* 3 *and move disk C on peg* 2. The corresponding first-order language operator is ($\{On(C,1),\ On(A,2),\ On(B,2)\}$, $\{On(C,2),\ On(A,3),\ On(B,3)\}$, $\{On(C,1), On(A,2), On(B,2)\}$). According to Knoblock's restriction, $On(C,1)$, $On(C,2)$, $On(A,2)$, $On(A,3)$, $On(B,2)$, $On(B,3)$ should have the same level value. After considering other operators, it is easy to know that all the nine literals should have the same level value to satisfy Knoblock's restriction. Thus, no abstraction can be built, because all the literals will be dropped off at the same time when we try to create an abstraction.

No.	Precondition	Add	Del
1	$\{On(A,1)\}$	$\{On(A,2)\}$	$\{On(A,1)\}$
2	$\{On(A,2)\}$	$\{On(A,1)\}$	$\{On(A,2)\}$
3	$\{On(A,1)\}$	$\{On(A,3)\}$	$\{On(A,1)\}$
4	$\{On(A,3)\}$	$\{On(A,1)\}$	$\{On(A,3)\}$
5	$\{On(A,2)\}$	$\{On(A,3)\}$	$\{On(A,2)\}$
6	$\{On(A,3)\}$	$\{On(A,2)\}$	$\{On(A,3)\}$
7	$\{On(B,1),\ On(A,3)\}$	$\{On(B,2)\}$	$\{On(B,1)\}$
8	$\{On(B,2),\ On(A,3)\}$	$\{On(B,1)\}$	$\{On(B,2)\}$
9	$\{On(B,1),\ On(A,2)\}$	$\{On(B,3)\}$	$\{On(B,1)\}$
10	$\{On(B,3),\ On(A,2)\}$	$\{On(B,1)\}$	$\{On(B,3)\}$
11	$\{On(B,2),\ On(A,1)\}$	$\{On(B,3)\}$	$\{On(B,2)\}$
12	$\{On(B,3),\ On(A,1)\}$	$\{On(B,2)\}$	$\{On(B,3)\}$
13	$\{On(C,1),\ On(A,3),\ On(B,3)\}$	$\{On(C,2)$	$\{On(C,1)\}$
14	$\{On(C,2),\ On(A,3),\ On(B,3)\}$	$\{On(C,1)\}$	$\{On(C,2)\}$
15	$\{On(C,1),\ On(A,2),\ On(B,2)\}$	$\{On(C,3)\}$	$\{On(C,1)\}$
16	$\{On(C,3),\ On(A,2),\ On(B,2)\}$	$\{On(C,1)\}$	$\{On(C,3)\}$
17	$\{On(C,2),\ On(A,1),\ On(B,1)\}$	$\{On(C,3)\}$	$\{On(C,2)\}$
18	$\{On(C,3),\ On(A,1),\ On(B,1)\}$	$\{On(C,2)\}$	$\{On(C,3)\}$

Table 3.1: The operators of the three-disk Hanoi tower problem

3.4.2 Holte et al.'s Method

Holte et al. [31] give a graph oriented method, called STAR algorithm, to create partitions on the graph of a state space for constructing abstractions. It builds partition blocks for the graph one at a time by picking a state to act as the hub of the partition block and gathering together all states that can be reached from the hub by a short path that does not pass through any other partition block [31]. The hub is selected randomly, and the maximum distance from the hub is specified by the user. When the partition of the graph is finished, an abstraction can be built according to the partition. Holte et al. [28] also give an algorithm *Hierarchical* A^* to search in the abstraction hierarchies efficiently.

Abstraction hierarchies built by Holte et al.'s method are complete abstraction hierarchies and can prevent backtracking. However, this method can only be used on explicit graphs, which means that all the states should be explored firstly for creating the explicit graph, which is not suitable for a large state space. Another problem of this method is that the abstractions created may not be semantically meaningful, because the hubs are selected randomly. The abstract states are no more than groups of randomly chosen original states and do not have meanings. Therefore, there are no abstract operators, and search in every level of abstraction should be carried out on an explicit graph without any operators.

3.4.3 Researches on Other Explicit Methods

Holte et al. are pioneers in researching on generating abstraction hierarchies explicitly, based on their work, other explicit methods are developed. Sturtevant and Jansen [78] improve Holte et al.'s method by making sure that all states in every partition block can be connected by edges into one line. Sturtevant and Buro [75] give another variant of improvement of Holte et al.'s method by making sure that every partition block is a clique, and propose a refinement procedure *Partial Refinement* A^* (PRA^*), which is suitable for clique-based abstraction. Later on, Sturtevant and Buro [76] enhance PRA^* to *Cooperative* PRA^* ($CPRA^*$) which is more efficient. Bulitko et al. [13] develop a search algorithm named *Path Refinement Learning Real-time Search* ($PR\ LRTS$), which can use clique-based abstraction to search state spaces efficiently. Botea et al. [11] provide the concept of sector-based abstraction and use a refinement procedure $Hierarchical\ Path\text{-}Finding\ A^*$ (HPA^*) to find paths in grid-based maps. Bulitko et al. [12] combine Holte et al.'s method with learning real-time search to speed up state space search. Sturtevant [74, 77] proposes a minimal-memory abstraction which can maintain the memory efficiency for search process. Bjornsson et al. [7] compare the efficiency of the different explicit methods by experiments.

Explicit methods for generating abstraction hierarchies have a wide range of applications such as pathfinding in map and computer games. But as explicit methods have to explore all states on explicit graphs, they are not suitable for a large state space.

3.5 Conclusion

In this section we introduce hierarchical state space search and abstraction hierarchy, identify two characteristics of good abstraction hierarchies. Related works about how to generate good abstraction hierarchies are reviewed, and the limitations of these works are discussed. In the following chapters, we will research granular computing based new methods for generating good abstraction hierarchies and show the superiority of these new methods over existing methods.

Chapter 4

GRANULATED ATTRIBUTED GRAPH

This chapter introduces new concepts in granular computing – attributed graph, granulated attributed graph and granulated attributed graph hierarchy, which can be used to represent state spaces and abstraction hierarchies. These new concepts are fundamental to our granular computing based state space search methods.

4.1 Motivations

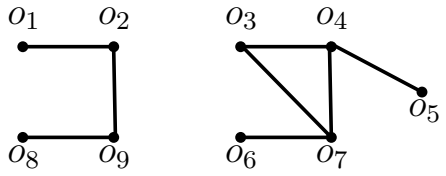
Information tables and DL-language are useful in discovering relations between granules and inferring knowledge from these relations, which makes granular computing a useful method in data mining [43, 60, 93, 98]. For example, in Table 2.1, we can find that the granule $m(Height, tall) = \{o_3, o_4, o_5, o_6, o_7\}$

is a coarse granule of the granule $m(Hair, red) = \{o_3, o_6\}$, and the formula $(Hair, red) \rightarrow (Height, tall)$ is satisfied by this information table, which suggests that if an object has value *red* on attribute $Hair$, it must have value *tall* on attribute $Height$.

From an information table we can only infer knowledge that is implied by inner properties of objects and granules and we are unable to make use of external relations between objects and granules [46]. In many practical cases, these external relations are also important for problem solving [110]. Figure 4.1 is an example. There are nine objects whose inner properties are expressed by an information table. Suppose each row in the table represents a person. The external relations can be derived from the fact that two persons may know each other or not. If two persons know each other, there is an external relation between them, otherwise there is no external relation between them. These external relations are expressed by an acquaintance graph in Figure 4.1b.

Object	*Height*	*Hair*	*Eyes*	*Weight*
o_1	*short*	*blond*	*brown*	*heavy*
o_2	*short*	*blond*	*brown*	*light*
o_3	*tall*	*red*	*blue*	*median*
o_4	*tall*	*dark*	*brown*	*light*
o_5	*tall*	*dark*	*brown*	*heavy*
o_6	*tall*	*red*	*blue*	*heavy*
o_7	*tall*	*dark*	*brown*	*median*
o_8	*short*	*blond*	*brown*	*median*
o_9	*short*	*blond*	*brown*	*median*

(a) Information table



(b) Acquaintance graph

Figure 4.1: An example of external relations

By analyzing the information table, we may find some relations between granules, such as all persons with *red Hair* must be *tall*. But if we want to know whether all *tall* persons know each other through common acquaintances or whether all *brown Hair* persons can know each other through common acquaintances, we cannot get an answer from the information table alone. The relations in terms of acquaintance are external relations that are beyond the information table.

In order to deal with such external relations, we introduce a new concept, called attributed graph, as a tool to represent a universe of a domain. An attributed graph combines the expressions of inner properties and external relations. An attributed graph has two parts, one is an information table and the other is a graph. These two parts give two perspectives of an attributed graph. We can see an attributed graph as a graph with an information table describing every vertex; we can also see an attributed graph as an information table with a graph expressing the external relations between objects.

4.2 Basic Definitions

We give the formal definition of an attributed graph as follows.

Definition 4.1. (Attributed graph) *An* attributed graph *for a universe of a domain is* $AG = (U, E, At, \{V_a | a \in At\}, \{I_a | a \in At\})$, *where*

- U *is a finite nonempty set of all the objects (or vertexes) in the universe of a domain,*

- *E is a relation on U, if $(v_1, v_2) \in E$, we say (v_1, v_2) is an edge,*
- *At is a finite nonempty set of attributes,*
- *V_a is a finite nonempty set of values for $a \in At$,*
- *$I_a : U \to V_a$ is an information function for $a \in At$.*

Each information function I_a is a total function that maps an object of U to exactly one value in V_a. The concrete content and values for the five elements U, E, At, $\{V_a|a \in At\}$, $\{I_a|a \in At\}$ depend on specific problems. An attributed graph can be divided into two parts: (U, E), and $(U, At, \{V_a|a \in At\}, \{I_a|a \in At\})$. By the definition, we know that (U, E) is in fact a graph and $(U, At, \{V_a|a \in At\}, \{I_a|a \in At\})$ is an information table. Therefore, we can see an attributed graph as a combination of an information table and a graph.

An attributed graph not only represents the relationships between objects in the universe U of a domain, but also represents conjunctively definable granulations of U. An attributed graph can be granulated by using conjunctively definable granulation.

Definition 4.2. (Granulated attributed graph) *Suppose $AG = (U, E, At, \{V_a|a \in At\}, \{I_a|a \in At\})$ is an attributed graph for a universe of a domain, G is a granulation on U defined by an attribute set Q, then $AG' = (U', E', At', \{V'_a|a \in At'\}, \{I'_a|a \in At'\})$ is an attributed graph for the granulation G, where*

- *U' is the set of all granules in G,*

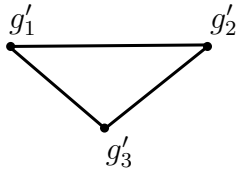
- *E' is a relation on U', $(g'_1, g'_2) \in E'$ if and only if $\exists v_1 \in g'_1 \exists v_2 \in g'_2((v_1, v_2) \in E)$, if $(g'_1, g'_2) \in E'$, we say (g'_1, g'_2) is an edge,*
- *$At' = Q$,*
- *$V'_a = V_a$ for $a \in At'$,*
- *$I'_a(g') = I_a(v)$ for $a \in At'$, where $v \in g'$.*

We call AG' a granulated attributed graph for a universe of a domain, or shortly, granulated attributed graph. *G is the* corresponding granulation *of AG', denoted by $G = granulation(AG')$*

An attributed graph and a granulated attributed graph can be expressed by using a graph along with an information table. Figure 4.1 gives an attributed graph for $U = \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9\}$. For the conjunctively definable granulation defined by $Weight$, the granulated attributed graph is shown in Figure 4.2. The granules are $g'_1 = \{o_1, o_5, o_6\}$, $g'_2 = \{o_2, o_4\}$, $g'_3 = \{o_3, o_7, o_8, o_9\}$. $o_1 \in g'_1$, $o_2 \in g'_2$ and (o_1, o_2) is an edge in the original attributed graph, so (g'_1, g'_2) is an edge in the granulated attributed graph. $o_6 \in g'_1$, $o_7 \in g'_3$ and (o_6, o_7) is an edge in the original attributed graph, so (g'_1, g'_3) is an edge in the granulated attributed graph. $o_2 \in g_2$, $o_9 \in g'_3$ and (o_2, o_9) is an edge in the original attributed graph, so (g'_2, g'_3) is an edge in the granulated attributed graph.

Granule	*Weight*
g'_1	*heavy*
g'_2	*light*
g'_3	*median*

(a) Information table for granulation



(b) Graph for granulation

Granule	*Objects*
g'_1	$\{o_1, o_5, o_6\}$
g'_2	$\{o_2, o_4\}$
g'_3	$\{o_3, o_7, o_8, o_9\}$

(c) Objects in granule

Figure 4.2: A granulated attributed graph AG'

As the granulated attributed graph AG' is defined by the granulation G and attribute set Q, and the granulation G is defined by the attribute set Q, the granulated attributed graph AG' is only defined by the attribute set Q. We use the denotation $AG' = granulated(Q)$ to express that AG' is the granulated attributed graph defined by the attribute set Q.

We can define a refinement-coarseness relation between granulated attributed graphs according to the refinement-coarseness relation between the corresponding granulations.

Definition 4.3. (Refinement-coarseness relation for granulated attributed graphs) *Suppose AG_1 and AG_2 are two granulated attributed graphs defined by attribute sets At_1 and At_2, respectively. G_1 and G_2 are the two corresponding*

granulations of AG_1 and AG_2, respectively. If $At_1 \supset At_2$ and $G_1 \ll G_2$, then AG_1 is a refined granulated attributed graph of AG_2 or AG_2 is a coarse granulated attributed graph of AG_1, denoted by $AG_1 \ll AG_2$.

Note that in Definition 4.3 there are two conditions $At_1 \supset At_2$ and $G_1 \ll G_2$. Sometimes for two granulated attributed graphs, only the second condition is satisfied. We name this relation weak refinement-coarseness.

Definition 4.4. (Weak refinement-coarseness relation for granulated attributed graphs) *Suppose AG_1 and AG_2 are two granulated attributed graphs defined by attribute sets At_1 and At_2, respectively. G_1 and G_2 are the two corresponding granulations of AG_1 and AG_2, respectively. If $G_1 \ll G_2$ and At_2 is not a subset of At_1, then AG_1 is a weak refined granulated attributed graph of AG_2 or AG_2 is a weak coarse granulated attributed graph of AG_1, denoted by $AG_1 \ll_{weak} AG_2$.*

As two different attribute sets may induce the same granulation, we define a kind of granulated attributed graphs that have the same granulation but different attribute sets.

Definition 4.5. (Homogeneous granulated attributed graphs) *Suppose AG_1 and AG_2 are two granulated attributed graphs defined by attribute sets At_1 are At_2, respectively. G_1 and G_2 are the two corresponding granulations of AG_1 and AG_2, respectively. If $At_1 \neq At_2$ and $G_1 = G_2$, then AG_1 and AG_2 are homogeneous granulated attributed graphs, denoted by $AG_1 \cong AG_2$.*

Based on these relations between granulated attributed graphs, we can create a hierarchical structure by a series of granulated attributed graphs, which will be discussed next.

4.3 Granulated Attributed Graph Hierarchy

According to the refinement-coarseness relation between granulated attributed graphs, we can have a kind of hierarchy for granulated attributed graphs.

Definition 4.6. (Granulated attributed graph hierarchy) *Suppose* $\{AG_0, AG_1, \cdots, AG_{n-2}, AG_{n-1}\}$ *is a sequence of granulated attributed graphs, where* $n > 1$. *If* $AG_i \ll AG_{i+1}$ *for* $0 \leq i < n-1$, *then this sequence is an* n-level granulated attributed graph hierarchy.

A granulated attributed graph hierarchy can be created by a sequence of ordered attribute sets, the constructive proof of the following theorem indicates a way to achieve this.

Theorem 4.1. *If* $At_{n-1} \subset At_{n-2} \subset \cdots \subset At_1 \subset At_0 \subseteq At$, $AG_i = granulated(At_i)$ *for* $0 \leq i \leq n-1$, *then* $AG_i \ll AG_{i+1}$ *or* $AG_i \cong AG_{i+1}$ *for* $0 \leq i < n-1$.

Proof: Let G_i be the corresponding granulation of AG_i for $0 \leq i \leq n-1$. According to Theorem 2.3, $G_i \ll G_{i+1}$ or $G_i = G_{i+1}$ for $0 \leq i < n-1$. According to Definition 4.3 and Definition 4.5, $AG_i \ll AG_{i+1}$ or $AG_i \cong AG_{i+1}$ for $0 \leq i < n-1$. ■

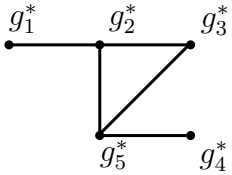
Suppose G and G' are corresponding granulations of two attributed graphs AG and AG', if G is a refined granulation of G', granules in G and granules in G' may have refinement-coarseness relations, which means that granules in G provide more details than granules in G'. Correspondingly, edges in AG may provide more details than edges in AG'. So there are also refinement-coarseness relations between edges in AG and edges in AG'.

Definition 4.7. (Refined edge) *Suppose* $AG = (U,\ E,\ At,\ \{V_a|a \in At\},\ \{I_a|a \in At\})$ *and* $AG' = (U',\ E',\ At',\ \{V'_a|a \in At'\},\ \{I'_a|a \in At'\})$ *are two granulated attributed graphs and* G *and* G' *are two corresponding granulations of* AG *and* AG'*, respectively. Assume* $G \ll G'$, $(g'_1, g'_2) \in E'$, $(g_1, g_2) \in E$. *If* $g_1 \preceq g'_1$ *and* $g_2 \preceq g'_2$, *then* (g_1, g_2) *is a refined edge of* (g'_1, g'_2).

An example of a refined edge is shown in Figure 4.2 and Figure 4.3. Figure 4.3 is a granulated attributed graph AG^* on the attributed graph in Figure 4.1. The granules are $g_1^* = \{o_1,\ o_5\}$, $g_2^* = \{o_2,\ o_4\}$, $g_3^* = \{o_3\}$, $g_4^* = \{o_6\}$, $g_5^* = \{o_7,\ o_8,\ o_9\}$. Figure 4.2 is another granulated attributed graph AG' on the same attributed graph in Figure 4.1. AG^* is a refined granulated attributed graph of AG'. Edge (g_1^*, g_2^*) is an refined edge of edge (g'_1, g'_2) because $g_1^* \preceq g'_1$ and $g_2^* \preceq g'_2$.

Granule	*Eyes*	*Weight*
g_1^*	*brown*	*heavy*
g_2^*	*brown*	*light*
g_3^*	*blue*	*median*
g_4^*	*blue*	*heavy*
g_5^*	*brown*	*median*

(a) Information table for granulation G^*



(b) Graph for granulation G^*

Granule	*Objects*
g_1^*	$\{o_1, o_5\}$
g_2^*	$\{o_2, o_4\}$
g_3^*	$\{o_3\}$
g_4^*	$\{o_6\}$
g_5^*	$\{o_7, o_8, o_9\}$

(c) Objects in granule

Figure 4.3: A granulated attributed graph AG^*

Theorem 4.2. *Suppose* $AG = (U, E, At, \{V_a|a \in At\}, \{I_a|a \in At\})$ *and* $AG' = (U', E', At', \{V'_a|a \in At'\}, \{I'_a|a \in At'\})$ *are two granulated attributed graphs. If* $AG \ll AG'$ *or* $AG \ll_{weak} AG'$*, then for every edge in* E' *there is a refined edge in* E*.*

Proof: Without losing generality, let $AG^0 = (U^0, E^0, At^0, \{V_a^0|a \in At^0\}, \{I_a^0|a \in At^0\})$ be the original attributed graph, and let (v'_1, v'_2) be an edge in E'. According to the definition of granulated attributed graph, there are two objects o_1, o_2 in U^0 such that $(o_1, o_2) \in E^0$ and $o_1 \in v'_1$, $o_2 \in v'_2$. Let v_1 and v_2 be granules

in AG that contains o_1 and o_2, respectively, then $(v_1, v_2) \in E$. According to the definition of refinement-coarseness relation for granulation, $v_1 \preceq v_1'$ and $v_2 \preceq v_2'$, so (v_1, v_2) is a refined edge for (v_1', v_2'). ■

Theorem 4.2 is important for proving the following Theorem 4.3, Theorem 4.4 and Theorem 5.1.

