USING EXPLANATION STRUCTURES TO SPEED UP LOCAL-SEARCH-BASED PLANNING

TIAN ZHENGMIAO

(M.Eng), NUS

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2012

Acknowledgements

First and foremost I offer my sincerest gratitude to my supervisor, Dr Nareyek Alexander, who has supported me throughout my thesis with his patience and knowledge whilst allowing me room to work in my own way. I appreciate all his contributions of time, ideas and funding, and without him this thesis would not have been completed and written. The joy and enthusiasm he has for his research was contagious and motivational for me. I am also thankful for the excellent experience of doing research under his friendly supervision.

The members of the planning group have contributed immensely to my personal and professional time at NUS. The group has been a source of friendships as well as good advice and collaboration. I am especially grateful for Amit Kumar. I learnt much from him both in research and implementation skills. He also helped me a lot during the last stages of this thesis in terms of implementation and testing on Crackpot. We worked together in II-Lab, and I very much appreciated his enthusiasm, intensity, willingness and hard work in researching AI planning. Other group members that I have the pleasure to work with or alongside of are: Manni Huang, Solomon See, Eric Vidal, and the numerous FYP students who have come through the lab.

My time at NUS was made enjoyable in large part due to the many friends and groups that became a part of my life. I am grateful for the camaraderie and time spent with my friends, for the couple Cao Le and Wang Yang who have supported me a lot since I came to Singapore, for Zou Xiaodan who was always ready for help, for graduate students Ye Xiuzhu and Liu Xiaomin by who my life in NUS was greatly

enriched, for graduate students Zhang Xiaoyang and Tan Jun in Signal Processing and VLSI Lab, and for other people and memories.

The Department of ECE has provided the support and equipment I have needed to produce and complete my thesis.

Finally, I would like to thank my family for their love, encouragement and support. Most of all for my loving, encouraging and patient husband Wang Huan, whose faithful support during the final stages of this thesis writing up is so appreciated. Thank you!

Tian Zhengmiao
NUS
September 2011

Table of Contents

Summary

Many examples of real-world autonomous agent applications can be found nowadays, from exploring space to cleaning floors. AI planning is a technique that is often used by autonomous agents, i.e., planning is a problem-solving task to produce a plan, which can then be performed by an autonomous agent. For example, when given a goal of "package should be in city1", a planning system utilizes possible actions, like "move truck from between two cities", "load/unload package to/from truck", etc., to generate a plan that is composed of a set of these actions to achieve the goal. This thesis focuses on dealing with planning systems that have loose plan structures designed to solve large-scale real-world problems. Loose plan structure involves actions in the plan that have no explicitly represented relations, like indicating that one action is added for achieving another. A lack of such causal information might result in an inefficient planning process.

The goal of this thesis is to speed up planning systems that have loose plan structures using local search approaches to create plans. To address the potential inefficiencies, we propose a novel technique that uses explanation structures to retain some causal information acquired during planning. To improve the planning performance by utilizing explanation structures, we generate Multiple-In-Single-Out (MISO) causal networks, and develop algorithms to update and exploit these structures, in order to dynamically generate macro-actions and operate on them.

To evaluate the proposed approach, we implemented a prototype based on a planning system named Crackpot. Our approach is promising to improve the planning performance by the usage of macro-actions.

List of Tables

List of Figures

Chapter 1 Introduction

Using AI techniques to solve real-world problems is pervasive in our real life. *Planning* is a particularly important AI technique for problem-solving, i.e., it is to come up with a set of actions that will achieve a given goal; this set of actions is known as *plan* [1]. For example, "gotoKitchen", "takeApple" and "eatApple", and so on, are the set of actions that can be added into the plan, achieving the goal that the player should not be hungry (as shown in Figure 1).

To find a plan, search methods are necessary, and are closely related to the planning performance (it can be on planning speed or plan quality). Local search methods are not new techniques used in planning to quickly find a plan, such as their usages in LPG [2] and Excalibur [3]. Planning using local search methods iteratively repairs the current plan to get a better successor plan until all inconsistencies are solved or its stopping criteria are satisfied. The plans are evaluated by an objective function.