4.4 Inner Granule Graph and Some Theorems

Suppose there are two granulated attributed graphs $AG \ll AG'$, g' is a granule in AG', $g_1, g_2, \cdots, g_n (n > 1)$ are granules in AG and are all the refined granules of g'. We can extract $g_1, g_2, \cdots, g_n$ from AG and form a graph, the vertexes of this graph are $g_1, g_2, \cdots, g_n$. As all the vertexes of this graph are refined granules of g', we call this graph as an inner granule graph of g' on AG.

Definition 4.8. (Inner granule graph) *Suppose* $AG = (U, E, At, \{V_a | a \in At\}, \{I_a | a \in At\})$ *and* $AG' = (U', E', At', \{V_a' | a \in At'\}, \{I_a' | a \in At'\})$ *are two granulated attributed graphs. Assume* $g' \in U'$, $AG \ll AG'$ *or* $AG \ll_{weak} AG'$. *An inner granule graph of* g' *on* AG, *written* $g'|AG'/AG$,*is a graph* (U^0, E^0), *where* $U^0 = \{g \in U | g \preceq g'\}$, $E^0 = \{(g_1, g_2) | (g_1, g_2) \in E \wedge g_1 \preceq g' \wedge g_2 \preceq g'\}$.

Take Figure 4.2 and Figure 4.3 as an example. For the granule g_3', the inner granule graph $g_3'|AG'/AG^*$ is shown by Figure 4.4. There are two vertexes g_3^* and g_5^* in this graph because g_3^* and g_5^* are refined granules of g_3'. There is an edge between g_3^* and g_5^*.

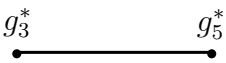


Figure 4.4: An inner granule graph of g_3' on AG^*

As a granulated attributed graph has graphical perspective, we can find some characteristics about the granulated attributed graph from the graphical view. The following theorem indicates such a characteristic.

Theorem 4.3. *Suppose* $AG = (U,\ E,\ At,\ \{V_a|a \in At\},\ \{I_a|a \in At\})$ *and* $AG' = (U',\ E',\ At',\ \{V_a'|a \in At'\},\ \{I_a'|a \in At'\})$ *are two granulated attributed graphs. Assume* $AG \ll AG'$ *or* $AG \ll_{weak} AG'$. *If the graph* (U', E') *is a connected graph and for every* $g' \in U'$, *the inner granule graph* $g'|AG'/AG$ *is a connected graph, then* (U, E) *is a connected graph.*

Proof: Without losing generality, let g_1 and g_2 be two granules in U, g_1' and g_2' be two granules in U', $g_1 \preceq g_1'$ and $g_2 \preceq g_2'$. If $g_1' = g_2'$, then g_1 and g_2 are both vertexes in graph $g_1'|AG'/AG$, as this graph is connected, so there is a path from g_1 to g_2. If $g_1' \neq g_2'$, there is a path from g_1' to g_2' in AG'. Let the path be $g_1'v_1'v_2'\cdots v_n'g_2'$. According to Theorem 4.2, there are refined edges for all edges on this path. As shown in Figure 4.5, let $(v_1,\ v_1^*)$ be a refined edge for (g_1', v_1'), (v_i, v_i^*) be a refined edge for $(v_{i-1}',\ v_i')$ for $1 < i \leq n$, and $(v_{n+1}, v*_{n+1})$ be a refined edge for (v_n', g_2'). Then $g_1 \preceq g'$, $v_1 \preceq g'$, and $v_i^* \preceq v_i'$ for $1 \leq i \leq n$, $v_i \preceq v_{i-1}'$ for $1 < i \leq n+1$, $v_{n+1}^* \preceq g_2'$, $g_2 \preceq g_2'$. As every inner granule graph is connected, there are paths from g_1 to v_1, from v_1^* to v_2, from v_2^* to v_3, $\cdots$, from v_{n-1}^* to v_n, from v_n^* to v_{n+1}, from v_{n+1}^* to g_2. So there is a path from g_1 to g_2. ■

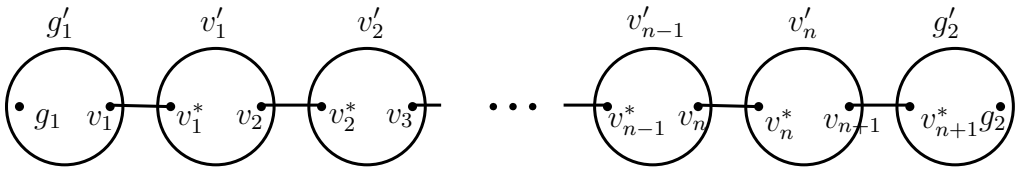


Figure 4.5: Connectivity theorem proof

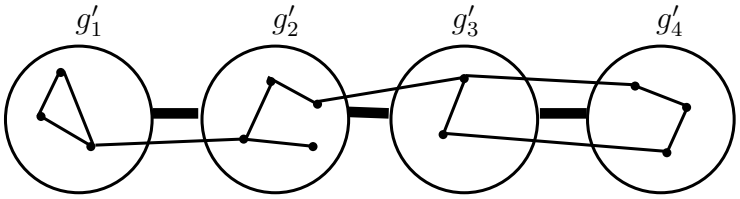


Figure 4.6: Connectivity theorem explanation

Figure 4.6 provides an explanation of Theorem 4.3. There are two granulated attributed graphs AG' and AG, $AG \ll AG'$ or $AG \ll_{weak} AG'$. There are four granules g'_1, g'_2, g'_3, g'_4 in AG'. The thin lines represent edges between granules in AG, and the thick lines represent edges between granules in AG'. We can see granules in AG as vertexes grouped by granules in AG'. There are four groups of vertexes. For every thick line between two groups, there is at least one thin line between vertexes that belong to these two groups respectively. All vertexes in one group can be connected by edges within this group. We can see that all the vertexes in all the four groups can be connected by thin lines.

Theorem 4.4. *Suppose* $AG = (U, E, At, \{V_a|a \in At\}, \{I_a|a \in At\})$ *and* $AG' = (U', E', At', \{V'_a|a \in At'\}, \{I'_a|a \in At'\})$ *are two granulated attributed graphs,* $AG \ll AG'$ *or* $AG \ll_{weak} AG'$, $g'_{start}g'_1g'_2 \cdots g'_ng'_{goal}$ *is a path in* (U', E'), $g_{start} \in g'_{start}$, $g_{goal} \in g'_{goal}$. *If for every* $g' \in U'$, *the inner granule graph* $g'|AG'/AG$ *is a*

connected graph, then there is a path from g_{start} to g_{goal} in (U, E).

Proof: According to Theorem 4.2 there are refined edges for all edges (g'_{start}, g'_1), (g'_1, g'_2), $\cdots$, (g'_n, g'_{goal}). As every inner granule graph is connected, by the same reasoning in the proof of Theorem 4.3, there is a path from g_{start} to g_{goal}.

■

The above theorems pave a path for using granulated attributed graphs to generate good abstraction hierarchies, in the next chapter we will see how to achieve this goal.

4.5 Conclusion

In this chapter, we propose a new concept of an attributed graph. An attributed graph can describe both inner properties and external relations among granules and granulations. Based on an attributed graph, the concept of granulated attributed graph is proposed and relations between granulated attributed graphs are discovered. According to these relations, we further propose a hierarchy named granulated attributed graph hierarchy. Finally, we give the concept of inner granule graph and prove some theorems about the inner granule graph. In the next chapter, we will introduce how to use attributed graphs and granulated attributed graphs to generate good abstraction hierarchies.

Chapter 5

GRANULAR STATE SPACE SEARCH

Based on attributed graphs and granulated attributed graphs, this chapter presents two new methods for constructing good abstraction hierarchies. We first explain how to represent a state space by an attributed graph and how to represent an abstraction by a granulated attributed graph and then give the two new methods by using these representations. Finally, we suggest a new refinement procedure named *PSRB Refinement* that can further speed up the refinement procedure.

5.1 Attributed Graph and State Space

An attributed graph gives us a new way to generate abstraction hierarchies by the fact that a state space can be expressed by an attributed graph. A granulated attributed graph hierarchy can be created from an attributed graph, and the

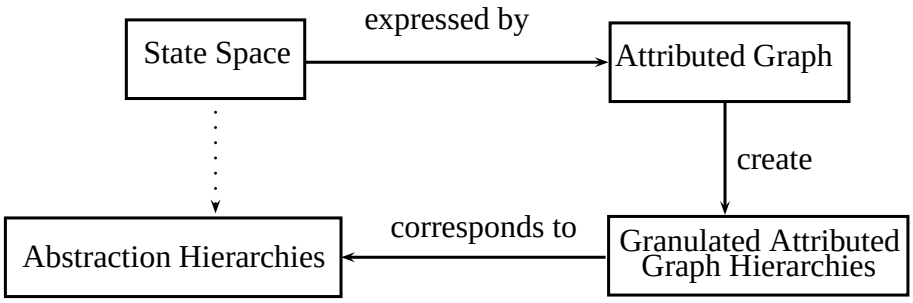


Figure 5.1: Creating abstraction hierarchies based on an attributed graph

granulated attributed graph hierarchy corresponds to an abstraction hierarchy of the state space expressed by the attributed graph. Figure 5.1 shows the three steps in creating an abstraction hierarchy through an attributed graph.

In the previous chapter, we showed how to create a granulated attributed graph hierarchy from an attributed graph. In this section, we deal with the first and last steps by explaining how to express a state space by an attributed graph, and how to transform a granulated attributed graph hierarchy into an abstraction hierarchy.

5.1.1 Attributed Graph Representation of a State Space

As a state space can be represented by a graph, it can be further represented by an attributed graph by introducing attributes and values to describe states and operators. These attributes and values are created according to the basic elements of a problem. A state is represented as a set of attribute-value pairs and an operator is represented as a tuple $(Precondition, Add, Del)$, where $Precondition$, Add and Del are all sets of attribute-value pairs. The representation of an oper-

ator is similar to that in Knoblock's method [35] in Section 3.4, the difference is that the *Precondition*, *Add*, and *Del* are now sets of attribute-value pairs instead of literals. As a state is represented by a set of attribute-value pairs, a state space is jointly represented by an information table and a graph; an information table describes states and a graph describes relations between states. The information table and the graph form an attributed graph, thus, a state space is represented by an attributed graph.

Consider the example of the three-disk Hanoi tower problem. We can create three attributes A, B, C to describe the three disks and create three values $1, 2, 3$ as their values. The three attributes and three values can form an information table and every state can be expressed uniquely by a set of attribute-value pairs. For example, $\{(A = 1), (B = 1), (C = 1)\}$ expresses the state that *disk A*, *disk B* and *disk C* are on *peg* 1. The information table is shown in Table 5.1. The information table along with the graph in Figure 3.2 form the attributed graph of the state space of the three-disk Hanoi tower problem. The operators represented by attribute-value pairs are shown in Table 5.2, these operators are not part of the attributed graph, but they will be used when transforming a granulated attributed graph hierarchy into an abstraction hierarchy.

5.1.2 Generating Abstraction Hierarchies based on Granulated Attributed Graph Hierarchies

After representing a state space by an attributed graph, we can derive a granulated attributed graph hierarchy, and generate an abstraction hierarchy. For

State	C	B	A
s_1	1	1	1
s_2	1	1	2
s_3	1	1	3
.....			
s_{25}	3	3	1
s_{26}	3	3	2
s_{27}	3	3	3

Table 5.1: The information table of the three-disk Hanoi tower problem

every granulated attributed graph in the hierarchy, assuming its attribute set is A', we take every granule in this granulated attributed graph as an abstract state. For every operator in the original state space, we delete all attribute-value pairs whose attributes do not belong to the attribute set A', which results in abstract operators. All the abstract states and abstract operators form an abstraction; the graph of the granulated attributed graph is the graph of the abstraction. As every granulated attributed graph can create an abstraction, all the abstractions created by all the granulated attributed graphs in the granulated attributed graph hierarchy form an abstraction hierarchy.

The algorithm of generating abstraction hierarchies based on granulated attributed graph hierarchies is given by Algorithm 5.1

No.	Precondition	Add	Del
1	$\{(A = 1)\}$	$\{(A = 2)\}$	$\{(A = 1)\}$
2	$\{(A = 2)\}$	$\{(A = 1)\}$	$\{(A = 2)\}$
3	$\{(A = 1)\}$	$\{(A = 3)\}$	$\{(A = 1)\}$
4	$\{(A = 3)\}$	$\{(A = 1)\}$	$\{(A = 3)\}$
5	$\{(A = 2)\}$	$\{(A = 3)\}$	$\{(A = 2)\}$
6	$\{(A = 3)\}$	$\{(A = 2)\}$	$\{(A = 3)\}$
7	$\{(B = 1), (A = 3)\}$	$\{(B = 2)\}$	$\{(B = 1)\}$
8	$\{(B = 2), (A = 3)\}$	$\{(B = 1)\}$	$\{(B = 2)\}$
9	$\{(B = 1), (A = 2)\}$	$\{(B = 3)\}$	$\{(B = 1)\}$
10	$\{(B = 3), (A = 2)\}$	$\{(B = 1)\}$	$\{(B = 3)\}$
11	$\{(B = 2), (A = 1)\}$	$\{(B = 3)\}$	$\{(B = 2)\}$
12	$\{(B = 3), (A = 1)\}$	$\{(B = 2)\}$	$\{(B = 3)\}$
13	$\{(C = 1), (A = 3), (B = 3)\}$	$\{(C = 2)$	$\{(C = 1)\}$
14	$\{(C = 2), (A = 3), (B = 3)\}$	$\{(C = 1)\}$	$\{(C = 2)\}$
15	$\{(C = 1), (A = 2), (B = 2)\}$	$\{(C = 3)\}$	$\{(C = 1)\}$
16	$\{(C = 3), (A = 2), (B = 2)\}$	$\{(C = 1)\}$	$\{(C = 3)\}$
17	$\{(C = 2), (A = 1), (B = 1)\}$	$\{(C = 3)\}$	$\{(C = 2)\}$
18	$\{(C = 3), (A = 1), (B = 1)\}$	$\{(C = 2)\}$	$\{(C = 3)\}$

Table 5.2: The attribute-value pair representations of operators

Algorithm 5.1 Generating abstraction hierarchies based on granulated attributed graph hierarchies

1: **function** $GenerateAbstractionHierarchy(GranulatedAttributedGraphHierarchy)$
2: $AbstractionHierarchy \leftarrow \emptyset$
3: $level \leftarrow$ highest level of $GranulatedAttributedGraphHierarchy$
4: $i \leftarrow 1$
5: **while** $i <= level$ **do**
6: $A \leftarrow$ the attribute set of the ith level granulated attributed graph in $GranulatedAttributedGraphHierarchy$
7: $SP \leftarrow$ all granules in $GranulatedAttributedGraphHierarchy$
8: $OP \leftarrow \emptyset$
9: **for all** $p \in$ all states in the original state space **do**
10: $OP = OP+$ delete A from p
11: $AbstractionHierarchy \leftarrow (SP, OP)$
12: **end for**
13: $i++$
14: **end while**
15: return $AbstractionHierarchy$
16: **end function**

Let us take the three-disk Hanoi tower problem as an example to illustrate the process of generating abstraction hierarchies based on granulated attributed graph hierarchies. When we have an attribute set $\{C\}$, we can get a granulated attributed graph whose information table is shown in Table 5.3 and whose graph

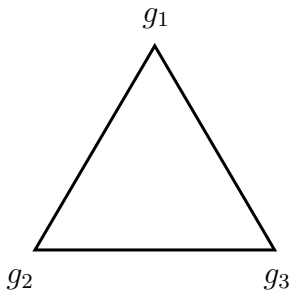


Figure 5.2: The graph of the granulated attributed graph defined by $\{C\}$

is shown in Figure 5.2, and the abstract operators are shown in Table 5.4.

granule	C
g_1	1
g_2	2
g_3	3

Table 5.3: The information table of the granulated attributed graph defined by $\{C\}$

We have shown how to generate an abstraction hierarchy by an attributed graph. The next question is how to generate a good abstraction hierarchy by an attributed graph. We will answer this question in the following section.

No.	Precondition	Add	Del
1	$\{(C = 1)\}$	$\{(C = 2)\}$	$\{(C = 1)\}$
2	$\{(C = 2)\}$	$\{(C = 1)\}$	$\{(C = 2)\}$
3	$\{(C = 1)\}$	$\{(C = 3)\}$	$\{(C = 1)\}$
4	$\{(C = 3)\}$	$\{(C = 1)\}$	$\{(C = 3)\}$
5	$\{(C = 2)\}$	$\{(C = 3)\}$	$\{(C = 2)\}$
6	$\{(C = 3)\}$	$\{(C = 2)\}$	$\{(C = 3)\}$

Table 5.4: The operators of the granulated attributed graph defined by $\{C\}$

5.2 Generating Good Granulated Attributed Graph Hierarchies

As we can generate abstraction hierarchies by creating granulated attributed graph hierarchies, we can convert the problem of generating good abstraction hierarchies into the problem of generating good granulated attributed graph hierarchies. In this section, we first discuss what are good granulated attributed graph hierarchies and then give methods to create good granulated attributed graph hierarchies.

5.2.1 Good Granulated Attributed Graph Hierarchies

A good abstraction hierarchy should satisfy two criteria: high completeness degree and few backtracking occurrences. We give a property of granulated attributed

graph hierarchies, which is sufficient to generate good abstraction hierarchies that satisfy the two criteria. This property considers the inner granule graph (see Definition 4.8) of every granule, thus we call it *inner connectivity*.

Definition 5.1. (Inner connectivity) *Suppose* $AG_0 \ll AG_1 \ll \cdots AG_{n-1} \ll AG_n$ *is a granulated attributed graph hierarchy. If for any* AG_i *and* AG_{i+1}, $0 \leq i < n$, *for every granule* $g \in AG_{i+1}$, *the inner granule graph* $g|AG_{i+1}/AG_i$ *is connected, then this granulated attributed graph hierarchy has the inner connectivity property.*

Theorem 5.1. *If a granulated attributed graph hierarchy of a state space has the inner connectivity property, then the corresponding abstraction hierarchy of the state space generated by this granulated attributed graph hierarchy is a complete hierarchy and can avoid any backtracking.*

Proof: We first prove that the abstraction hierarchy is complete. Without losing generality, let P be a solvable problem. The solution of P in the original state space can be mapped into a path $s_1^n s_2^n \cdots s_{m-1}^n s_m^n$ in AG_n, where s_1^n is the granule containing the start state and s_m^n is the granule containing the goal state. This path may contain a cycle. Now we compact this path by the following step: scan the states in the path one by one from s_1^n to s_m^n, if the current scanned state s_p^n is the same as an already scanned one s_j^n, then delete all the states from s_j^n to s_{p-1}^n inclusively. After this compact process, the new path $s_{m_1}^n s_{m_2}^n \cdots s_{m_{k-1}}^n s_{m_k}^n$ is a non-cyclic path and $s_{m_1}^n$ is the same as s_1^n and $s_{m_k}^n$ is the same as s_m^n, so this non-cyclic path is a solution in AG_n. For the first edge $(s_{m_1}^n,\ s_{m_2}^n)$, according

to Theorem 4.2, there is an edge (s_i^{n-1}, s_j^{n-1}), where s_i^{n-1} and s_j^{n-1} are granules in AG_{n-1} and $s_i^{n-1} \preceq s_{m_1}^n$, $s_j^{n-1} \preceq s_{m_2}^n$. As the inner graph $s_{m_1}^n | AG_n/AG_{n-1}$ is connected, there is a non-cyclic path from s_1^{n-1} to s_i^{n-1} and all granules on the path are in $s_{m_1}^n$, where s_1^{n-1} is the granule containing the start state. For the second edge $(s_{m_2}^n, s_{m_3}^n)$, according to Theorem 4.2, there is an edge (s_o^{n-1}, s_p^{n-1}), where s_o^{n-1} and s_p^{n-1} are granules in AG_{n-1} and $s_o^{n-1} \preceq s_{m_2}^n$, $s_p^{n-1} \preceq s_{m_3}^n$. As the inner graph $s_{m_2}^n | AG_n/AG_{n-1}$ is connected, there is a non-cyclic path from s_j^{n-1} to s_o^{n-1} and all granules on path are in $s_{m_2}^n$. The s_1^{n-1} and s_o^{n-1} is connected by a non-cyclic path. By the same reasoning, we can connect s_1^{n-1} to s_e^{n-1} by a non-cyclic path, where s_e^{n-1} is the granule containing the goal state. We know that a path is found in AG_{n-1} by the ordered refinement procedure. By the same reasoning, the solution in the original state space can be found by the ordered refinement procedure. So this abstraction hierarchy is complete.