Some local-search-based planning systems (they are also called "planners") that have loose plan structures are designed to solve large-scale and complex real-world problems [4] (systematic search might be not applicable for those problems). Loose plan structure involves actions in the plan that have no explicitly represented relations, like indicating that one action is added for achieving another. However, a lack of such causal information might result in an inefficient planning process. Although lots of local-search-based heuristics are developed to improve the planning performance, like heuristic using randomization to jump out of local minima, there is still a lot of room for improvement. Let's look at a concrete planning example as shown in Figure 1 for a better understanding of this problem.

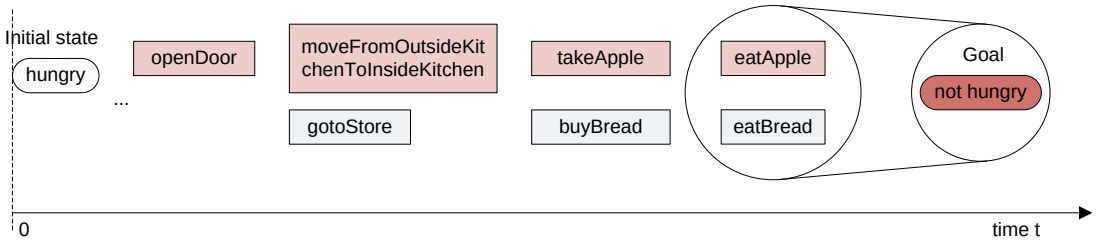


Figure 1: An Apple Domain Example

In the example, the goal given to the planner is that a player should not be hungry (initially he is hungry). To achieve the goal, the planner generates a plan that is composed of a set of actions: "openDoor", "moveFromOutsideKitchenToInside-Kitchen", "takeApple" and "eatApple, i.e., the set of actions that are shadowed in red color in Figure 1. Temporal constraints can be enforced on these actions such that they can be executed in a given order. However, the agents that execute the plan don't know why these actions are added into the plan, since a set of actions need not be causally connected.

If the problem domain is complete, static and very simple, the planning process can be very easy. However, lots of real-world domains don't have those features. They might be dynamic or open - that is, the domain information can be modified during the planning process. One of the scenarios is that, in a multi-agent domain, after the planner generates the above plan for an agent, the other agents might change the environment before the plan is completely executed. For example, another player might suddenly lock the door of the kitchen and destroy the key. Then the previous plan for the first player becomes infeasible, because the action "openDoor" is infeasible without the key. The state of "Having key" is a precondition of "openDoor", and it is currently inconsistent. These infeasible or useless actions will reduce the plan quality. Thus, the planner needs to repair the current plan in order to successfully achieve the given goal again.

There are two ways that can be used to repair plans: removing actions that have inconsistencies and adding new actions to resolve existing actions' inconsistencies. Due to the loose plan structure in the above example, a problem hinders the repairing process; the planner doesn't know which action is added for what purpose. Thus, the planner uses a greedy and potentially inefficient way that iteratively removes the action that has the most significant inefficiency from the current plan, or adds a new action to resolve the inconsistency. Four iterations are needed for removing the four actions from the previous plan and three more iterations are needed for adding three new actions: "gotoStore", "buyBread" and "eatBread" into the current plan. Lots of computation time is needed in every iteration to make a greedy choice to repair the current plan.

The above inefficient behavior and the time costs might be acceptable in some applications that have soft requirements. However, taking mobile electrical devices as an example, they might not be able to have processor as fast as desktop computers. Thus, there is a need to improve the planning performance of these planners.

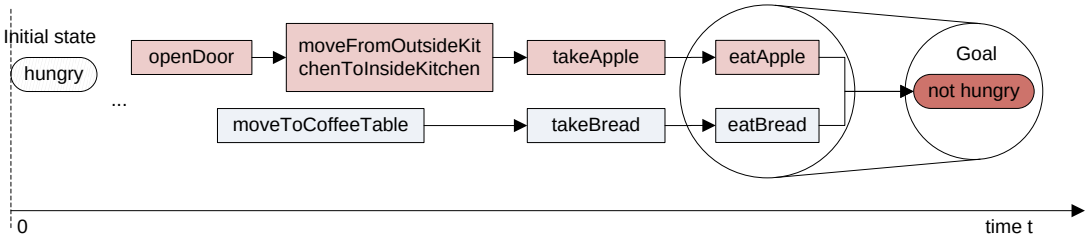


Figure 2: An Apple Domain Example with Explanations

Let's have a look at what the planner can do if the causal information in the plan is straightforward (refer to the explicit representation of the causal relations between actions in Figure 2). When the action "openDoor" in the current plan become infeasible, the planner can conclude that the sequence of actions: "moveFromOutside-KitchenToInsideKitchen", "takeApple" and "eatApple" are all required to achieve the given goal, by forwards exploiting the explicit causal information. Thus, once

"openDoor" is infeasible, these three actions will also become infeasible. Therefore it is reasonable for the planner to consider removing all of them in one iteration. The time cost of exploiting such straightforward causal information promises to be less than the time cost of searching for successor plans and analyzing them in the four iterations (in each of which one action is removed). Thus, the planning process can be made potentially more efficient by utilizing causal information, and the usage allows AI algorithms to be used in real-time on low power processor.

1.1 Goals of the Thesis

The goal of this thesis is to speed up planners that have loose plan structures using local search approaches to create plans.

As mentioned above, the planners can quickly have a further view for searching for better and more reasonable successor plans, by using the straightforward causal information. Thus, to address the potential inefficiencies of using loose plan structures, we propose a novel technique using "explanation structures" to retain some causal information that is acquired during planning, generate a type of causal networks that is named Multiple-In-Single-Out (MISO) in this thesis and develop two associated algorithms. The first algorithm updates the MISO causal network whenever the plan is changed while the second algorithm exploits the MISO causal network to yield more reasonable and more significant change that can better improve the plan. Detailed contents of this research will be introduced in Chapter 3 and some implementation related issues will be introduced in Chapter 4. In the next section, we will clarify the methodology of doing this research.

1.2 Methodology

This thesis consists of 5 chapters.

In Chapter 1, we first described our motivations of doing research in the field of AI planning, and using causal explanation to improve planning performance. Next, we declared the goal of this research, and proposed a novel approach to achieve the goal. Finally, the methodology of doing research is introduced in this section. The proposed approach for improving the planning performance can be divided into four parts:

1) Designing causal explanation structure;
2) Developing an algorithm updating a type of causal networks named MISO;
3) Developing an algorithm for exploiting MISO causal network;
4) Implementation of the above three parts in Crackpot and evaluation.

To achieve the above four points, the knowledge of AI planning, algorithms, search paradigms, explanation concepts are essential. In addition, the other lessons that are in the field of AI, algorithm and statistics knowledge are also contributive to the thesis. Learning statistics knowledge is for the purpose of using probability knowledge in the above two algorithms and for presenting the evaluation results.

The background, related work and some analysis of plan structures are presented in Chapter 2. For the purpose of enhancing the loose plan structures, features of the loose plan structures are analyzed and generalized. Next, to get inspiration from other robust plan structures, different kinds of planning paradigms are reviewed and four of them that have commonly used plan structures are analyzed in this research. However, in order to analyze planning paradigms one should have a background of AI planning and search paradigm. Thus, this background needs to be acquired as a foundation of the analysis mentioned above.