As every inner graph is connected, it means that every sub-problem can be solved, so there is no need to backtrack. Thus, this abstraction hierarchy can avoid any backtracking. ■

To check whether a granulated attributed graph hierarchy has the inner connectivity property, we need to investigate every two consecutive layers of a granulated attributed graph hierarchy, and for every granule of every granulated attributed graph (except the original attributed graph) in this granulated attributed graph hierarchy, we must create an inner graph on the consecutive lower layer. The following theorem indicates an easier way to check whether a granulated attributed graph hierarchy has the inner connectivity property, which only re-

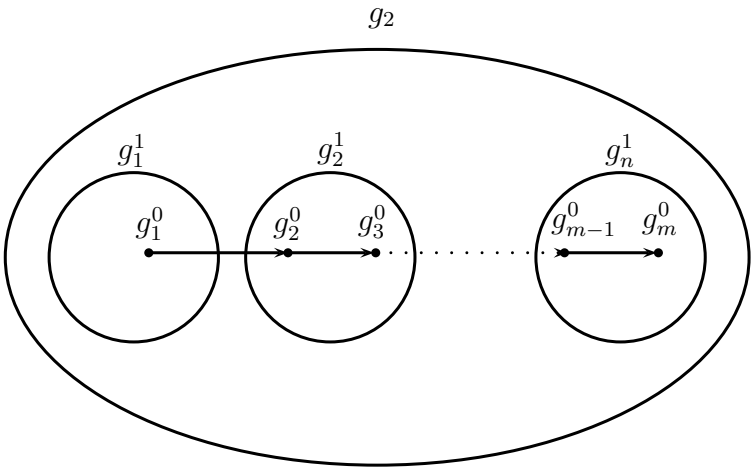


Figure 5.3: Relations between paths and granules

quires creating inner graphs on the original attributed graph for every granulated attributed graph.

Theorem 5.2. *Given three granulated attributed graphs $AG_0 \ll AG_1 \ll AG_2$, if for every granule $g_1 \in AG_1$, $g_1|AG_1/AG_0$ is connected and for every granule $g_2 \in AG_2$, $g_2|AG_2/AG_0$ is connected, then for every granule $g_2 \in AG_2$, $g_2|AG_2/AG_1$ is connected.*

Proof: Let g_1^1 and g_n^1 be any two granules in AG_1 that are refined granules of g_2, and let g_1^0 and g_m^0 be two granules in AG_0, $g_1^0 \preceq g_1^1$, $g_m^0 \preceq g_n^1$. Because $g_2|AG_2/AG_0$ is connected, there is a path from g_1^0 to g_m^0 and all granules on the path are in g_2. Let this path be $g_1^0 g_2^0 g_3^0 \cdots g_{m-1}^0 g_m^0$. We choose and connect all the granules in AG_1 that contain the granules on the path to get a new path in AG_1, which is $g_1^1 g_2^1 \cdots g_n^1$. All the granules on this new path are in g_2. As g_1^1 and g_n^1 are any two refined granules of g_2, $g_2|AG_2/AG_1$ is connected. Figure 5.3 shows the relations between these paths and granules. ■

Theorem 5.2 gives us an easier way to check whether a granulated attributed graph hierarchy has the inner connectivity property. That is, we do not need to investigate every two consecutive layers in a granulated attributed graph hierarchy, we can only investigate every granulated attributed graph and the original attributed graph. Take Figure 5.3 as an example, let us assume AG_0 is the original attributed graph. If we consider every two consecutive layers of the granulated attributed graph hierarchy such as AG_2 and AG_1, we should first create a graph (G^1, E^1), where G^1 is the set of all refined granules of g_2 in AG^1, then we investigate whether this graph is connected. If we consider $g_2|AG_2/AG_0$, we do not need to create any graph, we can directly use the graph in AG_0 and consider the part that is in g_2 and check whether it is connected.

5.2.2 The Exhaustive Method

According to Theorem 5.2, we can easily find a method to create good granulated attributed graph hierarchies. Algorithm 5.2 is a straightforward algorithm to find a sequence of attribute sets that can define a good granulated attributed graph hierarchy. This algorithm first finds all candidate subsets (from step 4 to step 9) and then finds an ordered sequence from these candidate subsets. This obtained ordered sequence can induce an abstraction hierarchy that has the inner connectivity property.

Algorithm 5.2 Exhaustive algorithm

1: AG_0 is the original attributed graph.
2: $R \leftarrow \emptyset$
3: $A \leftarrow$ all non-empty subset of At
4: **for all** $a \in A$ **do**
5: generate an granulated attributed graph AG defined by a
6: **if** every granule $g \in AG$, $g|AG/AG_0$ is connected **then**
7: $R \leftarrow R + g$
8: **end if**
9: **end for**
10: find an order sequence S in R
11: return S

A drawback of this algorithm is that it is not efficient. This algorithm will examine $2^{|At|}$ granulated attributed graphs, where $|At|$ is the number of attributes. In every granulated attributed graph, it will try to examine paths between every two states. Let n be the number of all vertexes in the original state space, d be the average number of vertexes in one granule, m be the average number of vertexes explored for finding a path between two vertexes. As for every granule there are $\frac{d(d-1)}{2}$ paths that should be found, and there are $\frac{n}{d}$ granules in one granulated attributed graph, the total vertexes explored in this algorithm are $2^{|At|}m\frac{n}{d}\frac{d(d-1)}{2}$, which are equal to $2^{|At|}mn\frac{(d-1)}{2}$. The cost of this algorithm is exponential to the number of attributes and proportional to the number of all vertexes in the original attributed graph. When the original state space is large or when the attributes

are many, this algorithm becomes impracticable. We need another algorithm to find good abstraction hierarchies efficiently.

5.2.3 The Learning Based Method

In the exhaustive method, we exhaustively examine all granulated attributed graphs one by one, which is not efficient. We now discuss a new method based on the machine learning principle that can learn good granulated attributed graph hierarchies quickly from sample paths.

Principle

The principle of machine learning is to learn the best hypothesis from samples [1, 83], which is demonstrated in Figure 5.4. There is an instance space which contains all the instances, every instance x has a value $c(x)$. There is a hypothesis space which contains all the hypotheses. For a hypothesis h and an instance x, this hypothesis determines a value $h(x)$ for this instance. If $h(x) = c(x)$, this hypothesis gives a correct value to x, otherwise it gives an incorrect value to x. There is a sample set which is a small subset of instance space, the value of every instance in the sample set is known. The purpose of machine learning is to get a hypothesis h' with the help of the sample set, so that h' gives correct values to as many of the instances in the instance space as possible.

There are many methods in machine learning to learn the best hypothesis [84], one class of methods is to learn the hypothesis gradually such as the ID3 classification method [61].

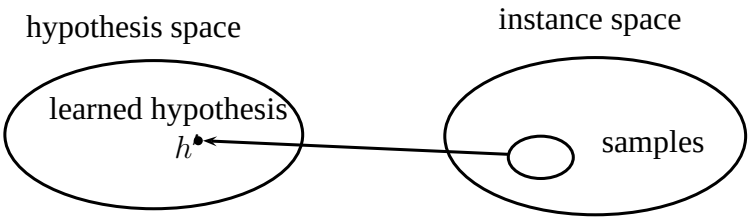


Figure 5.4: Machine learning principle

We adapt the machine learning principle to learn a good granulated attributed graph hierarchy from a sample set, which is demonstrated in Figure 5.5. There is a set of all solvable problems in a state space, a set of all granulated attributed graph hierarchies, and a sample set which is a small subset of the set of all solvable problems and the solutions for all problems in the sample set have already been found. The sample set is created by randomly selecting a group of problems and finding solutions for these problems by breadth-first search method; if a solution can be found for a problem, this problem is added into the sample set. The purpose of our learning based method is to get a granulated attributed graph hierarchy G' with the help of the sample set, so that G' can solve as many more problems in the solvable problem set as possible. The granulated attributed graph hierarchy generated by the learning based method may not completely satisfy the inner connectivity property, but it can satisfy this property as much as possible.

In this learning based method, we put a restriction on the attribute subsets. That is, the highest level in the granulated attributed graph hierarchy has an attribute subset of only one attribute, and every level, except the highest and lowest level, has an attribute subset of one more attribute than the immediate higher level.

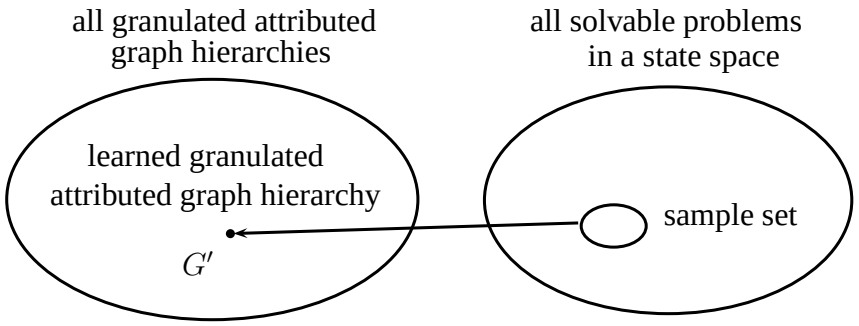


Figure 5.5: Machine learning principle for learning good granulated attributed graph hierarchies

The basic steps of our method are that we first use sample paths to examine all the attributes and choose the most appropriate one as the highest level attribute set. Then we use sample paths to examine the remaining attributes and choose the most appropriate one, which will be combined with the highest level attribute set to form the second highest level attribute set. Repeat this process until the attribute set of the second lowest level is found.

Learning Criteria

One basic issue in our learning based method is to judge which attribute is the most appropriate. To deal with this question, we introduce two concepts named *redundant vertex* and *conflict value*. The definitions of redundant vertex and conflict value are given as follows.

Definition 5.2. (Redundant vertex and conflict value) *Let $v_1v_2\cdots v_{n-1}v_n$ be a path in an attributed graph AG and AG' be a granulated attributed graph of AG.*

1. *The first vertex v_1 is not a redundant vertex.*

2. *For a vertex v_k on the path, where $k > 1$, if there are two vertexes v_i and v_j on the path such that $i < k < j$, v_i and v_j are in the same granule of AG', v_k and v_i are in different granules of AG', and v_i is not a redundant vertex, then v_k is a* redundant vertex *for AG'. Otherwise v_k is not a redundant vertex for AG'.*

The total number of redundant vertexes in the path is the conflict value *of this path for AG'.*

Figure 5.6 is an example of redundant vertexes. The path in AG is $s_1s_2s_3s_4s_5s_6$ $s_7s_8s_9s_{10}s_{11}$. The corresponding path in AG' is $s'_1s'_2s'_1s'_2s'_3s'_4s'_5s'_3s'_6$. We know that s_2 is a redundant vertex for AG', because the previous vertex s_1 is not a redundant vertex, s_1 and the afterward vertex s_3 are in the same granule s'_1, while s_2 is in a different granule s'_2. For s_3, although s_2 and s_4 are in the same granule while s_3 is in a different granule, as s_2 is a redundant vertex, s_3 is not a redundant vertex. For the same reason, s_7, s_8, s_9 are all redundant vertexes for AG'. As there are 4 redundant vertexes, the conflict value of this path for AG' is 4.

The existence of redundant vertexes means that the corresponding path in the higher level granulated attributed graph is not non-cyclic. For example, in Figure 5.6, the redundant vertex s_2 means that there is a cycle $s'_1s'_2s'_1$ in the corresponding path in the higher level granulated attributed graph. Redundant vertexes may affect the completeness degree. In order to have a high completeness degree, the number of redundant vertexes, or the conflict value, should be as small

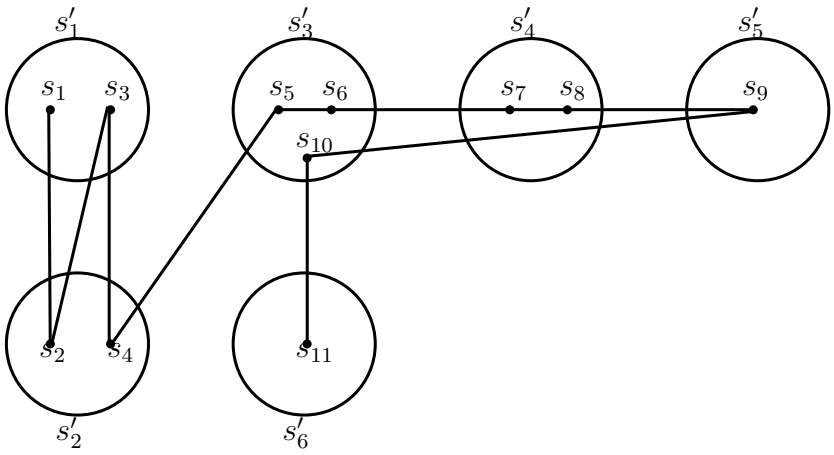


Figure 5.6: An example of redundant vertexes

as possible. Therefore, we can judge whether an attribute set is appropriate for a sample path in the term of the conflict value, the less the conflict value, the more appropriate of this attribute set.

Learning Based Algorithm

Now we can present the algorithm for generating good granulated attributed graph hierarchies. Our algorithm contains three major functions.

The first function is to calculate the conflict value of a given path for a granulated attributed graph created by a given attribute set. The pseudo-code of this function is given by Algorithm 5.3. The inputs of this function are the given path and attribute set. Because the first vertex and the last vertex cannot be redundant vertexes, the function iterates vertexes in the path from the second vertex to the second last vertex (from step 5 to step 26). For every iterated vertex (the ith vertex), the function considers every previous vertex of the ith vertex and every post vertex of the ith vertex (from step 7 to step 22). If the previous

vertex is not a redundant vertex, and the previous vertex and the ith vertex are not in the same granule (step 11), and the post vertex and the previous vertex are in the same granule (step 16), then the ith vertex is a redundant vertex (step 17). If no such previous vertex and post vertex are found, the ith vertex is a non redundant vertex (step 24). The function returns the total number of redundant vertexes in this path.

Algorithm 5.3 Calculating conflict values

```
1: function CalculateConflictValue(Path, AttributeSet)
2:     create a granulated attributed graph AG from AttributeSet
3:     mark the first vertex as non-redundant
4:     i ← 2
5:     while i is less than the length of Path do
6:         exitloop ← false
7:         for all pre ∈ (1..i − 1) do
8:             if exitloop = true then
9:                 exit
10:            end if
11:            if the preth vertex is non-redundant and the preth vertex and the
       ith vertex are not in the same granule of AG then
12:                for all post ∈ (i + 1..length of Path) do
13:                    if exitloop = true then
14:                        exit
15:                    end if
16:                    if the postth vertex and the preth vertex are in the same
       granule of AG then
17:                        mark the ith vertex as redundant
18:                        exitloop ← true
19:                    end if
20:                end for
```

Algorithm 5.3 Calculating conflict values (continued I)

```
21:             end if
22:         end for
23:         if exitloop = false then
24:             mark the ith vertex as non-redundant
25:         end if
26:     end while
27:     return the number of marked redundant vertexes
28: end function
```

The second function is to generate the most appropriate attribute set for an abstraction. The pseudo-code of this function is given by Algorithm 5.4. The inputs of this function are the whole attribute set, the already generated immediate higher level attribute set and the sample paths. Because we only choose one appropriate attribute and combine it with the already generated immediate higher level attribute set, all the candidate attributes from which we can choose are the whole attribute set minus the already generated immediate higher level attribute set (step 2). The function iterates all candidate attributes (from step 5 to 15). For every iterated attribute, the function combines it with the already generated immediate higher level attribute set to form a new attribute set *NewAttributeSet* (step 6) and gets the total conflict value of all the sample paths for the granulated attributed graph created by *NewAttributeSet* (from step 7 to 10). Then the function identifies the smallest conflict value and choose the most appropriate attribute set (from step 11 to 14), this attribute set is returned by the function

and can be used to generate an abstraction.

Algorithm 5.4 Generating attribute set

```
1: function GenerateOne(WholeAttributesSet, HigherLevelSet, SamplePaths)
2:     CandidateSet ← WholeAttributesSet − HigherLevelSet
3:     BestSet ← ∅
4:     BestConflictValue ← MaxInt
5:     for all a ∈ CandidateSet do
6:         NewAttributeSet ← a + HigherevelSet
7:         NewConflictValue ← 0
8:         for all p ∈ SamplePaths do
9:             NewConflictValue = NewConflictValue + CalculateConflictValue(p, NewAttributeSet)
10:        end for
11:        if NewConflictValue < BestConflictValue then
12:            BestConflictValue = NewConflictValue
13:            BestSet = NewAttributeSet
14:        end if
15:    end for
16:    return BestSet
17: end function
```

The third function is to generate an attribute set sequence for an abstraction hierarchy. The pseudo-code of this function is given by Algorithm 5.5. The inputs

of this function are the attribute set, the sample paths and the required number of levels of this abstraction hierarchy. The function generates all abstractions from the top level to the first level (from step 4 to 11). For every level, the function generates the most appropriate attribute set for the abstraction (step 7) and adds this attribute set into the sequence (step 8). Finally, the function returns the sequence of the attribute sets.

Algorithm 5.5 Generating attribute set sequence

```
1: function GenerateSequence(AttributesSet, SamplePaths, level)
2:     Sequence ← ∅
3:     HighSet ← ∅
4:     i ← level − 1
5:     while i > 0 do
6:         BestSet ← ∅
7:         BestSet=GenerateOne(AttributesSet, HighSet, SamplePaths)
8:         Sequence = Sequence + BestSet
9:         HighSet = BestSet
10:        i − −
11:    end while
12:    return Sequence
13: end function
```

Our algorithm can be illustrated by the three-disk Hanoi tower problem. Assume the sample paths are $\{path_1 = s_1s_3s_6s_5s_{23}s_{22}s_{25}s_{27}, path_2 = s_1s_3s_6s_5s_{23}s_{24}s_{21}$ s_{19}, $path_3 = s_1s_2s_8s_7\}$. First we try to find the top level attribute set. We cal-

culate the conflict values of these three sample paths for attribute set $\{C\}$. The graph of the granulated attributed graph defined by $\{C\}$ is shown by Figure 5.7. The information table is shown by Table 5.5. The three paths in the granulated attributed graph are shown by Figure 5.8. There is no redundant vertex for $path_1$ on this granulated attributed graph, so the conflict value of $path_1$ for this granulated attributed graph is 0. By the some reason, the conflict values of $path_2$ and $path_3$ for this granulated attributed graph are also 0. The total conflict value of sample paths for this granulated attributed graph is 0.

We then calculate the conflict values of these three sample paths for attribute set $\{B\}$. The graph of the granulated attributed graph defined by $\{B\}$ is shown by Figure 5.9. The information table is shown by Table 5.6. The three paths in the granulated attributed graph are shown by Figure 5.10. For $path_1$, there is no redundant vertex, the conflict value of $path_1$ for this granulated attributed graph is 0. For $path_2$, s_3 and s_{21} are in the same granule g_1, s_6 is in granule g_2 which is different from g_1, and s_6 is between s_3 and s_{21}, thus, s_6 is a redundant vertex. By the same reason, s_5, s_{23}, s_{24}, are redundant vertexes. The conflict value of $path_2$ for this granulated attributed graph is 4. For $path_3$, there is no redundant vertex. The total conflict value of sample paths for this granulated attributed graph is 4.

Finally, we calculate the conflict values of these three sample paths for attribute set $\{A\}$. The graph of the granulated attributed graph defined by $\{A\}$ is shown by Figure 5.11. The information table is shown by Table 5.7. The three paths in the granulated attributed graph are shown by Figure 5.12. For $path_1$, s_3

is a redundant vertex, because s_3 is between s_1 and s_{22}, s_3 is in granule g_3, s_1 and s_{22} are in a different granule g_1. For the same reason, s_6, s_5, s_{23} are redundant vertexes. The conflict value of $path_1$ for this granulated attributed graph is 4. For $path_2$, s_3,s_6, s_5,s_{23},s_{24},s_{21} are redundant vertexes. The conflict value of $path_2$ for this granulated attributed graph is 6. For $path_3$, s_2 and s_8 are redundant vertexes, the conflict value of $path_3$ is 2. The total conflict value of sample paths for this granulated attributed graph is 12.