After getting the essential background, the detailed research and the corresponding implementation are carried out in Chapter 3 and Chapter 4,

respectively. Enhancing the loose plan structures by using causal information also increases the planners' time and memory costs. Therefore, we have to find a trade-off between the extra costs and the cost reduction due to the causal information. To find the balance point, we first did a case study and characterized some cases in terms of when to explain causal information. Next, in terms of the way of keeping the causal information, four types of causal networks are analyzed. The current research is focused on one type of causal networks named Multiple-In-Single-Out (MISO), because MISO appears to be relatively more reasonable and less costly than the other types. Since the plan structures will be updated after the plan changes, an updating MISO algorithm is essential. However, enhancing the loose plan structures is not our ultimate purpose. Instead, the aim is to speed-up planning by utilizing the enhanced plan structures. Thus, an exploiting algorithm is necessary.

The planning system named Crackpot is chosen as the base system to test the performance of this proposed approach. However, at the beginning of my candidature, Crackpot was not a completed system. Thus, both constructing Crackpot and implementing our approach on Crackpot were important. Since the current research is focused on the causal information that is related to symbolic attributes, my work of implementing Crackpot is related to symbolic attributes (refer to Chapter 4). Besides, evaluating the approach by using two types of domain problems is also one of the experimental settings. Thus, a background of domain representation and various planning domains should also be a part of the literature review. The contents in Chapter 4 includes an overview of crackpot architecture, its planning process with the two algorithms highlighted, designed models of explanation structures in Crackpot, etc.

Finally, a conclusion is drawn and possible future work of the research is listed in Chapter 5.

Chapter 2 Background and Literature Analysis

To obtain an understanding of the problem of local-search-based planning, it is necessary to have a general background of both planning and local search. This background will be introduced in Section 2.1. Furthermore, as mentioned in the previous chapter, our research is focused on the local-search-based planners that have loose plan structures. When repairing plans, those planners might make trivial or wrong decisions on choosing plan successors with a lack of straightforward connections between actions. These decisions will slow down the planning. As analyzed in Section 1.2, enhancing the loose plan structures promises to reduce the time costs. However, this reduction is not simply proportional to the amount of this information that is used to enhance the plan structures. The issues on what information is useful and how to represent the information is beneficial are also important. To address these two issues, we will give some analyses on four planning paradigms that have commonly used robust plan structures, after introducing the above general background. These analyses are focused on the plan structures (refer to textbook [1,5] for more details if interested). Next, because we use a concept "explanation" to store some causal information in this research, we will also give an introduction on some other explanation concepts used in other areas and their respective usages in Section 2.2. Our explanation usage is to reasonably group some of the actions together dynamically during the planning. The usage is similar to the concept of macro-action in the field of AI planning. For this sake, we also make some analysis on macro-action in the last part of this chapter.

2.1 AI Planning Introduction

Planning is a vast field and a key area in AI. There are many practical planning applications in industry. A few examples are design and manufacturing planning, military operations planning, games, space exploration, web service composition and workflow construction on a computational grid. Planning techniques are introduced in progressively more ambitious systems over a long period, such as local search techniques [5].

Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems[1]. To do good research, it is worthwhile to take a look at how AI planning researchers conduct their research on AI planning as well as the achievements and performance of their approaches.

In AI planning, typically there are two main ways to improve the planning performance: proper plan representations with respect to different kinds of planning applications, and search algorithms which can take advantage of the representation of the planning problems. We deal with local-search-based issues in this research. Nonetheless, both planning and search algorithms are huge topics. This thesis present will not cover all planning systems and searching techniques. Instead, we first give readers a sense about what kinds of domain problems we are interested to solve and what features those problems have. After that, with respect to those features, we explain why local-search-based planners that have loose plan structure are dominant in solving above problems. However, potentially inefficient and unintelligent search is one of disadvantages of using loose plan structure. To have a good understanding of how other planners benefit from using robust plan structures and what might be helpful for our research, we analyze some commonly used planning paradigms in terms of their plan structures. On the other hand, search techniques play an important

role in planning, and they are inseparable from plan structures. Thus, we briefly analyze two search paradigms in advance of analyzing planning paradigms.

2.1.1 Properties of Real-World Environments

With respect to some properties of the planning problem environment, planning that deals with fully observable, deterministic, finite, static, and discrete problems with restricted goals and implicit time is classified as classical planning [1]. Furthermore, classical planning is also *offline* planning regardless of the current dynamics, if any, during the planning process.

In contrast, most of real-world environments are so complex that they have properties like: partially observable, non-deterministic, sequential, continuous, dynamic, and multi-agent (an agent is the one that can perceive the environment through sensors and act upon the environment through actuators [1], like a robot). For practical purposes, *online planning* sometimes is needed for real-world planning problems. "Online" indicates that plan making and execution are interleaved, in order to handle changes in state of an environment, which is typical for real-world scenarios.

Thus, the properties of real-world environment are completely different from that of the classical planning problem. The comparisons are listed as follows:

- *Partially observable*. It means the entire state of the environment is not fully visible to an external sensor. For example, in a multi-agent environment, an agent cannot see what actions other agents perform that might change the whole environment. While in classical planning, the complete state of the environment is known at each point in time.
- *Non-deterministic*. If the outcome by executing an action on a state is always the same, the environment is deterministic, otherwise the

environment is non-deterministic. In a complex and competitive environment, it is usually not practical to keep track of all aspects all the time. When the environment is partially observable it could appear to be non-deterministic. For example, taxi driving is non-deterministic because the traffic situation can be unpredictable. In contrast, in a deterministic environment of two rooms that can be cleaned by a robot, if the robot is currently in a room, it can enact "Clean" and the room will always be clean after that.

- *Sequential*. It means the current decision could affect all future decisions, like taxi driving environment. Otherwise, the environment is episodic, like the above cleaning robot environment. Many classical problems are episodic.
- *Dynamic or semidynamic*. If an environment changes over time when the agent is deliberating (or we can say "planning"), then it is dynamic, otherwise, it is static. Taxi driving can be dynamic or semidynamic (that is assuming the environment doesn't change, but the taxi driver will get penalty if the car doesn't move).
- *Continuous*. In the real world, actions always have explicit durations, or goals are to be achieved before a time slot according to a temporal constraint, thus explicit time is necessary. While in classical planning implicit time is used.
- *Multi*-agent. For example, in taxi driving problem, there are multiple drivers in the whole environment, who can affect each other.
- *Extended goals*. In the real world, not only the final goal but also the states traversed are concerned. The form is to set some constraints on the

trajectories of planning. For example, in Logistics-type problems, a truck is required to accommodate at most one package at one time.

- *Infinite*. Resources, like food, in a real world can be consumed or produced and this will cause the environment to have infinite states. For example, a state can be "the i^{th} bread exists" or "the i^{th} bread doesn't exist". Breads keep being consumed and produced. Thus, the quantity of the breads will be infinite and result in the fact that these bread-related states are infinite.

The diversity of the real world problems which have combinational properties with some or all of above properties causes the difficulty of planning. Our explanation-based approach works for planners that have a subset of the features listed above, except the *Non-deterministic* feature. A great deal of research targeted at solving real-world problems has been done on planning, including research on search techniques for good planning performance, like finding solution plans faster. We will analyze search paradigms in planning in the next subsection.

2.1.2 Search Paradigms in Planning

To quickly find a solution, two search paradigms are commonly used in AI planning: refinement search and local search, as highlighted in Figure 3.