By comparing the total conflict values of sample paths, we know that the attribute C should be chosen for the top level granulated attributed graph.

We move on to the second highest level. The remaining attributes are A and B. The possible attribute sets for this level are $\{A, C\}$ and $\{B, C\}$. The graph of the granulated attributed graph defined by $\{B, C\}$ is shown by Figure 5.13, The information table is shown by Table 5.8. The three paths in the granulated attributed graph are shown by Figure 5.14. For $path_1$, $path_2$ and $path_3$, there is no redundant vertex. The total conflict value of sample paths for this granulated attributed graph is 0.

The graph of the granulated attributed graph defined by $\{A, C\}$ is shown by Figure 5.15. The information table is shown by Table 5.9. The three paths in the granulated attributed graph are shown by Figure 5.16. For $path_1$ and $path_2$, there is no redundant vertex. For $path_3$, s_2 and s_8 are redundant vertexes. The total conflict value of sample paths for this granulated attributed graph is 2.

By comparing the total conflict values of sample paths, we know that the attribute set $\{B, C\}$ should be chosen for the second highest level granulated

attributed graph. We find the ordered sequence of attribute sets, which is $\{C\}$, $\{B, C\}$. We can create a granulated attributed graph hierarchy accordingly.

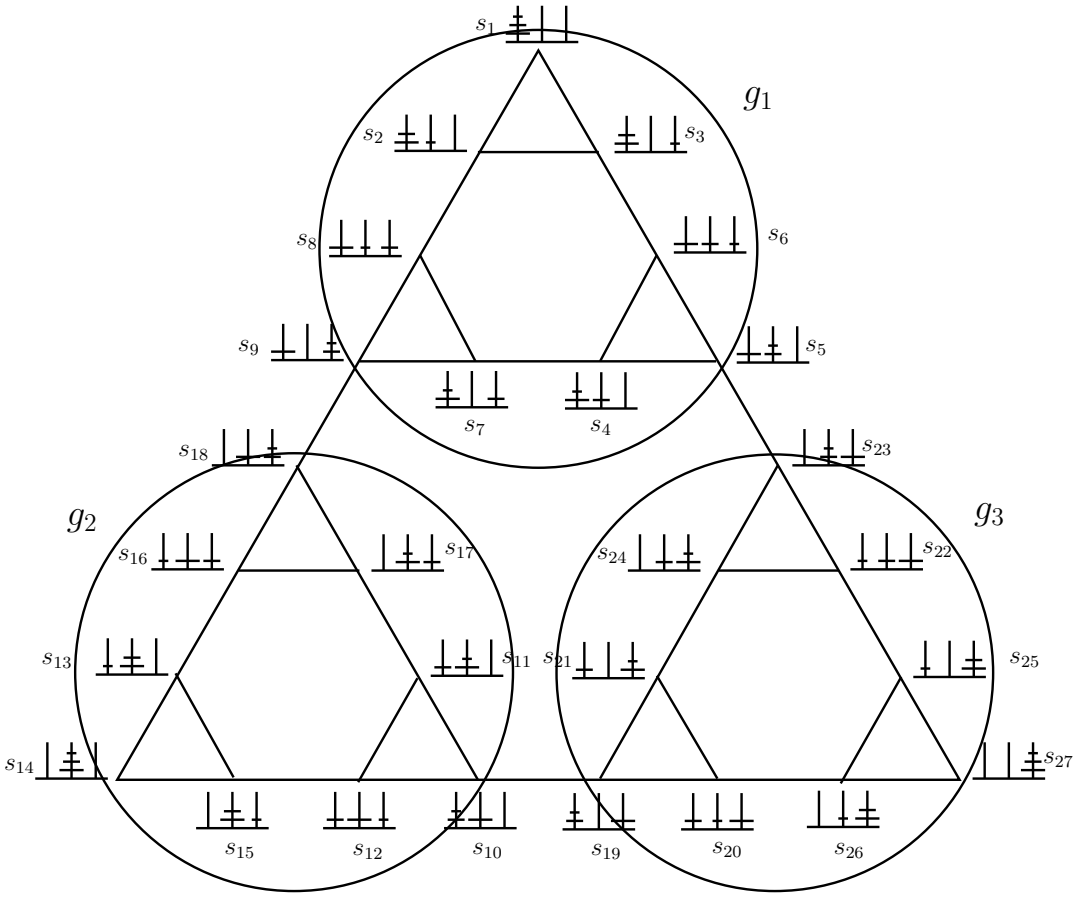


Figure 5.7: The granulated attributed graph defined by {C}

granule	*C*
g_1	1
g_2	2
g_3	3

Table 5.5: The information table of the granulated attributed graph defined by {C}

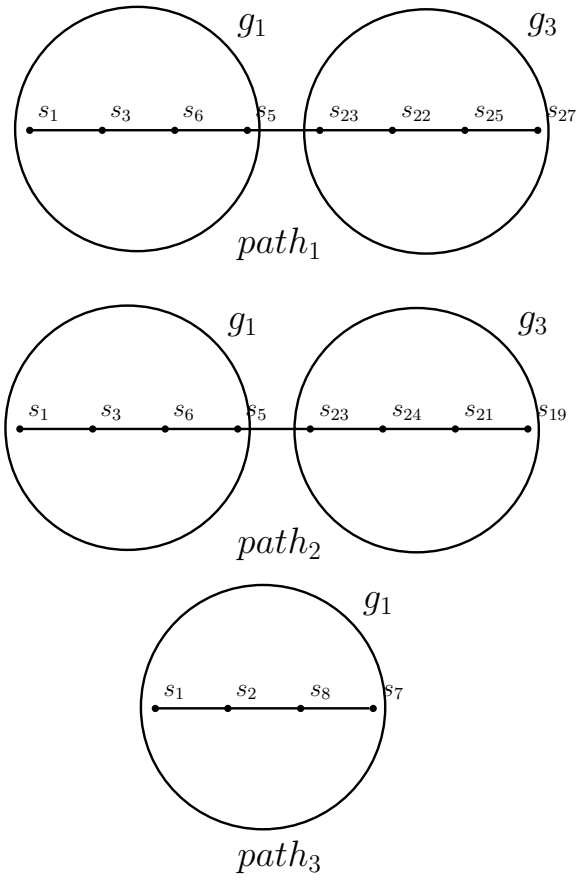


Figure 5.8: The paths in the granulated attributed graph defined by {C}

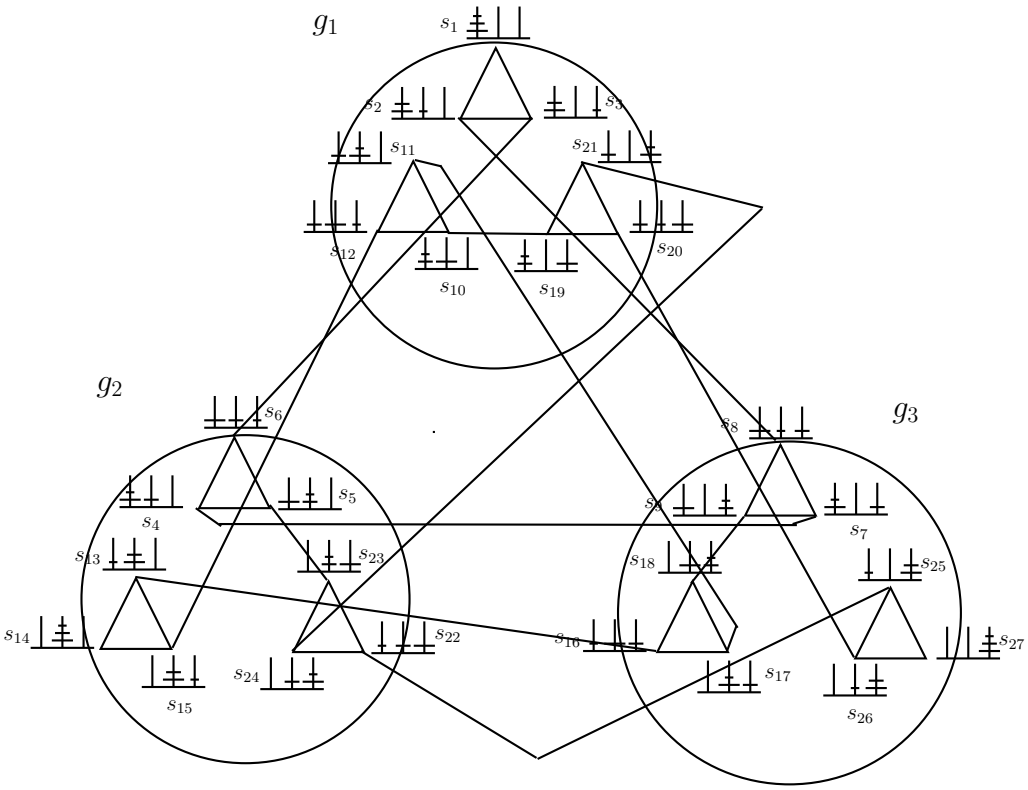


Figure 5.9: The granulated attributed graph defined by {B}

granule	*B*
g_1	1
g_2	2
g_3	3

Table 5.6: The information table of the granulated attributed graph defined by {B}

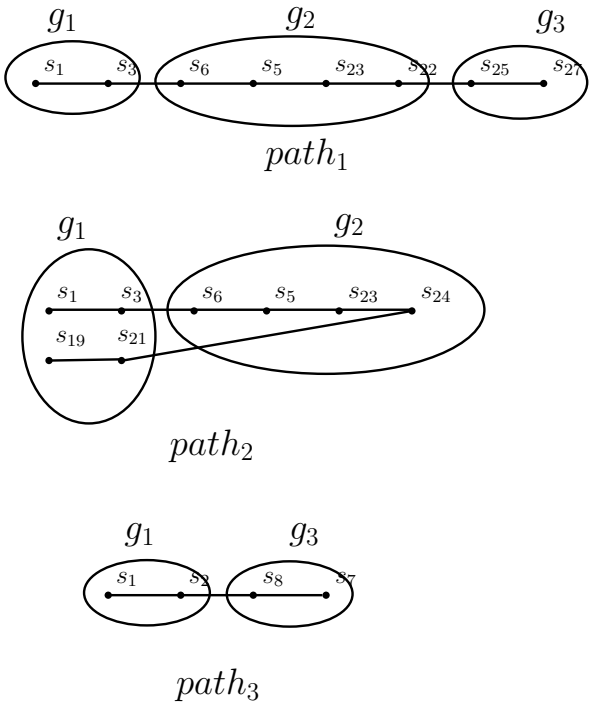


Figure 5.10: The paths in the granulated attributed graph defined by {B}

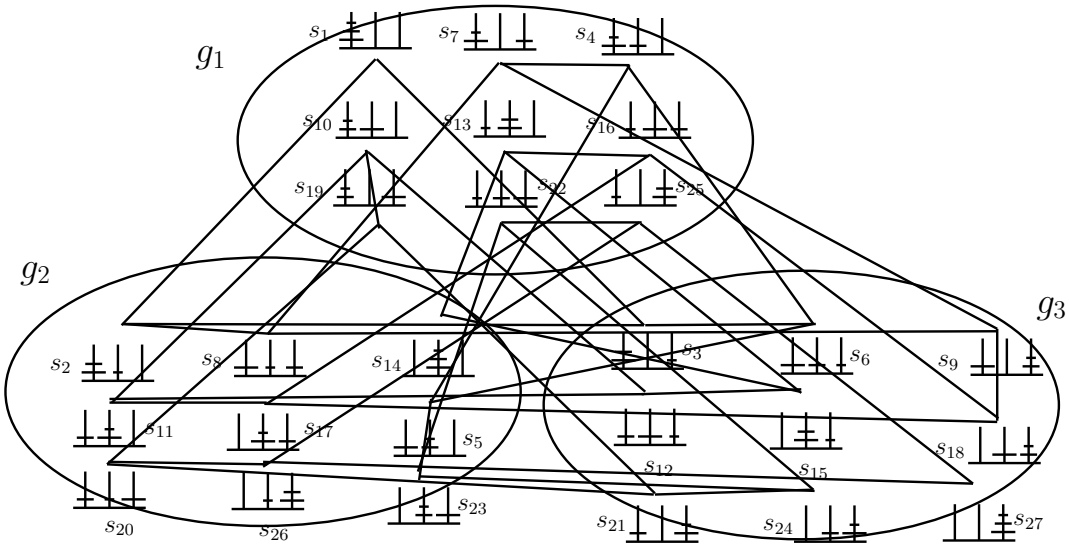


Figure 5.11: The granulated attributed graph defined by {A}

granule	*A*
g_1	1
g_2	2
g_3	3

Table 5.7: The information table of the granulated attributed graph defined by {A}

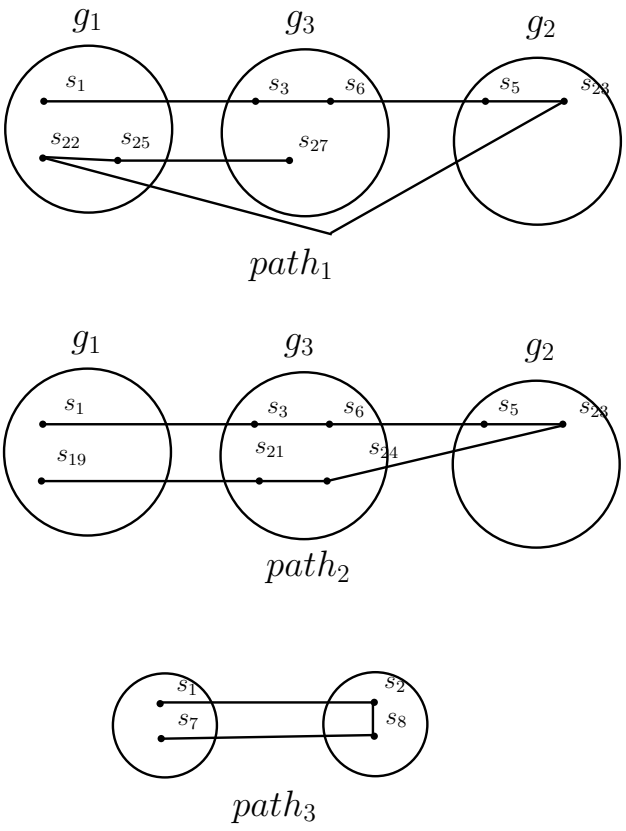


Figure 5.12: The paths in the granulated attributed graph defined by {A}

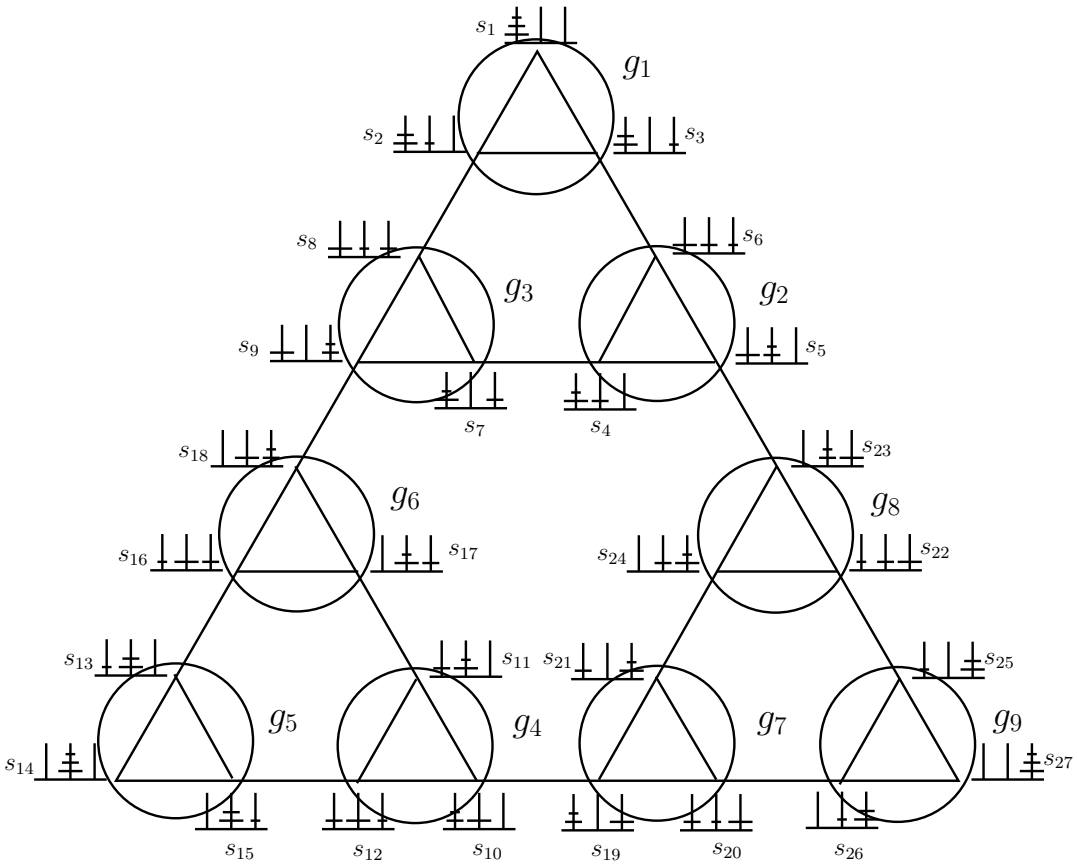


Figure 5.13: The granulated attributed graph defined by $\{B, C\}$

$granule$	B	C
g_1	1	1
g_2	2	1
g_3	3	1
g_4	1	2
g_5	2	2
g_6	3	2
g_7	1	3
g_8	2	3
g_9	3	3

Table 5.8: The information table of the granulated attributed graph defined by $\{B, C\}$

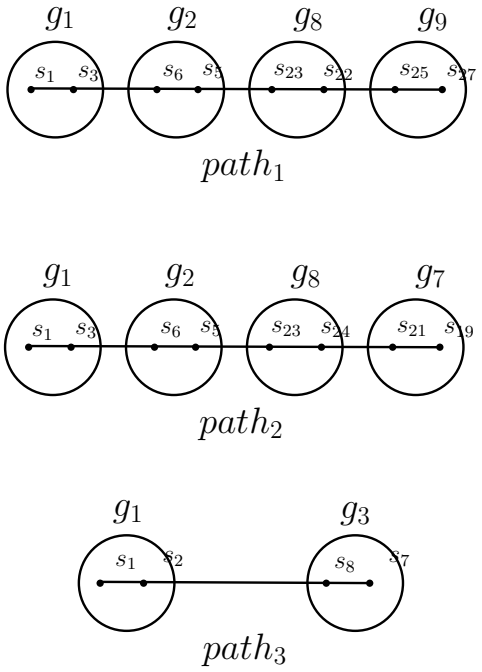


Figure 5.14: The paths in the granulated attributed graph defined by $\{B, C\}$

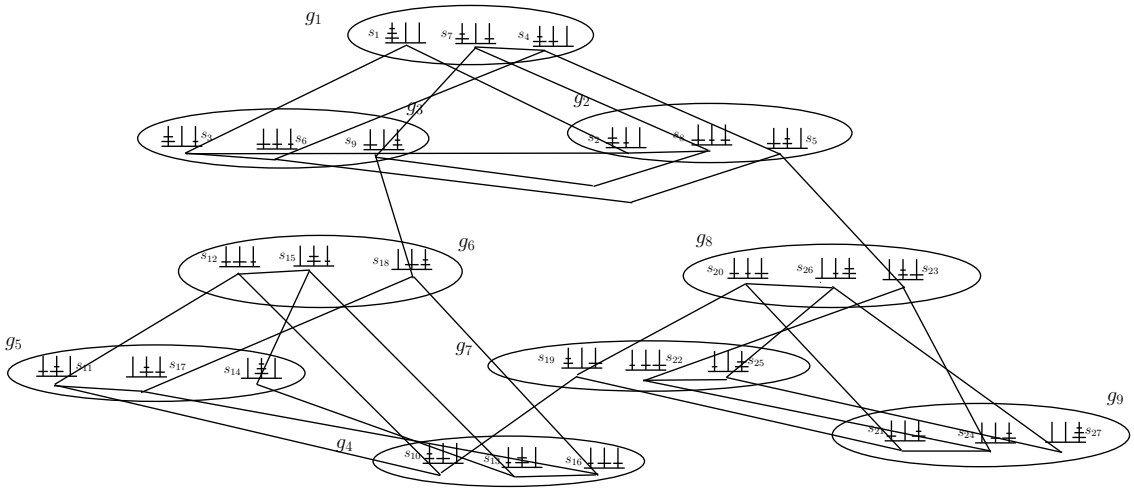


Figure 5.15: The granulated attributed graph defined by $\{A, C\}$

granule	C	A
g_1	1	1
g_2	1	2
g_3	1	3
g_4	2	1
g_5	2	2
g_6	2	3
g_7	3	1
g_8	3	2
g_9	3	3