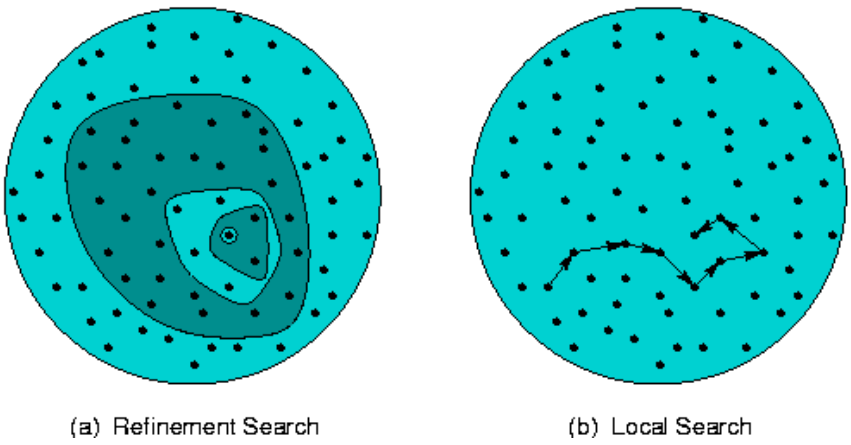


Figure 3: Search Paradigms (taken from [3])

2.1.2.1 Refinement Search

Refinement search is also called *split-and-prune* search [6]. Subbarao Kambhampati pointed out that "Almost all classical planners that search in the space of plans use refinement search to navigate the space of ground operator sequences and find a solution for the given planning problem" in his technical report in 1993[7]. "Ground operator" is called "action" in some other planners.

A refinement based planner starts with a partial plan and repeatedly adds details to the partial plans until all constraints (plan candidate set are implicitly represented as a generalized constraint set) are satisfied. Each time, by adding more details, the search space can be split into two parts as shown in Figure 3. One part of the plan space is to be pruned, in which each of the plan candidates is inconsistent with some constraints. The split-and-prune process is to be repeatedly done on the remaining plan space until a solution candidate (also can be called a solution plan) can be extracted from the remaining plan space in a bounded time. Note that in the refinement process, backtracking is sometimes necessary.

Traditionally, refinement techniques apply a *complete* search. As compared to exhaustively systematic search, refinement search ensures much greater planning efficiency by repeatedly eliminating large part of plan search space that is provably irrelevant. *Total-order*, *partial-order* and *hierarchical* planning are typical instances of refinement-search-based planning (refer to[5] for more details of these three planners). In these planners, their plan structures contain some information that ensures the backtracking. We will later give more detailed analysis on these planning paradigms.

Nonetheless, it is usually not feasible to consider the whole search space for a variety of real-world problems. With regard to such "Infinite" property of the problem

environment, techniques which stop the search at some point become necessary, such as local search techniques. Local search is to be analyzed in the next subsection.

2.1.2.2 Local Search

A *Local search* method starts from a candidate solution and iteratively moves to another solution in its neighborhood in the space of candidate solution, until an optimal solution is found or a time bound is elapsed. The solutions that the current candidate solution can move to are called *neighbor*s of the current candidate solution. For a local-search-based planner, any partial plan can be a candidate solution, and the operation of updating the current plan with another plan is called *repairing*.

Typically, every partial plan has more than one neighbor. Thus, quality evaluation on plans which are in the neighborhood of the current plan is necessary in order to find an optimal plan. Plan quality evaluation can be done by an *objective function*.

Local search algorithms have been used to improve planning efficiency in a somewhat indirect way [8]. For example, in every iteration local search methods typically estimate only some (not all) of plans in the neighborhoods in a bound time and heuristically move to one/some of evaluated plans. Thus, a local search algorithm is typically incomplete.

On the other hand, unlike systematic search algorithms, which need to keep a large amount of explored plans together with searching histories because of backtracking if necessary, a typical local search algorithm stores only the current plan and doesn't retain the trajectories of searching history. Thus it has low memory requirements. The needed memory is O(1) level to the plan space.

Local search methods have found application in many domains. A well-known Walksat procedure is for solving SAT problems [2][8][9]. Inspired by Walksat, LPG uses stochastic local search procedure Walkplan[2] for solving planning graphs. GSAT was introduced by Selman, Levesque & Mitchell (1992), which solves hard satisfiability problems using local search where the repairs consist of changing the truth value of a randomly chosen variable. The cost function in GSAT is the number of clauses satisfied by current truth assignment [8]. Excalibur (Nareyek, 1998)[3] uses local search to facilitate an uncomplicated and quick handling the environment's dynamics with interleaved sensing, planning and execution.