Table 5.9: The information table of the granulated attributed graph defined by $\{A, C\}$

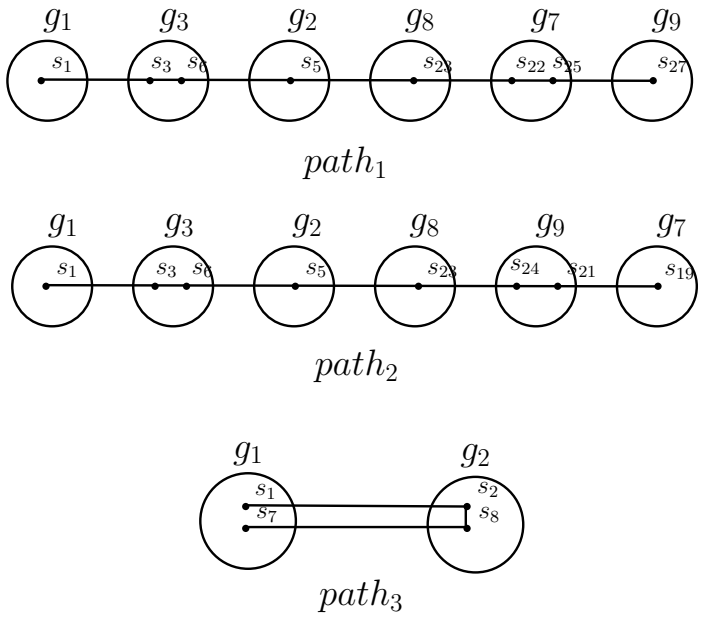


Figure 5.16: The paths in the granulated attributed graph defined by $\{A, C\}$

5.3 A New Refinement Procedure

In this section, we propose a new refinement procedure, named *Partial Solution Retained Backtracking* ($PSRB$) refinement. $PSRB$ refinement is a ordered refinement procedure. The techniques in $PSRB$ refinement are different from those in the traditional refinement procedure. The major purpose of $PSRB$ refinement is to reduce the impact of backtracking to as few as possible. Thus, $PSRB$ refinement is more efficient than the traditional refinement procedure.

As we know that backtracking means that the old sub-solutions to sub-problems or the higher level abstract solutions have to be abandoned. The impact of backtracking is that the work for finding the old sub-solutions and the higher level abstract solutions is wasted. The principle of reducing the impact of back-

tracking is to reduce the waste of work by using part of the old sub-solutions or the higher level abstract solutions as much as possible. $PSRB$ refinement can reuse the old sub-solutions or the higher level abstract solutions to a great extent, as well as speeding up the search process. It consists of three parts, namely, state-oriented depth-first search, prioritized operators, and recursive breakpoint method.

5.3.1 State-Oriented Depth-First Search

In $PSRB$ refinement, the searches in all levels of abstractions and original state space are depth-first search. The search in top level abstraction is a traditional depth-first search. The search in every non-top level abstraction and original state space is an *abstract solution restrained depth-first search*. That is, whenever we explore a state s, we only consider the succeeding states that are either in the same pre-image set of s or in the succeeding pre-image set if this succeeding pre-image set exists. The succeeding pre-image set is the pre-image set of the next abstract state in the higher level abstract solution. As in $PSRB$ refinement we consider abstract states in the higher level abstract solution, this search is state-oriented. There does not exist the step of choosing a pre-image of an abstract operator. The search process is not explicitly divided into sub-problems. The abstract states in the higher level abstract solution act as constraints on choosing search branches. Every explored state is recorded so that it will not be explored again.

Figure 5.17 is an example of the state-oriented depth-first search in a 2-level

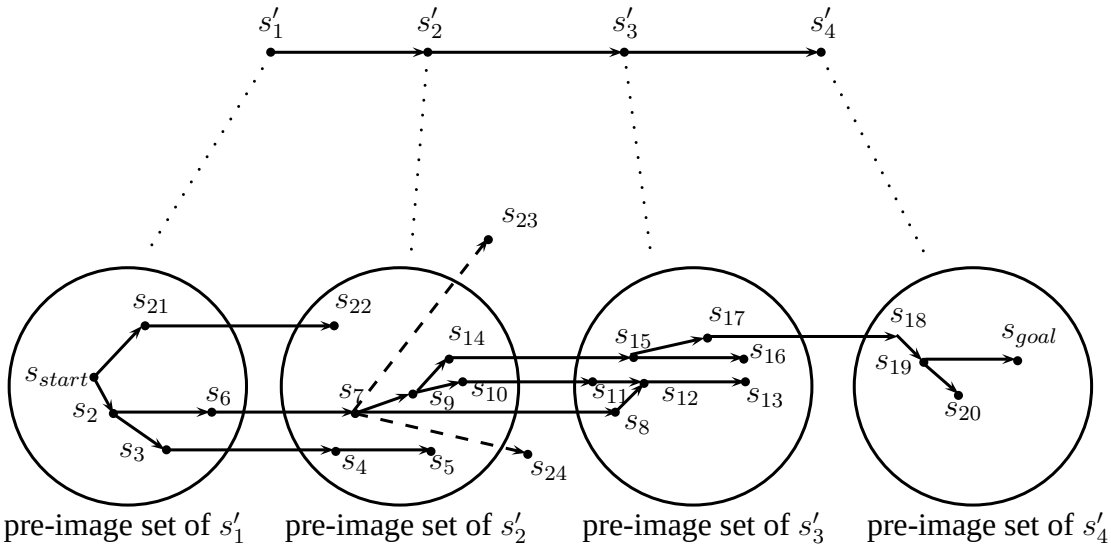


Figure 5.17: The state-oriented depth-first search

abstraction hierarchy. The abstract solution is $s'_1 s'_2 s'_3 s'_4$. The search process in the original state space first visits states in this order: s_{start}, s_2, s_3, s_4, s_5, s_6, s_7 ,s_8, s_{12}, s_{13}, s_9, s_{10}, s_{11}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{goal}, it then detects the solution $s_2 s_6 s_7 s_9 s_{14} s_{15} s_{17} s_{18} s_{19} s_{goal}$. State s_7 has four unexplored succeeding states s_8, s_9, s_{23} and s_{24}, the abstract solution constrains the search to explore only s_8 and s_9 and cut off s_{23} and s_{24} which are neither in the same pre-image set of s_7 nor in the succeeding pre-image set (the pre-image set of s'_3).

One advantage of the state-oriented depth-first search over the traditional refinement procedure is that it does not need to choose a pre-image of an abstract operator. The drawback of choosing a pre-image of an abstract operator is that search is restricted to only one transitional operator (the operator that transforms a state into another state in the succeeding pre-image set). Thus, when one transitional operator fails, the search for another transitional operator can-

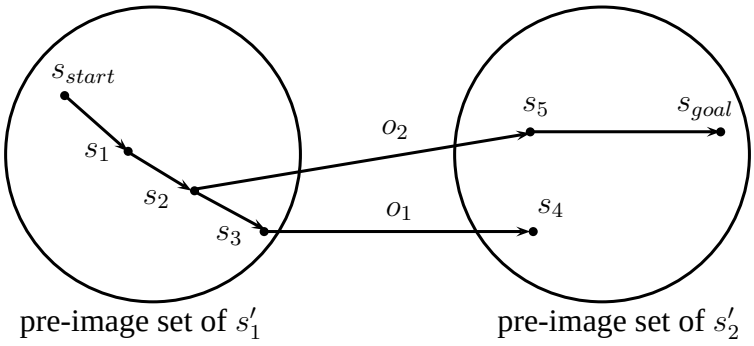


Figure 5.18: The advantage of the state-oriented depth-first search

not reuse the previous work. The state-oriented depth-first search does not have such a drawback. Take Figure 5.18 as an example. In the traditional refinement procedure, o_1 is first chosen and a solution $s_{start}s_1s_2s_3$ to the first sub-problem is found. As the second sub-problem cannot be solved, the first solution is abandoned and o_2 is chosen. The search process starts from s_{start} to find another solution $s_{start}s_1s_2$ to the first sub-problem. We can see that the work for finding a path from s_{start} to s_2 is done twice. In the state-oriented depth-first search, this extra work can be avoided. When the search finds that s_4 is a dead end, it goes back to s_2 to explore an alternative branch that leads to s_5.

5.3.2 Prioritized Operators

Based on an abstract solution, operators are prioritized into two parts, priority set and non-priority set. The priority set is the pre-image set of the corresponding abstract operator in the abstract solution. All other operators are the non-priority set. When exploring a state, the operators in the priority set are considered first, then the operators in the non-priority set are considered. Because the priority

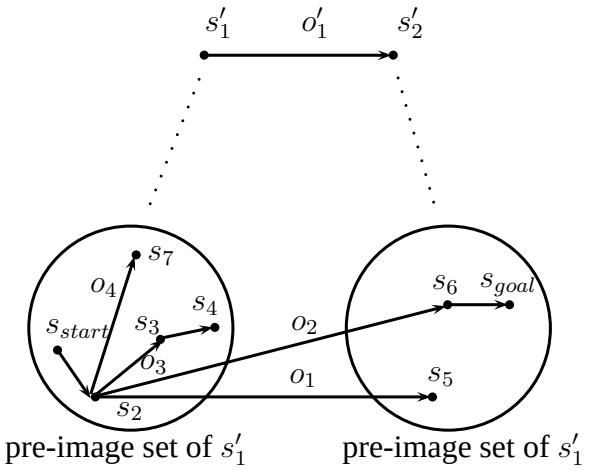


Figure 5.19: Prioritized operators

operators are more likely to transform states in the current pre-image set into a state in the succeeding pre-image set, the priority operators are applied to the states first for trying to reach a state in the succeeding pre-image set earlier.

For example, in Figure 5.19, the abstract operator o'_1 has two pre-images o_1 and o_2. For all states within the pre-image set of s'_1, the priority set are operators o_1 and o_2, all other operators are the non-priority set. When exploring s_2, o_1 is first applied to s_2, which leads to a dead end. Then o_2 is applied to s_2, which leads to s_{goal}. Thus, other operators such as o_3 and s_4 can be ignored. If we do not distinguish o_1, o_2, o_3 and o_4, we may apply o_3 or s_4 to s_2 first, which leads to a dead end.

Note that operators in the non-priority set may also transform states in the current pre-image set to a state in the succeeding pre-image set. There may be another abstract operator o'_2 that can transform s'_1 to s'_2, and the operators in the pre-image set of o'_2 are in the non-priority set, but they can still transform

states in the pre-image set of s'_1 into states in the pre-image set of s'_2.

5.3.3 A Recursive Breakpoint Method

Breakpoints are used in abstract solution backtracking. The traditional breakpoint method records the failure place, or breakpoint, when searching a solution in an abstraction, and goes back to the higher abstraction to give up all the states in the higher abstract solution that are after the breakpoint [35]. For example, assume there is a 3-level abstraction hierarchy $SP \ll SP' \ll SP''$, abstract solutions S'' to SP'' and S' to SP' are found, but the search in SP fails at a breakpoint. The traditional breakpoint method can use this breakpoint in SP to help abandon abstract states in S' when backtracking to SP'. If the search process continues to backtrack to SP'', the traditional breakpoint method does not consider the breakpoint in SP to help abandon abstract states in S''. That is, the traditional breakpoint method only considers the breakpoint from the immediate lower level abstraction. If we can consider the breakpoints from all lower level abstractions, we may improve the efficiency of backtracking. Recursive breakpoint method is such a breakpoint method.

As we use depth-first search in $PSRB$ refinement, we can make a search tree to record the search history of the search process in every abstraction. Every node in this tree is a state that has already been explored. The leaf nodes are dead end states, and states that cause lower level search to fail are also regarded as dead end states. When we go back to a higher level abstraction, we consider all dead end states and choose the one that is the nearest to the goal state as

the breakpoint. The distance from dead end states to the goal state is measured by how far the dead end states' corresponding abstract states are from the goal state's corresponding abstract state in the abstract solution.

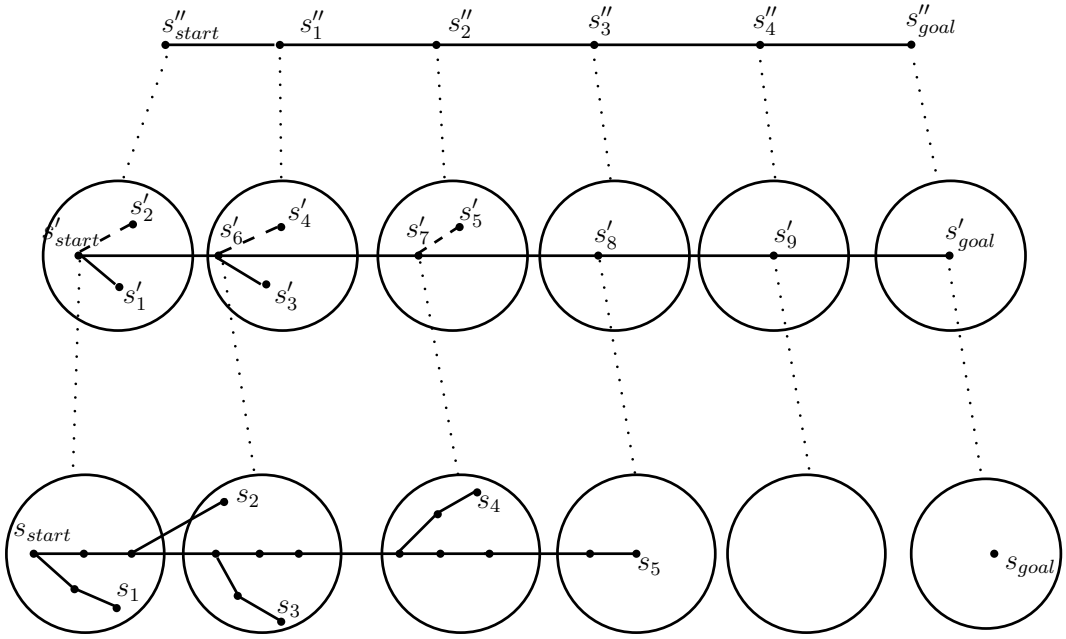


Figure 5.20: The recursive breakpoint method

We use Figure 5.20 to explain the recursive breakpoint method. There are three abstractions $SP \ll SP' \ll SP''$. A solution in SP'' is found, which is $s''_{start}s''_1s''_2s''_3s''_4s''_{goal}$. Also, a solution in SP' is found, which is $s'_{start}s'_6s'_7s'_8s'_9s'_{goal}$. The solid lines are the already explored edges, s'_1 and s'_3 are dead end states. The search in SP is finished without finding a solution and all the explored edges form a search tree. s_1, s_2, s_3, s_4 and s_5 are dead end states. As the search in SP is finished and s_{goal} is not reached, the search fails, we compare all the dead end states in SP and find that s_5 is the nearest to s_{goal}. This is because s_5 corresponds to the abstract state s'_8, and s'_8 is nearer to s'_{goal} than any other dead end states'

corresponding abstract states. Now we go back to SP', s_5 is the breakpoint and its corresponding abstract state s'_8 should be recorded as a dead end state in SP'. For the abstract solution in SP', all states after s'_8 are abandoned. The partial solution is $s'_{start}s'_6s'_7s'_8$, and search starts from s'_8 again in SP'. Finally, all other edges (the dashed lines) in SP' are explored and no other solutions are found in SP', which means that we should go back to SP''. Again, we compare all the dead end states and find that s'_8 is the nearest to s'_{goal}, so we take s'_8 as the breakpoint. s'_8's corresponding abstract state s''_3 is recorded as a dead end state and all abstract states in the solution after s''_3 are abandoned. Then, the search starts from s''_3 again in SP''. If there is a higher abstraction above SP'', the same method is used to identify the breakpoint in SP'' and report the breakpoint to its higher abstraction. This method can be used in any abstraction hierarchies with any number of levels. Note that in this example the breakpoint s'_8 is an abstract state of the breakpoint s_5. This does not mean that a higher level breakpoint must be the abstract state of a lower level breakpoint. As shown by Figure 5.21, if SP' has a different search tree, s'_{11} should be the breakpoint instead of s'_8 for SP', because s'_{11} is a dead end state and is nearer to the goal state than s'_8.

As long as a solution can be found by an abstraction hierarchy, the recursive breakpoint method can find this solution. This can be seen as follows. If the search in an abstraction fails, we should find another abstract solution in the immediate higher abstraction. Let us assume that the old higher level abstract solution is $s'_{start}s'_1s'_2s'_3s'_4s'_5s'_6s'_{goal}$, and the abstract state of the breakpoint is s'_4. As we use depth-first search, in normal case when we search for a new solution

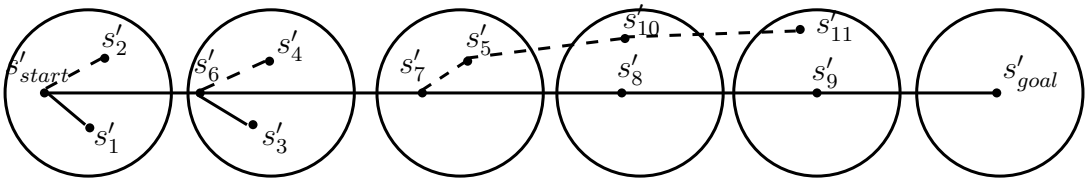


Figure 5.21: A different search tree

from the old one, we drop the last state in the old solution and try the untried operators on the second last state. That is, we drop s'_{goal}, resume the search from s'_6 by trying untried operators on s'_6. If all operators on s'_6 cannot lead to a solution, we then drop s'_6 and resume the search from s'_5. The breakpoint tells us that we do not need to resume the search from s'_6, we can drop all the states after s'_4, and resume the search from s'_4. This is because the breakpoint means that the search in the lower level abstraction cannot reach the pre-image set of s'_5. If we resume from s'_5 and find a new solution, this new solution still has $s'_{start}s'_1s'_2s'_3s'_4s'_5$, which means that the lower level search will still fail. We can safely drop all states after s'_4.

5.3.4 Algorithms

Algorithm 5.6 gives the algorithm to get prioritized operators and non-prioritized operators for an abstract operator in an abstraction at a level. The input parameters are *AttributeSet*, which is the attribute set of the abstraction, and *AbstractOperator*, which is the abstract operator. The output parameters are *PrioritizedOperators* and *NonPrioritizedOperators* which are the prioritized set and non-prioritized set, respectively.

Algorithm 5.6 Generating prioritized operators for an abstraction

```
1: procedure GetPrioritizedOperators(AttributeSet, AbstractOperator,
   PrioritizedOperators, NonPrioritizedOperators)
2:     PrioritizedOperators ← ∅
3:     NonPrioritizedOperators ← ∅
4:     for all p ∈ original operator set do
5:         op is created by removing all attributes in p that are not in
   AttributeSet
6:         if op's higher abstract operator is AbstractOperator then
7:             PrioritizedOperators ← op
8:         else
9:             NonPrioritizedOperators ← op
10:        end if
11:    end for
12: end procedure
```

The algorithm for finding solutions by $PSRB$ Refinement is given by Algorithm 5.7. The function *FindSolutions* is a recursive function to find the solutions at all levels of an abstraction hierarchy by $PSRB$ refinement. The input parameters are *SequenceOfAttributeSets*, *HigherOperatorSolution* and *HigherStateSolution*. *SequenceOfAttributeSets* is the sequence of all the attribute sets from the current processed level to the ground level. *HigherOperatorSolution* is the higher level abstract solution expressed by operators. *HigherStateSolution* is the higher level abstract solution expressed by states. The output parame-

ters of the function are *ReturnedBreakpoint*, *ReturnedSequenceOfOperatorSolution*, *ReturnedSequenceOfStateSolution*. *ReturnedBreakpoint* is the breakpoint of the $PSRB$ refinement for the higher level. *ReturnedSequenceOfOperatorSolution* is the sequence of found solutions (expressed by operators) from the current processed level to the ground level. *ReturnedSequenceOfStateSolution* is the sequence of found solutions (expressed by states) from the current processed level to the ground level. The function returns a boolean value $true$ or $false$. If it returns $true$, *ReturnedBreakpoint* is undefined. If it returns $false$ *ReturnedSequenceOfOperatorSolution* and *ReturnedSequenceOfStateSolution* are undefined. The meanings of variables are given in Table 5.10 and Table 5.11.

The function first initializes variables (from step 2 and step 13). In the while loop (from step 14 to step 78), it tries to find the solution of current processed level and then find all solutions of below levels recursively. In every iteration of the while loop, it first finds the prioritized set and non-prioritized set of the operators for *CurrentAbstractOperator* (from step 15 to step 17). Then it tries to apply all operators to *CurrentState* (from step 18 to step 65). In step 18, $op \in$ *PrioritizedOperators+NonPrioritizedOperators* means prioritized operators are tried first, then non-prioritized operators are tried.

When a new state $ImmediateState$ is got, the next execution depends on fours cases.

Case 1 *ImmediateState* is a goal state (from step to 25 step 50). In this case, the operator *op* and the state *ImmediateState* are put into *OperatorSolution* and *StateSolution*, respectively (from step 26 to step 27). Then the func-

Variable	*Meaning*
CurrentAttributeSet	This is the attribute set of current processed level.
CurrentState	This is the state that is being explored.
NextAbstractState	This is *CurrentState*'s abstract state's succeeding abstract state.
CurrentAbstractOperator	This is the current abstract operator in the higher abstract solution, which transforms *CurrentState*'s abstract state to *NextAbstractState*.
GoalStates	This is the goal states in current processed level.
NearestDeadEnd	This records the dead end state that is the nearest to the goal state at the higher level. Every time a new dead end state is identified, *NearestDeadEnd* will be replaced by the new dead end state if the new dead end state is more nearer to the goal.
StateSolution	This records the solution expressed by states at the current processed level.

Table 5.10: Meanings of variables

Variable	*Meaning*
OperatorSolution	This records the solution expressed by operators at the current processed level.
ExploredStateOperatorPairs	This records the already tried operators for every state.
PrioritizedOperators	This is the prioritized set of operators of the current processed level.
NonPrioritizedOperators	This is the non-prioritized set of operators of the current processed level.

Table 5.11: Meanings of variables (continued)

tion creates a sequence of attribute sets for the below levels (step 28). If *LowerSequenceOfAttributeSets* is empty, it means there is no lower level, then the function can return true (from step 29 to step 32). If there are lower levels, the function recursively gets all the solutions of the below levels (from step 33 to step 35). If it successfully gets all the solutions of the below levels, it returns true (from step 36 to step 38), else it removes all states and operators after *ReturnedLowerBreakpoint* in the *StateSolution* and *OperatorSolution*, respectively, and records the breakpoint as a dead end state by updating *NearestDeadEnd* then searches again from the *ReturnedLowerBreakpoint* (from step 39 to step 49).

Case 2 There is no *HigherStateSolution* (which means the current processed level

is the top level) or *ImmediateState* is in the same pre-image set as *CurrentState* (from step 51 to step 55). In this case, the function records the new state *ImmediateState* and searches again from *ImmediateState* (from step 52 to step 55).

Case 3 *ImmediateState* is in the pre-image set of *NextAbstractState*. In this case, the function records the new state *ImmediateState*, pushes *op* and *ImmediateState* into *OperationSolution* and *StateSolution*, respectively (from step 56 to step 59). Because the search now considers the next abstract state of the higher abstract solution, the function changes *CurrentAbstractOperotr* and *NextAbstractState* (from step 60 to step 61) and searches from *ImmediateState* again (step 62).

Case 4 None of the above cases is true. In this case, the function continues to try another operator (step 65).

If the execution exits from the loop of trying all operators, it means no *ImmediateState* can be got from *CurrentState*, so *CurrentState* is a dead end state. The function replaces *NearestDeadEnd* with *CurrentState* if the abstract state of *CurrentState* is after the abstract state of *NearestDeadEnd* in *HigherStateSolution* (from step 66 to step 68). Then the function tries to search again from the state before *CurrentState* (from step 69 to step 70). If there is no state before *CurrentState*, it means the search fails, no solution can be found (from step 71 to step 73). Else, it searches again from the state before *CurrentState* (from step 75 to step 76).

Algorithm 5.7 Finding the solutions at all levels of an abstraction hierarchy by *PSRB* refinement

1: **function** *FindSolutions*(*SequenceOfAttributeSets*, *HigherOperatorSolution*, *HigherStateSolution*, *ReturnedBreakpoint*, *ReturnedSequenceOfOperatorSolution*, *ReturnedSequenceOfStateSolution*)

2: *CurrentAttributeSet* ← the first attribute set of *SequenceOfAttributeSets*

3: *CurrentAbstractOperator* ← the first abstract operator in *HigherOperatorSolution*

4: *GoalStates* is created by removing *CurrentAttributeSet* from goal states

5: *CurrentState* is created by removing *CurrentAttributeSet* from the start state

6: *NextAbstractState* ← the second abstract state in *HigherStateSolution*

7: *NearestDeadEnd* ← *CurrentState*

8: *StateSolution* ← ∅

9: push *CurrentState* into *StateSolution*

10: *OperatorSolution* ← ∅

11: *ReturnedSequenceOfOperatorSolution* ← ∅

12: *ReturnedSequenceOfStateSolution* ← ∅

13: *ExploredStateOperatorPairs* ← ∅

14: **while** true **do**

Algorithm 5.7 Finding the solutions at all Levels of an abstraction hierarchy by $PSRB$ refinement (continued I)

15: $PrioritizedOperators \leftarrow \emptyset$

16: $NonPrioritizedOperators \leftarrow \emptyset$

17: $GetPrioritizedOperators(CurrentAttributeSet$, $CurrentAbstractOperator$, $PrioritizedOperators$, $NonPrioritizedOperators)$

18: **for all** $op \in PrioritizedOperators + NonPrioritizedOperators$ **do**

19: **if** ($CurrentState$,op) is in $ExploredStateOperatorPairs$ **then**

20: continue

21: **end if**

22: push $CurrentState$,op) into $ExploredStateOperatorPairs$

23: $ImmediateState$ is created by transforming $CurrentState$ by op

24: **if** $ImmediateState$ is not already in $StateSolution$ **then**

25: **if** $ImmediateState$ is in $GoalStates$ **then**

26: push op into $OperatorSoltion$

27: push $ImmediateState$ into $StateSolution$

28: $LowerSequenceOfAttributeSets$ is created by removing the first attribute set from $SequenceOfAttributeSets$

29: **if** $LowerSequenceOfAttributeSets = \emptyset$ **then**

Algorithm 5.7 Finding the solutions at all levels of an abstraction hierarchy by *PSRB* refinement (continued II)

30: push *OperatorSoltion* into *ReturnedSequenceOfOperatorSolution*

31: push *StateSoltion* into *ReturnedSequenceOfStateSolution*

32: return true

33: **else**

34: *ReturnedLowerBreakpoint* $\leftarrow \emptyset$

35: **if** *FindSolutions*(*LowerSequenceOfAttributeSets*, *OperatorSoltion*, *StateSoltion*, *ReturnedLowerBreakpoint*, *ReturnedSequenceOfOperatorSolution*, *ReturnedSequenceOfStateSolution*) = *true* **then**

36: push *OperatorSoltion* into *ReturnedSequenceOfOperatorSolution*

37: push *StateSoltion* into *ReturnedSequenceOfStateSolution*

38: return true

39: **else**

40: remove all states after *ReturnedLowerBreakpoint* in *StateSolution*

Algorithm 5.7 Finding the solutions at all levels of an abstraction hierarchy by *PSRB* refinement (continued III)

41: remove corresponding operators in *OperatorSolution*

42: *CurrentState* ← *ReturnedLowerBreakpoint*

43: *NextAbstractState* ← the abstract state next to the abstract state of *CurrentState* in *HigherStateSolution*

44: *CurrentAbstractOperator* ← the abstract operator corresponds to *NextAbstractState* in *HigherOperatorSolution*

45: **if** the abstract state of *CurrentState* is after the abstract state of *NearestDeadEnd* in *HigherStateSolution* **then**

46: *NearestDeadEnd* = *CurrentState*

47: **end if**

48: break

49: **end if**

50: **end if**

51: **else if** *HigherStateSolution* = ∅ or *ImmediateState* is in the same pre-image set as *CurrentState* **then**

52: *CurrentState* ← *ImmediateState*

53: push *op* into *OperatorSoltion*

54: push *ImmediateState* into *StateSolution*

55: break

56: **else if** *ImmediateState* is in the pre-image set of *NextAbstractState* **then**

Algorithm 5.7 Finding the solutions at all levels of an abstraction hierarchy by $PSRB$ refinement (continued IV)

57: $CurrentState \leftarrow ImmediateState$

58: push op into $OperatorSoltion$

59: push $ImmediateState$ into $StateSolution$

60: $CurrentAbstractOperator \leftarrow$ the abstract operator next to $CurrentAbstractOperator$ in $HigherOperatorSolution$

61: $NextAbstractState \leftarrow$ the abstract state next to $NextAbstractState$ in $HigherStateSolution$

62: break

63: **end if**

64: **end if**

65: **end for**

66: **if** the abstract state of $CurrentState$ is after the abstract state of $NearestDeadEnd$ in $HigherStateSolution$ **then**

67: $NearestDeadEnd = CurrentState$

68: **end if**

69: pop the last state from $StateSolution$

70: pop the last operator from $OperatorSolution$

71: **if** $StateSolution = \emptyset$ **then**

72: $ReturnedBreakpoint \leftarrow$ the abstract state of $NearestDeadEnd$

73: return false

Algorithm 5.7 Finding the solutions at all levels of an abstraction hierarchy by $PSRB$ refinement (continued V)

```
74:         else
75:             CurrentState ← the last state in StateSolution
76:             continue
77:         end if
78:     end while
79: end function
```

5.4 Advantages of Granular State Space Search

One advantage of granular state space search is that the abstraction hierarchies have clear structures and every abstract state is semantically meaningful. As proposed by Pedrycz [57], granules should be justifiable, that is granules should have meanings. The granular computing methods in granular state space search guarantee that every granule is meaningful. Thus, the abstraction hierarchies in granular computing are easy to understand, which gives us clear views of problems and can help us understand problems. And the structures of the abstractions are clear in the sense that connections within an abstract state and between abstract states are clear and understandable. For example, the abstractions shown in Figure 5.7 and Figure 5.13 have clear semantical meanings, that is every abstract state can be easily described, e.g., g_1 in Figure 5.7 can be described as "*disk* C is on *peg* 1". On the other hand, the abstractions shown in Figure 5.15 and

Figure 5.11 do not have clear semantical meanings and structures, and these abstractions are not chosen by granular state space search. The semantic meanings and clear structures make it easy to further process the abstraction hierarchies in granular state space search, such as storing and retrieving abstraction hierarchies.

The other advantage of granular state space search is that it is efficient. As the learning based method tries its best to reduce the occurrences of backtrackings and increase completeness degree of abstractions, it may have a good efficiency. We will show its efficiency by experiment in the next chapter.

5.5 Conclusion

In this chapter, we first explain how to use an attributed graph to represent a state space and how to use a granulated attributed graph to generate an abstraction hierarchy. We propose two methods for generating good abstraction hierarchies by using attributed graphs. We explain that the abstraction hierarchies created by our learning based method are semantically meaningful and easily understandable. Furthermore, we propose a new refinement procedure. In the next chapter, we will use experimental results to demonstrate that our learning based method for generating good abstractions has good efficiency.

Chapter 6

EXPERIMENTAL EVALUATIONS

In the previous chapter, we explained that our learning based method can generate abstraction hierarchies with semantical meanings and clear structures, which makes it easy to understand the abstraction hierarchies and to further process the abstraction hierarchies. In this chapter, we show the good efficiency of the learning based method by comparing the performance with that of Knoblock's method, Holte et al.'s method, depth-first search, and clique-based abstraction method [75], which is one of the most efficient methods.

We provide experimental results on applying the learning based method with $PSRB$ refinement to solve problems in four groups, i.e., a group of 5-puzzle problems, a group of 8-puzzle problems, a group of 7-disk Hanoi tower problems and a group of 9-disk Hanoi tower problems. Problems in the same group have the same state space and different start states or goal states. We solve all problems in

each group to verify the efficiency of our learning based method. The states and abstract states of the problems in these four groups can be easily expressed by state variables [22], thus these problems are suitable to be solved by our learning based method and other methods that generate abstraction hierarchies implicitly; and our learning based method can generate meaningful and understandable abstractions for these problems. We do not test problems which have only explicit graphs of state spaces, such as game maps [7], because there is not a good way to express these explicit graphs by state variables, thus it is hard to apply our learning based method to these problems.

6.1 Description of Problems

We described the three-disk Hanoi tower problem in Example 3.1. The 7-disk Hanoi tower problem and the 9-disk Hanoi tower problem are two variants of the three-disk Hanoi tower problem, from which the only difference is the number of disks. There are 7 and 9 disks in these two problems respectively, the rules are the same as those in the three-disk version. A group of 7-disk Hanoi tower problems contains all problems with the same state space, but with different start states or goal states.

In the 8-puzzle problem, there are 9 distinct tiles and a board with positions of 3 rows and 3 columns. Each tile occupies one position. There is one special tile among these 9 tiles, which is a blank tile. The blank tile can swap with any tiles that are adjacent to it. A state is a configuration of these tiles on these positions.

A solution is a sequence of movements that changes the start state into the goal state. If we choose different start states, we get different problems, all of which belong to the group of 8-puzzle problems. The 5-puzzle problem is similar to the 8-puzzle problem except that there are 6 distinct tiles and a board with positions of 3 rows and 2 columns.

6.2 Experiment Method

There are two steps in the problem solving for every group of problems. First we use training to get the ordered sequence of attribute sets, then we use this ordered sequence of attribute sets to build an abstraction hierarchy and use the $PSRB$ refinement to solve problems.

In the training method, we randomly select start state and goal state pairs. We then use breadth-first search to find solutions for every start state and goal state pair. As a solution may not exist, we set the searching depth as 20. That is, if a solution cannot be found within 20 levels of searching depth, we abort the search. If a solution is found within 20 levels of searching depth, we record this solution as a training sample path. For every group of problems, we obtain 50 training sample paths. We use these 50 training sample paths to choose the appropriate attributes by calculating the conflict values.

There are some considerations when solving a problem with an abstraction hierarchy and refinement procedure. First, in order to avoid searching exhaustively for an unsolvable problem, we should limit the searching depth for every depth-

first search in the $PSRB$ refinement. Second, as an abstraction hierarchy may not solve all the solvable problems, we combine the abstraction hierarchy method with brute-force search method, that is, if an abstraction hierarchy cannot find a path for a problem, we use depth-first search without abstraction hierarchies to search for the solution to this problem again. As most problems can be solved by abstraction hierarchies, only a few problems need brute-force search, the average efficiency of solving all the problems should be still better than brute-force search.

For the group of 5-puzzle problems and the group of 8-puzzle problems, we use attributes to represent the positions and values to represent the tiles. We use the first order logic language to describe the operators. Let us take the group of 5-puzzle problems as an example. We first number the positions, which are shown in Figure 6.1. We introduce 6 attributes A_0, A_1, A_2, A_3, A_4, A_5. Every attribute describes a position. The value range for every attribute is {0, 1, 2, 3, 4, 5}. Every value represents a tile. An attribute value pair means that a specific tile is on a specific position. For example, the attribute value pair $(A_1, 1)$ means that the 1st tile is on position 1. Value 0 represents the blank tile. If an attribute's value is 0, it means that the blank tile is on the position described by this attribute. A state is represented like $\{(A_0, 2)$, $(A_1, 4)$, $(A_2, 3)$, $(A_3, 5)$, $(A_4, 0)$, $(A_5, 1)\}$. Operators are shown by Table 6.1. The first operator means that if the blank tile is on position 0 and the 1st tile is on position 1, then change the blank tile to position 1 and the 1st tile to position 0.

For the groups of 7-disk Hanoi tower problems and 9-disk Hanoi tower problems, we use attributes to represent the disks and values to represent the pegs.

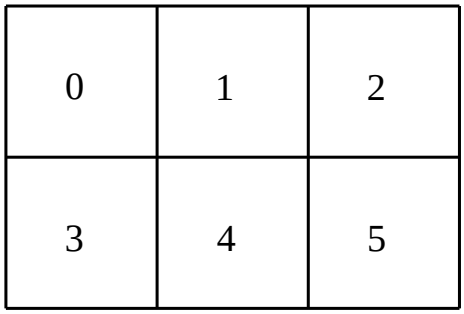


Figure 6.1: The position numbers of the group of 5-puzzle problems

No.	Precondition	Add	Del
1	$\{(A_0, 0), (A_1, 1)\}$	$\{(A_0, 1), (A_1, 0)\}$	$\{(A_0, 0), (A_1, 1)\}$
2	$\{(A_0, 0), (A_1, 2)\}$	$\{(A_0, 2), (A_1, 0)\}$	$\{(A_0, 0), (A_1, 2)\}$
1	$\{(A_0, 0), (A_1, 3)\}$	$\{(A_0, 3), (A_1, 0)\}$	$\{(A_0, 0), (A_1, 3)\}$
.....			

Table 6.1: The operators of the group of 5-puzzle problems

No.	Precondition	Add	Del
1	$\{(A_1, 1)\}$	$\{(A_1, 2)\}$	$\{(A_1, 1)\}$
2	$\{(A_1, 2)\}$	$\{(A_1, 1)\}$	$\{(A_1, 2)\}$
3	$\{(A_1, 1)\}$	$\{(A_1, 3)\}$	$\{(A_1, 1)\}$
.....			

Table 6.2: The operators of the group of 7-disk Hanoi tower problems

The operators are expressed by the first order logic language. Let us take the group of 7-disk Hanoi tower problems as an example. We use 7 attributes to represent 7 disks, which are A_1, A_2, A_3, A_4, A_5, A_6, A_7. The values for every attribute are {0, 1, 2}, which means *peg* 1, *peg* 2 and *peg* 3. An attribute value pair (A_1, 1) means that the disk A_1 is on *peg* 1. Operators are given by Table 6.2, which are similar to those in Knoblock's method(Table 3.1).

There is one remaining issue in our learning based method, that is we do not know what is the perfect number of levels for the abstraction hierarchies. So we will try different number of levels in the test for every group of problems.

6.3 Experimental Results and Analysis

The test environment is as follows: AMD E-450 1.65GHz CPU, 4GB memory, Windows 7 operating system, 200GB hard disk.

6.3.1 The Group of 8-Puzzle Problems

Knoblock's method fails to find any abstractions, because the operators do not satisfy the restrictions needed for this method. Therefore, when using Knoblock's method, we can only get a 1-level abstraction hierarchy, which is the original state space itself. Knoblock's method degrades to a brute-force method.

Results by Depth-First Search

The results by depth-first search for 1000 problems are shown by Table 6.3. Every problem is a pair of start state and goal state generated uniformly by random. In the group of 8-puzzle problems, only half of the problems are solvable. We can see that when we set depth limit as 25, depth-first search can solve all the solvable problems. When the depth limit is less than 25, it can solve only part of the solvable problems.

depth limit	solved problems	average explored states
15	58	17548
17	117	31990
20	216	61369
25	500	102321

Table 6.3: The results by depth-first search for the group of 8-puzzle problems

Results by Our Learning Based Method

After running the training method, we calculate that the attribute A_8 is the best one which has the least conflict value, followed by A_6, A_7, A_5, A_0, A_1, A_2, A_3, A_4.

The first ordered sequence of attribute sets we select is $\{A_8\} \subset \{A_8, A_6\} \subset \{A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$. We use this sequence to create a 3-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are shown by Table 6.4. In the column *depth limits* of every tuple, the first number is the searching depth for the top level abstraction, the second number is the searching depth for the first level abstraction, and the third number is the searching depth for the ground level abstraction (or the original state space). For every problem, we first try to solve it by the abstraction hierarchy. If the abstraction hierarchy fails, we then try to solve it by depth-first search. So the total solved problems are divided into two parts, one is solved by the abstraction hierarchy, the other is solved by depth-first search. In the column *solved problems* of every tuple, the number outside the bracket is the number of solved problems, the number inside the bracket is the number of problems solved by the abstraction hierarchy. The average explored states in this table include the cost of the training method.

depth limits	solved problems	average explored states
$(3, 4, 11)$	500(491)	32720
$(3, 4, 10)$	500(436)	37924
$(3, 4, 9)$	500(421)	39205
$(3, 4, 7)$	500(397)	42287
$(3, 5, 8)$	500(382)	43945
$(3, 6, 9)$	500(399)	42077
$(3, 7, 7)$	500(431)	39454

Table 6.4: The results by the abstraction hierarchy created by $\{A_8\}$, $\{A_8, A_6\}$, $\{A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$ for the group of 8-puzzle problems

The second ordered sequence of attribute sets we select is $\{A_8\} \subset \{A_8, A_6\} \subset \{A_8, A_6, A_7\} \subset \{A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$. We use this sequence to create a 4-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are shown by Table 6.5. The meanings of the column *depth limits* and column *solved problems* are the same as the previous test.

depth limits	solved problems	average explored states
$(3, 6, 6, 6)$	500(418)	46093
$(3, 6, 5, 7)$	500(409)	46547
$(3, 5, 6, 8)$	500(397)	47169
$(6, 6, 6, 5)$	500(419)	45787
$(3, 6, 7, 7)$	500(431)	42050
$(3, 6, 8, 9)$	500(455)	39923
$(3, 6, 9, 9)$	500(472)	37028

Table 6.5: The results by the abstraction hierarchy created by $\{A_8\}$, $\{A_8, A_6\}$, $\{A_8, A_6, A_7\}$, $\{A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$ for the group of 8-puzzle problems

The third ordered sequence of attribute sets we select is $\{A_8\} \subset \{A_8, A_6\} \subset \{A_8, A_6, A_7\} \subset \{A_8, A_6, A_7, A_5\} \subset \{A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$. We use this sequence to create a 5-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are shown by Table 6.6. The meanings of the column *depth limits* and column *solved problems* are the same as the previous test.

depth limits	solved problems	average explored states
$(2, 3, 3, 7, 6)$	500(371)	51120
$(2, 3, 3, 5, 6)$	500(365)	52978
$(3, 3, 3, 5, 7)$	500(398)	49402
$(3, 3, 5, 5, 7)$	500(419)	47673
$(2, 3, 4, 7, 6)$	500(397)	49675
$(3, 3, 3, 6, 7)$	500(407)	48121
$(3, 3, 3, 3, 6)$	500(364)	53126

Table 6.6: The results by the abstraction hierarchy created by $\{A_8\}$, $\{A_8, A_6\}$, $\{A_8, A_6, A_7\}$, $\{A_8, A_6, A_7, A_5\}$, $\{A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$ for the group of 8-puzzle problems

Results by Holte et al.'s Method

The results by Holte et al.'s method is shown by Table 6.7.

radius of abstraction	solved problems	average explored states
2	500	58726
3	500	53795
4	500	56947
5	500	57433
6	500	51209
7	500	50127

Table 6.7: The results by Holte et al.'s method for the group of 8-puzzle problems

Results by Clique-Based Abstraction Method

The results by clique-based abstraction method(size-2 clique) is shown by Table 6.8.

solved problems	average explored states
500	57923

Table 6.8: The results by clique-based abstraction method for the group of 8-puzzle problems

6.3.2 The Group of 5-Puzzle Problems

Knoblock's method again fails to find any abstractions, because the operators do not satisfy the restrictions needed by this method. This time Holte et al.'s method can solve this problem.

Results by Depth-First Search

The results by depth-first search for 1000 problems are shown by Table 6.9. Every problem is a pair of start state and goal state generated uniformly by random. In the group of 5-puzzle problems, only half of the problems are solvable. We can see that when we set depth limit as 25, depth-first search can solve all the solvable problems. When the depth limit is less than 25, it can solve only part of the solvable problems.

depth limit	solved problems	average explored states
10	100	88
15	317	134
20	487	221
25	500	397

Table 6.9: The results by depth-first search for the group of 5-puzzle problems

Results by Our Learning Based Method

After running the training method, we calculate that the attribute A_5 is the best one which has the least conflict value, followed by A_3, A_0, A_2, A_1, A_4.

The first ordered sequence of attribute sets we select is $\{A_5\} \subset \{A_5, A_3\} \subset \{A_0, A_1, A_2, A_3, A_4, A_5\}$. We use this sequence to create a 3-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are shown by Table 6.10. The meanings of the column *depth limits* and column *solved problems* are the same as the test in the group of 8-puzzle problems.

depth limits	solved problems	average explored states
$(2, 3, 8)$	500(497)	97
$(2, 3, 7)$	500(471)	102
$(2, 3, 6)$	500(442)	115
$(2, 4, 6)$	500(461)	106

Table 6.10: The results by the abstraction hierarchy created by $\{A_5\}$, $\{A_5, A_3\}$, $\{A_0, A_1, A_2, A_3, A_4, A_5\}$ for the group of 5-puzzle problems

The second ordered sequence of attribute sets we select is $\{A_5\} \subset \{A_5, A_3\} \subset \{A_5, A_3, A_0\} \subset \{A_0, A_1, A_2, A_3, A_4, A_5\}$. We use this sequence to create a 4-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are shown by Table 6.11. The meanings of the column *depth limits* and column *solved problems* are the same as the test in the group of 8-puzzle problems.

depth limits	solved problems	average explored states
$(3, 4, 7, 6)$	500(461)	103
$(2, 4, 6, 6)$	500(443)	105
$(2, 4, 5, 7)$	500(437)	107
$(2, 4, 4, 7)$	500(429)	110

Table 6.11: The results by the abstraction hierarchy created by $\{A_5\}$, $\{A_5, A_3\}$, $\{A_5, A_3, A_0\}$, $\{A_0, A_1, A_2, A_3, A_4, A_5\}$ for the group of 5-puzzle problems

The third ordered sequence of attribute sets we select is $\{A_5\} \subset \{A_5,\ A_3\} \subset \{A_5,\ A_3,\ A_0\} \subset \{A_5,\ A_3,\ A_0,A_2\} \subset \{A_0,\ A_1,\ A_2,\ A_3,\ A_4,\ A_5\}$. We use this sequence to create a 5-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are as Table 6.12 shows. The meanings of the column *depth limits* and column *solved problems* are the same as the test in the group of 8-puzzle problems.

depth limits	solved problems	average explored states
$(3, 3, 5, 5, 5)$	500(421)	117
$(2, 3, 4, 7, 6)$	500(418)	121
$(3, 3, 3, 6, 6)$	500(411)	123

Table 6.12: The results by the abstraction hierarchy created by $\{A_5\}$, $\{A_5,\ A_3\}$, $\{A_5,\ A_3,\ A_0\}$, $\{A_5,\ A_3,\ A_0,\ A_2\}$, $\{A_0,\ A_1,\ A_2,\ A_3,\ A_4,\ A_5\}$ for the group of 5-puzzle problems

Results by Holte et al.'s Method

The results by Holte et al.'s method is shown by Table 6.13.

radius of abstraction	solved problems	average explored states
2	500	96
3	500	100
4	500	121
5	500	127
6	500	109
7	500	142

Table 6.13: The results by Holte et al.'s method for the group of 5-puzzle problems

Results by Clique-Based Abstraction Method

The results by clique-based abstraction method(size-2 clique) is shown by Table 6.14.

solved problems	average explored states
500	99

Table 6.14: The results by clique-based abstraction method for the group of 5-puzzle problems

6.3.3 The Group of 7-Disk Hanoi Tower Problems

Both Holte et al.'s method and Knoblock's method can be applied to this group of problems.

Results by Depth-First Search

The results by depth-first search for 1000 problems are shown by Table 6.22. Every problem is a pair of start state and goal state generated uniformly by random. All the problems are solvable. We can see that when we set depth limit as 127, depth-first search can solve all the solvable problems. When the depth limit is less than 127, it can solve only part of the solvable problems.

depth limit	solved problems	average explored states
60	512	447
100	796	767
127	1000	960

Table 6.15: The results by depth-first search for the group of 7-disk Hanoi tower problems

Results by Our Learning Based Method

After running the training method, we calculate that the attribute A_7 is the best one which has the least conflict value, followed by A_6, A_5, A_4, A_3, A_2, A_1.

The first ordered sequence of attribute sets we select is $\{A_7\} \subset \{A_7, A_6\} \subset \{A_1, A_2, A_3, A_4, A_5, A_6, A_7\}$. We use this sequence to create a 3-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are shown by Table 6.16. The meanings of the column *depth limits* and column *solved problems* are the same as the test in the group of 8-puzzle problems.

depth limits	solved problems	average explored states
$(2, 5, 20)$	1000(870)	287
$(2, 5, 30)$	1000(912)	251
$(2, 5, 40)$	1000(1000)	219

Table 6.16: The results by the abstraction hierarchy created by $\{A_7\}$, $\{A_7, A_6\}$, $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7\}$ for the group of 7-disk Hanoi tower problems

The second ordered sequence of attribute sets we select is $\{A_7\} \subset \{A_7, A_6\} \subset \{A_7, A_6, A_5\} \subset \{A_1, A_2, A_3, A_4, A_5, A_6, A_7\}$. We use this sequence to create a 4-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are shown by Table 6.17. The meanings of the column *depth limits* and column *solved problems* are the same as the test in the group of 8-puzzle problems.

depth limits	solved problems	average explored states
$(2, 5, 7, 7)$	1000(721)	301
$(2, 5, 7, 10)$	1000(910)	247
$(2, 10, 8, 20)$	1000(1000)	221

Table 6.17: The results by the abstraction hierarchy created by $\{A_7\}$, $\{A_7, A_6\}$, $\{A_7, A_6, A_5\}$, $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7\}$ for the group of 7-disk Hanoi tower problems

Results by Holte et al.'s Method

The results by Holte et al.'s method are shown by Table 6.18.

radius of abstraction	solved problems	average explored states
2	1000	249
3	1000	238
4	1000	248
5	1000	263
6	1000	276
7	1000	296

Table 6.18: The results by Holte et al.'s method for the group of 7-disk Hanoi tower problems

Results by Knoblock's Method

The Knoblock's method finds the same abstraction hierarchies as our learning based method. We use the two abstraction hierarchies that are created by $\{A_7\}$, $\{A_7, A_6\}$, $\{A_1,\ A_2,\ A_3,\ A_4,\ A_5,\ A_6,\ A_7\}$ and by $\{A_7\}$, $\{A_7, A_6\}$, $\{A_7, A_6, A_5\}$, $\{A_1,\ A_2,\ A_3,\ A_4,\ A_5,\ A_6,\ A_7\}$.

depth limits	solved problems	average explored states
$(2, 5, 20)$	1000(870)	281
$(2, 5, 30)$	1000(912)	245
$(2, 5, 40)$	1000(1000)	212

Table 6.19: The results by Knoblock's method for the abstraction hierarchy created by $\{A_7\}$, $\{A_7, A_6\}$, $\{A_1,\ A_2,\ A_3,\ A_4,\ A_5,\ A_6,\ A_7\}$ for the group of 7-disk Hanoi tower problems

depth limits	solved problems	average explored states
$(2, 5, 7, 7)$	1000(721)	291
$(2, 5, 7, 10)$	1000(910)	239
$(2, 10, 8, 20)$	1000(1000)	217

Table 6.20: The results by Knoblock's method for the abstraction hierarchy created by $\{A_7\}$, $\{A_7, A_6\}$, $\{A_7, A_6, A_5\}$, $\{A_1,\ A_2,\ A_3,\ A_4,\ A_5,\ A_6,\ A_7\}$ for the group of 7-disk Hanoi tower problems

Results by Clique-Based Abstraction Method

The results by clique-based abstraction method(size-3 clique) is shown by Table 6.21.

solved problems	average explored states
500	256

Table 6.21: The results by clique-based abstraction method for the group of 7-disk Hanoi tower problems

6.3.4 The Group of 9-Disk Hanoi Tower Problems

Both Holte et al.'s method and Knoblock's method can be applied to this group of problems.

Results by Depth-First Search

The results by depth-first search for 1000 problems are shown by Table 6.22. Every problem is a pair of start state and goal state generated uniformly by random. All the problems are solvable. We can see that when we set depth limit as 511, depth-first search can solve all the solvable problems. When the depth limit is less than 511, it can solve only part of the solvable problems.

depth limit	solved problems	average explored states
200	503	3612
400	812	6902
511	1000	10211

Table 6.22: The results by depth-first search for the group of 9-disk Hanoi tower problems

Results by Our Learning Based Method

After running the training method, we calculate that the attribute A_9 is the best one which has the least conflict value, followed by A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1.

The first ordered sequence of attribute sets we select is $\{A_9\} \subset \{A_9, A_8\} \subset \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$. We use this sequence to create a 3-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are shown by Table 6.23. The meanings of the column *depth limits* and column *solved problems* are the same as the test in the group of 8-puzzle problems.

depth limits	solved problems	average explored states
$(2, 8, 50)$	1000(678)	7462
$(2, 8, 100)$	1000(845)	6021
$(2, 8, 150)$	1000(1000)	5986

Table 6.23: The results by the abstraction hierarchy created by $\{A_9\}$, $\{A_9, A_8\}$, $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$ for the group of 9-disk Hanoi tower problems

The second ordered sequence of attribute sets we select is $\{A_9\} \subset \{A_9, A_8\} \subset \{A_9, A_8, A_7\} \subset \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$. We use this sequence to create a 4-level abstraction hierarchy and use this abstraction hierarchy to solve 1000 problems. The results are shown by Table 6.24. The meanings of the column *depth limits* and column *solved problems* are the same as the test in the group of

8-puzzle problems.

depth limits	solved problems	average explored states
$(2, 8, 20, 30)$	1000(670)	7230
$(2, 8, 20, 50)$	1000(812)	6515
$(2, 8, 30, 70)$	1000(1000)	5131

Table 6.24: The results by the abstraction hierarchy created by $\{A_9\}$, $\{A_9, A_8\}$, $\{A_9, A_8, A_7\}$, $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$ for the group of 9-disk Hanoi tower problems

Results by Holte et al.'s Method

The results by Holte et al.'s method are shown by Table 6.25.

radius of abstraction	solved problems	average explored states
2	1000	8211
3	1000	8379
4	1000	7062
5	1000	7912
6	1000	7336
7	1000	7317

Table 6.25: The results by Holte et al.'s method for the group of 9-disk Hanoi tower problems

Results by Knoblock's Method

The Knoblock's method finds the same abstraction hierarchies as our learning based method. We use the two abstraction hierarchies that are created by $\{A_9\}$, $\{A_9, A_8\}$, $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$ and by $\{A_9\}$, $\{A_9, A_8\}$, $\{A_9, A_8, A_7\}$, $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$ to solve 1000 problems

depth limits	solved problems	average explored states
$(2, 8, 50)$	1000(678)	7401
$(2, 8, 100)$	1000(845)	5981
$(2, 8, 150)$	1000(1000)	5912

Table 6.26: The results by Knoblocks method for the abstraction hierarchy created by $\{A_9\}$, $\{A_9, A_8\}$, $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$ for the group of 9-disk Hanoi tower problems

depth limits	solved problems	average explored states
$(2, 8, 20, 30)$	1000(670)	7192
$(2, 8, 20, 50)$	1000(812)	6493
$(2, 8, 30, 70)$	1000(1000)	5071

Table 6.27: The results by Knoblocks method for the abstraction hierarchy created by $\{A_9\}$, $\{A_9, A_8\}$, $\{A_9, A_8, A_7\}$, $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$ for the group of 9-disk Hanoi tower problems

Results by Clique-Based Abstraction Method

The results by clique-based abstraction method(size-3 clique) is shown by Table 6.28.

solved problems	average explored states
500	6702

Table 6.28: The results by clique-based abstraction method for the group of 9-disk Hanoi tower problems

6.3.5 Comparison and Analysis

Table 6.29 is the comparison between the five methods. We compare the number of states that are explored to solve all the solvable problems (500 problems for the group of 8-puzzle problems and the group of 5-puzzle problems, 1000 problems for the group of 7-disk Hanoi tower problems and the group of 9-disk Hanoi tower problems). For depth-first search and clique-based abstraction method, there is only one result for every group of problems. For our learning based method and Knoblock's method, different choices of abstraction hierarchies and searching depth limits lead to different results for the same group of problems, so we consider the worst case, best case and average case for every group of problems. For Holte et al.'s method, different choices of radius of abstraction lead to different results for the same group of problems, so we consider the worst case, best case and average case for every group of problems.

		The Group of 8-Puzzle Problems	The Group of 5-Puzzle Problems	The Group of 7-Disk Hanoi Tower Problems	The Group of 9-Disk Hanoi Tower Problems
Depth-First Search		102321	397	960	10211
Our Learning Based Method	worst case	53126	123	301	7462
	best case	32720	97	219	5131
	average case	44490	110	254	6390
Holte et al.'s Method	worst case	58726	142	296	8379
	best case	50127	96	238	7062
	average case	54706	116	262	7703
Knoblock's Method	worst case	N/A	N/A	291	7401
	best case	N/A	N/A	212	5071
	average case	N/A	N/A	249	6342
Clique-Based Abstraction Method		57923	99	256	6702

Table 6.29: The comparison between five methods

First, we compare our learning based method with depth-first search. Our learning based method is far more better than depth-first search. Even for the worst case, the cost of our learning based method is 52% of that of depth-first search for the group of 8-puzzle problems, 31% for the group of 5-puzzle problems, 31% for the group of 7-disk Hanoi tower problems and 73% for the group of 9-disk Hanoi tower problems. When comparing by average case, the costs of our learning based method are 43%, 28%, 26%, 63% of those of depth-first search for the four groups of problems respectively.

Second, we compare our learning based method with Holte et al.'s method. For the group of 8-puzzle problems, for the average case, our learning based method is 19% less costly than Holte et al.'s method. For the group of 5-puzzle problems and the group of 7-disk Hanoi tower problems, for the average case, our learning based method is a little more efficient than Holte et al.'s method (5% and 3% less costly, respectively). For the group of 9-disk Hanoi tower problems, for the average case, our learning based method is 17% less costly than Holte et al.'s method.

Third, we compare our learning based method with Knoblock's method. Our learning based method is more applicable than Knoblock's method. Knoblock's method cannot be applied to half of the four groups of problems because the operators in these domains cannot satisfy the restrictions of Knoblock's method. For the other two groups of problems, Knoblock's method generates the same abstraction hierarchies as our learning based method does, but Knoblock's method is more efficient than our learning based method (2% and 1% less costly for average

case for the group of 7-disk Hanoi tower problems and the group of 9-disk Hanoi tower problems, respectively). This is because our learning based method need an overhead of training process which costs some resource. In fact, Knoblock's method also need an overhead of analyzing operators and calculating level values for literals, as the cost of this overhead cannot be measured by the number of explored states, it is not considered in the comparison. Therefore, if considering the overhead of Knoblock's method, the efficiency gap between our learning based method and Knoblock's method is less than 2% and 1% for the group of 7-disk Hanoi tower problems and the group of 9-disk Hanoi tower problems respectively.

Finally, we compare our learning based method with clique-based abstraction method. Our learning based method is more efficient than clique-based abstraction method for three groups of problems while less efficient for one group of problems. For the group of 8-puzzle problems, for the average case, our learning based method is 23% less costly than clique-based abstraction method. For the group of 5-puzzle problems, for the average case, our learning based method is 10% more costly than clique-based abstraction method. For the group of 7-disk Hanoi tower problems, for the average case, our learning based method is 1% less costly than clique-based abstraction method. For the group of 9-disk Hanoi tower problems, for the average case, our learning based method is 5% less costly than clique-based abstraction method.

6.4 Conclusion

In this chapter, we chose four groups of problems (the group of 8-puzzle problems, the group of 5-puzzle problems, the group of 7-disk Hanoi tower problems and the group of 9-disk Hanoi tower problems) to test five methods (depth-first search method, our learning based method, Holte et al.'s method, Knoblock's method and clique-based abstraction method). We first explained how to use attribute value pairs to represent states and operators in these four groups of problems so that our learning based method can be applied to these groups of problems. Then, we gave the test results and compared the results between our learning based method and the other four methods. From the comparison we can know that our learning based method is more efficient than depth-first search method and Holte et al.'s method. Compared with clique-based abstraction method, our learning based method is a little more efficient in three out of the four groups of problems. Finally, our method is almost as efficient as Knoblock's method but is more applicable than Knoblock's method.

Chapter 7

CONCLUSION AND FUTURE RESEARCH

In this chapter, we summarize contributions of the thesis and outline some possible future research directions.

7.1 Summary and Contributions

In this thesis, we develop models and representing tools in granular computing, and use these models and tools to design methods for solving state space search problems efficiently and effectively. The abstraction hierarchies created by our methods are clearly understandable. The experimental results show that the efficiency and applicability of our learning based method are comparable to the most advanced hierarchical state space search methods. The specific contributions of this thesis are listed as follows.

- We propose the top-down progressive computing model that provides a way of generating granular structures and solving problems by using granular structures. It enriches the literature on models and structures in the methodological perspective of the granular triangle and gives a general guidance for using granular computing for problem solving.

- We propose the concepts of attributed graphs, granulated attributed graphs and granulated attributed graph hierarchies and prove important properties about these concepts. They help us to develop the computational perspective of the granular computing triangle and create a new horizon for a wide range of applications of granular computing.

- We provide a way to represent state spaces, abstractions and abstraction hierarchies by attributed graphs, granulated attributed graphs and granulated attributed graph hierarchies, respectively. These representations are suitable for new applications of granular computing in artificial intelligence and pave the path of creating more efficient state space search methods.

- We examine two characteristics of good abstraction hierarchies that can make the hierarchical state space search efficient and effective. We propose a property, namely the inner connectivity property, for granulated attributed graph hierarchies. We prove a theorem that if a granulated attributed graph hierarchies has the inner connectivity property, the represented abstraction hierarchy has the two characteristics of good abstraction hierarchies. This theorem makes it possible to create good abstraction hierarchies by using

granulated attributed graph hierarchies, and hence makes the granular state space search possible.

- We construct the exhaustive method to create good abstraction hierarchies by generating good granulated attributed graph hierarchies, and evolve this method into a more efficient learning based method by introducing the machine learning techniques. Our learning based method can solve state space search problems more efficiently and effectively than other popular hierarchical state space search methods. Furthermore, the abstractions generated by our learning based method have semantic meanings and are easy to understand.

- We introduce a new refinement procedure, which can save previous work for backtrackings, and thus, the efficiency of state space search can be further improved.

7.2 Future Research

The learning based method generates abstraction hierarchies that have fewer backtracking and higher completeness degree, but this method does not guarantee non-backtracking and the highest completeness degree. That is, the abstraction hierarchies found by the learning based method may not be as good as those found by the exhaustive method. Future research should improve the performance of learning based method so that it can generate abstraction hierarchies as good as those generated by exhaustive method.

Another potential research is the choice of the number of levels for abstraction hierarchies. If there are only a few levels, the abstraction at every level will be big and the search in every abstraction will be costly. On the other hand, if there are many levels, the search in every abstraction may be quick, but the overhead of backtracking may be increased which affects the total costs. Future research may be dedicated to find good ways to get the best number of levels of abstraction hierarchies.

References

[1] E. Alpaydin. *Introduction to Machine Learning*. MIT Press, Cambridge, 2004.

[2] F. Bacchus and Q. Yang. The expected value of hierarchical problem solving. In *Proceedings of the National Conference on Artificial Intelligence*, pages 369–375, 1992.

[3] J.A. Baier, C. Fritz, M. Bienvenu, and S.A. McIlraith. Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In *Proceedings of the Twenty-Third National Conference on Artificial Intelligence*, pages 1509–1512, 2008.

[4] J.A. Baier and S.A. McIlraith. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the Twenty-First National Conference on Artifical Intelligence*, pages 788–795, 2006.

[5] C. Berge. *Graphs and Hypergraphs*. North-Holland, Amsterdam, 1973.

[6] W. Bibel. *Automated Theorem Proving*. Vieweg, Braunschweig, 1987.

[7] Y. Bjornsson, M. Enzenberger, R. Holte, J. Schaeffer, and P. Yap. Comparison of different grid abstractions for pathfinding on maps. *International Joint Conference on Artificial Intelligence*, 18:1511–1512, 2003.

[8] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of the Fifth European Conference on Planning*, pages 359–371, 1999.

[9] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[10] B. Bonet and H. Geffner. Mgpt: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research*, 24:933–944, 2005.

[11] A. Botea, M. Muller, and J. Schaeffer. Near optimal hierarchical pathfinding. *Journal of Game Development*, 1:7–28, 2004.

[12] V. Bulitko, N. Sturtevant, and M Kazakevich. Speeding up learning in real-time search via automatic state abstraction. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1349–1354, 2005.

[13] V. Bulitko, N. Sturtevant, J. Lu, and Yau T. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research*, 30:51–100, 2007.

[14] P.J. Cameron. *Combinatorics: Topics, Techniques, Algorithms*. Cambridge University Press, Melbourne, 1994.

[15] B. Carre. *Graphs and Networks*. Clarendon Press, Oxford, 1979.

[16] N. Caspard and B. Monjardet. Some lattices of closure systems on a finite set. *Discrete Mathematics and Theoretical Computer Science*, 6:163–190, 2004.

[17] Y.H. Chen and Y.Y. Yao. Multiview intelligent data analysis based on granular computing. In *Proceedings of 2006 IEEE International Conference on Granular Computing*, pages 281–286, 2006.

[18] Y.H. Chen and Y.Y. Yao. A multiview approach for intelligent data analysis based on data operators. *Information Sciences*, 178:1–20, 2008.

[19] T. Colburn and G. Shute. Abstraction in computer science. *Minds & Machines*, 17:169–184, 2007.

[20] G.P. Conger. The doctrine of levels. *The Journal of Philosophy*, 22:309–321, 1925.

[21] J. Euzenat. Granularity in relational formalisms - with application to time and space representation. *Computational Intelligence*, 17:703–737, 2001.

[22] A. Felner, N. Sturtevant, and J. Schaeffer. Abstraction-based heuristics with true distance computations. *Symposium on Abstraction, Reformulation and Approximation*, 9:74–81, 2009.

[23] Q.R. Feng, D.Q. Miao, J. Zhou, and Y. Cheng. A novel measure of knowledge granularity in rough sets. *International Journal of Granular Computing, Rough Sets and Intelligent Systems*, 1:233–251, 2010.

[24] R. Fikes, P. Hart, and N. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[25] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[26] M. Gupta, R.S. Rao, A. Pande, and A.K. Tripathi. Design pattern mining using state space representation of graph matching. *Advances in Computer Science and Information Technology*, 131:318–328, 2011.

[27] M. Helmert, P. Haslum, and J. Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, pages 176–183, 2007.

[28] R. Holte, M. Perez, R. Zimmer, and A. MacDonald. Hierarchical a*: Searching abstraction hierarchies efficiently. In *Proceedings of the National Conference on Artificial Intelligence*, pages 530–535, 1996.

[29] R.C. Holte and B.Y. Choueiry. Abstraction and reformulation in artificial intelligence. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 358:1197–1204, 2003.

[30] R.C. Holte, J. Grajkowski, and B. Tanner. Hierarchical heuristic search revisited. *Lecture Notes in Artificial Intelligence*, 3607:121–133, 2005.

[31] R.C. Holte, T. Mkadmi, R.M. Zimmer, and A.J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85:321–361, 1996.

[32] K. Hornsby. Temporal zooming. *Transactions in GIS*, 5:255–272, 2001.

[33] D. Joslin and J. Roach. A theoretical analysis of conjunctive-goal problems. *Artificial Intelligence*, 41:97–106, 1989.

[34] C.A. Knoblock. A theory of abstraction for hierarchical planning. In *Proceedings of the First International Workshop in Change of Representation and Inductive Bias*, pages 53–65, 1988.

[35] C.A. Knoblock. *Generating Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning*. Kluwer Academic, Boston, 1993.

[36] R. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1985.

[37] J. Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50:36–42, 2007.

[38] B. Larsen, E. Burns, W. Ruml, and R.C. Holte. Searching without a heuristic: Efficient use of abstraction. In *Proceedings of the Twenty-Fourth National Conference on Artifical Intelligence*, pages 114–120, 2010.

[39] T.J. Li and Y.L. Jing. Rough set approximations on granular structures and feature characterizations. In Y.C. Zhang, C. Alfredo, J.H. Ma, K.I. Chung, T. Arslan, and X.F. Song (eds.) *Database Theory and Application, Bio-Science and Bio-Technology*, pages 79–88. Springer, Heidelberg, 2010.

[40] J.Y. Liang and Y.H. Qian. Axiomatic approach of knowledge granulation in information system. In *Proceedings of the Nineteenth Australian Joint Conference on Artificial Intelligence*, pages 1074–1078, 2006.

[41] J.Y. Liang, Z. Shi, D. Li, and M.J. Wierman. Information entropy rough entropy and knowledge granulation in incomplete information systems. *International Journal of General Systems*, 35:641–654, 2006.

[42] R. Liang, H. Ma, and M. Huang. Pruning search space for heuristic planning through action utility analysis. *Communications in Computer and Information Science*, 201:78–86, 2011.

[43] T.Y. Lin and E. Louie. Data mining using granular computing: Fast algorithms for finding association rules. In T.Y. Lin, Y.Y. Yao, and L.A. Zadeh (eds.) *Data Mining, Rough Sets and Granular Computing*, pages 23–45. Springer, Heidelberg, 2002.

[44] D. W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, Amsterdam, 1978.

[45] G.F. Luger and W.A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison Wesley Longman, Reading, 1997.

[46] J. Luo and Y.Y. Yao. Granular state space search. In *Proceedings of the Twenty-Fourth Canadian Conference on Artificial Intelligence*, pages 285–290, 2011.

[47] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, pages 142–149, 1996.

[48] H. Moravec. *Mind Children, the Future of Robot and Human Intelligence*. Harvard University Press, Cambridge, 1988.

[49] D.S. Nau and T.C. Chang. Hierarchical representation of problem-solving knowlege in a frame-based process planning system. *Journal of Intelligent Systems*, 1:29–44, 1986.

[50] A. Newell and H.A. Simon. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, 1972.

[51] Z. Pawlak. *Rough Sets: Theoretical Aspects of Reasonging about Data*. Kluwer Academic, Dordrecht, 1991.

[52] Z. Pawlak. Granularity of knowledge, indiscernibility and rough sets. In *Proceedings of 1998 IEEE International Conference on Fuzzy Systems*, pages 106–110, 1998.

[53] Z. Pawlak, J. Grzymala-Busse, R. Slowinski, and W. Ziarko. Rough sets. *Communications of the ACM*, 38:89–95, 1995.

[54] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, 1984.

[55] W. Pedrycz. *Granular Computing: An Emerging Paradigm*. Physica-Verlag, Heidelberg, 2001.

[56] W. Pedrycz. Granular computing with shadowed sets. In *Proceedings of the Tenth International Conference on Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 23–32, 2005.

[57] W. Pedrycz. The principle of justifiable granularity and an optimization of information granularity allocation as fundamentals of granular computing. *Journal of Information Processing Systems*, 7:397–412, 2011.

[58] J.F. Peters, Z. Pawlak, and A. Skowron. A rough set approach to measuring information granules. In *Proceedings of International Conference on Computer Software and Applications*, pages 1135–1139, 2002.

[59] L. Polkowski. On fractal dimension in information systems. In M. Inuiguchi, S. Hirano, and S. Tsumoto (eds.) *Rough Set Theory and Granular Computing*, pages 79–88. Springer, Heidelberg, 2003.

[60] L. Polkowski. A model of granular computing with applications: Granules from rough inclusions in information systems. In *Proceedings of 2006 IEEE International Conference on Granular Computing*, pages 9–16, 2006.

[61] J.R. Quinlan. Induction of decision tree. *Machine Learning*, 1:81–106, 1986.

[62] I. Refanidis and I. Vlahavas. The grt planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research*, 15:115–161, 2001.

[63] L. Rokach. *Pattern Classification Using Ensemble Methods*. World Scientific, Singapore, 2010.

[64] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, 2003.

[65] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[66] E.D. Sacerdoti. *A Structure for Plans and Behavior*. American Elsevier, New York, 1977.

[67] L.P. Shiu and C.Y. Sin. Top-down, middle-out, and bottom-up processes: A cognitive perspective of teaching and learning economics. *International Review of Economics Education*, 5:60–72, 2006.

[68] H.A. Simon. The architecture of complexity. In *Proceedings of the American Philosophical Society*, pages 467–482, 1962.

[69] H.A. Simon. The organization of complex systems. In H.H. Pattee (ed.) *Hierarchy Theory: The Challenge of Complex Systems*, pages 1–27. George Braziller, New York, 1973.

[70] A. Skowron and J. Stepaniuk. Information granules: Towards foundations of granular computing. *International Journal of Intelligent Systems*, 16:57–85, 2001.

[71] A. Skowron and P. Synak. Hierarchical information maps. In *Proceedings of the Tenth International Conference on Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 622–631, 2005.

[72] G. Stapleton, J. Masthoff, J. Flower, A. Fish, and J. Southern. Automated theorem proving in euler diagram systems. *Journal of Automated Reasoning*, 39:431–470, 2007.

[73] D. Stewart. Interview with herbert simon. *Omni Magazine*, 1994.

[74] N. Sturtevant. Memory-efficient abstractions for pathfinding. In *Proceedings of the Third Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 31–36, 2007.

[75] N. Sturtevant and M. Buro. Partial pathfinding using map abstraction and refinement. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1392–1397, 2005.

[76] N. Sturtevant and M. Buro. Improving collaborative pathfinding using map abstraction. In *Proceedings of the Second Artificial Intelligence for Interactive Digital Entertainment Conference*, pages 80–85, 2006.

[77] N. Sturtevant, A. Felner, M. Barer, J. Schaeffer, and N. Burch. Memory-based heuristics for explicit state spaces. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 609–614, 2009.

[78] N. Sturtevant and R. Jansen. An analysis of map-based abstraction and refinement. *Symposium on Abstraction, Reformulation and Approximation*, 7:344–358, 2007.

[79] A. Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.

[80] J.D. Tenenberg. *Abstraction in Planning*. Ph.d. thesis, Computer Science Department, University of Rochester, 1988.

[81] D.E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

[82] J. Wing. Computational thinking. *Communications of the ACM*, 49:33–35, 2006.

[83] I.H. Witten, E. Frank, and M.A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2005.

[84] X. Wu, V. Kumar, and J.R. Quinlan. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14:1–37, 2008.

[85] F.F. Xu, Y.Y. Yao, and D.Q. Miao. Rough set approximations in formal concept analysis and knowledg spaces. In *Proceedings of the Seventeenth International Symposium of Foundations of Intelligent Systems*, pages 319–328, 2008.

[86] W.H. Xu and W.X. Zhang. Measuring roughness of generalized rough sets defined by a covering. *Fuzzy Sets and Systems*, 158:2443–2455, 2007.

[87] F. Yang, J. Culberson, R.C. Holte, U. Zahavi, and A. Felner. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research*, 32:631–662, 2008.

[88] Q. Yang. *Improving the Efficiency of Planing.* Ph.d. thesis, Computer Science Department, University of Maryland, 1989.

[89] Q. Yang. Solving the problem of hierarchical inaccuracy in planning. In *Proceedings of the Eighth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 140–145, 1990.

[90] J.T. Yao. A ten-year review of granular computing. In *Proceedings of 2007 IEEE Internationational Conference on Granular Computing*, pages 734–739, 2007.

[91] Y.Y. Yao. A partition model of granular computing. *Transactions on Rough Sets I*, 3100:232–253, 2004.

[92] Y.Y. Yao. Perspectives of graular computing. In *Proceedings of 2005 IEEE International Conference on granular computing*, pages 85–90, 2005.

[93] Y.Y. Yao. Granular computing for data mining. In *Proceedings of SPIE Conference on Data Mining, Intrusion Detection, Information Assurance, and Data Networks Security*, pages 1–12, 2006.

[94] Y.Y. Yao. A note on definability and approximations. *Transactions on Rough Sets VII*, 4400:274–282, 2007.

[95] Y.Y. Yao. Granular computing: Past, present and future. In *2008 IEEE International Conference on Granular Computing*, pages 80–85, 2008.

[96] Y.Y. Yao. A unified framework of granular computing. In W. Pedrycz, A. Skowron, and V. Kreinovich (eds.) *Handbook of Granular Computing*, pages 401–410. Wiley, West Sussex, 2008.

[97] Y.Y. Yao. Integrative levels of granularity. In A. Bargiela and W. Pedrycz (eds.) *Human-Centric Information Processing Through Granular Modelling*, pages 31–47. Springer, Heidelberg, 2009.

[98] Y.Y. Yao. Interpreting concept learning in cognitive informatics and granular computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 39:855–866, 2009.

[99] Y.Y. Yao. Human-inspired granular computing. In J.T. Yao (ed.) *Novel Developments in Granular Computing: Applications for Advanced Human Reasoning and Soft Computation*, pages 1–15. IGI, Hershey, 2010.

[100] Y.Y. Yao. Artificial intelligence perspectives on granular computing. *Granular Computing and Intelligent Systems*, 13:17–34, 2011.

[101] Y.Y. Yao and J. Luo. Top-down progressive computing. In *Proceedings of Rough Sets and Knowledge Technology*, pages 734–742, 2011.

[102] Y.Y. Yao and S.K.M. Wong. Representation, propagation and combination of uncertain information. *International Journal of General Systems*, 23:59–83, 1994.

[103] Y.Y. Yao and B.X. Yao. Covering based rough set approximations. *Information Sciences*, 200:91–107, 2012.

[104] Y.Y. Yao, N. Zhang, and D.Q. Miao. Set-theoretic approaches to granular computing. *Fundamenta Informaticae*, 115:247–264, 2012.

[105] Y.Y. Yao and L.Q. Zhao. A measurement theory view on the granularity of partitions. *Information Sciences*, 213:1–13, 2012.

[106] Y.Y. Yao and N. Zhong. Granular computing using information tables. In T.Y. Lin, Y.Y. Yao, and L.A. Zadeh (eds.) *Data Mining, Rough Sets and Granular Computing*, pages 102–124. Physica-Verlag, Heidelberg, 2002.

[107] Y.Y. Yao and N. Zhong. Granular computing. *Wiley Encyclopedia of Computer Science and Engineering*, 2007.

[108] Y. Ye and J.K. Tsotsos. Knowledge granularity and action selection. In F. Giunchiglia (ed.) *Artificial Intelligence: Methodology, Systems, and Applications*, pages 475–488. Springer, Heidelberg, 1998.

[109] L.A. Zadeh. Towards a theory of fuzzy information granulation and its centrality in human reasoning and fuzzy logic. *Fuzzy Sets and Systems*, 90:111–127, 1997.

[110] B. Zhang and L. Zhang. *Theory and Applications of Problem Solving*. North-Holland, Amsterdam, 1992.

[111] M.Q. Zhao, Q. Yang, and D.Z. Gao. Axiomatic definition of knowledge granularity and its construction method. In *Proceedings of the Third International Conference on Rough Sets and Knowledge Technology*, pages 348–354, 2008.

[112] Y. Zhao, Y.Y. Yao, and J.T. Yao. Level-wise construction of decision trees for classification. *International Journal of Software Engineering and Knowledge Engineering*, 16:103–126, 2006.

[113] R. Zhou and E.A. Hansen. Space-efficient memory-based heuristics. In *Proceedings of the Nineteenth National Conference on Artifical Intelligence*, pages 677–682, 2004.

[114] J.D. Zucker. A grounded theory of abstraction in artificial intelligence. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 358:1293–1309, 2003.