2.1.2.3 Discussion

Some comparison between systematic search (like refinement search) and local search on several aspects are listed as follows:

- *Speed and complexity*. Compared to systematic search, which takes prohibitively long and uses large amount of resources, local search reveals its advantages in complexity of both planning speed and memory requirement. The use of local search has become very popular for tackling complex real-world optimization problems; complete search methods are still not powerful enough for solving these kinds of problems, because the search space of real-world domains is combinatorial in nature. For example, systematic search methods are computationally costly in problems that use large number of actions or objects, constraints by time and resources, and so on [1]. Furthermore, supported by various local-search-based heuristics well developed in the past twenty years, local search algorithms can often find reasonable solutions in large or infinite (continuous) space.

- *Optimal plan*. If a problem has a solution, there is no guarantee that the optimal solution will always be found by using local search, because the search is incomplete. However, it is guaranteed by systematic search algorithms, like refinement search.
- *Proving unsatisfiability*. In cases where no solution is found, local search is unable to prove the unsatisfiability, while refinement search algorithms will return a failure in this case after exploring the whole search space.
- *Anytime planning*. Anytime planning is another advantage of local-search-based planners. It means that the planner can output a plan at anytime even though the plan quality might be not optimal. In the real world, an anytime solution is sometimes needed. Refinement search terminates either with a ground plan or a failure, and a plan is found when one branch is exploited completely.

In a word, for large combinatorial problems [10] including complex structures, dynamic changes and anytime computations [4,11], local search methods have been effectively used. Thus, we take local-search-based planning as our research object.

In recent years, several meta-heuristics have been proposed to extend local search in various ways [12].

A tricky issue in the context of real-world problems is that some space usually contains many local minima which cause difficulty for local search algorithms to get a global optimal plan. To escape from local minima, various researches on heuristics have been undertaken and good results have been achieved by incorporation of randomness, multiple simultaneous searches, and other improvements. In recent years, to address the problem of jumping out of local minima, there has been a great deal of research and experimentation to find a good balance between greediness and

randomness [1]. For example, after evaluating some neighbors in a bound time, if some better neighbors can be found, then a local search algorithm can heuristically move to one of them. Otherwise it randomly moves to one of neighbors with a probability.

Some advanced algorithms, such as Variable-depth search, simulated annealing and tabu search, were used to minimize the probability of being stuck in a low-quality optimum (local minimum) [12]. Variable-depth search is based on applying a sequence of steps as opposed to only one step at each of iteration. When a worse neighbor is chosen, simulated annealing selects it with some probability which is decreased over time analogous to physical temperature annealing. Simulated annealing guarantees that it converges asymptotically to the optimal solution, but it requires exponential time.

Another issue is that local search might repeatedly explore one/some of explored plans because of the fact that local search doesn't retain the search history, and it searches locally. Ideas like using tabu-list to retain the last *k* visited plans are used to address the issue. Empirical studies showed that tabu search can help improve the planning performance (the size the neighborhood can be decreased and searching can be speed up). It can also consider a solution of higher cost if it lies in an unexplored part of the space.

Inspired by tabu search mentioned above, appropriately retaining some useful information during search can accelerate search. In this thesis, we propose a novel approach that uses explanations structures to retain some other useful plan information and use them to accelerate planning. The detailed case study and research will be introduced in Chapter 3.

2.1.3 Analyzing Plan Structures

As of now, we have a basic understanding of features of real-world planning problems and two search paradigms commonly used in AI planning. In this subsection, we will first analyze general features of loose plan structures. To get an inspiration of how to enhance loose plan structures from others, we will analyze four planning paradigms that have commonly used plan structures.

2.1.3.1 Features of Loose Plan Structure

Excalibur is a planning system that uses loose plan structure for solving real-world planning problems. Figure 4 illustrates a plan example in Excalibur.

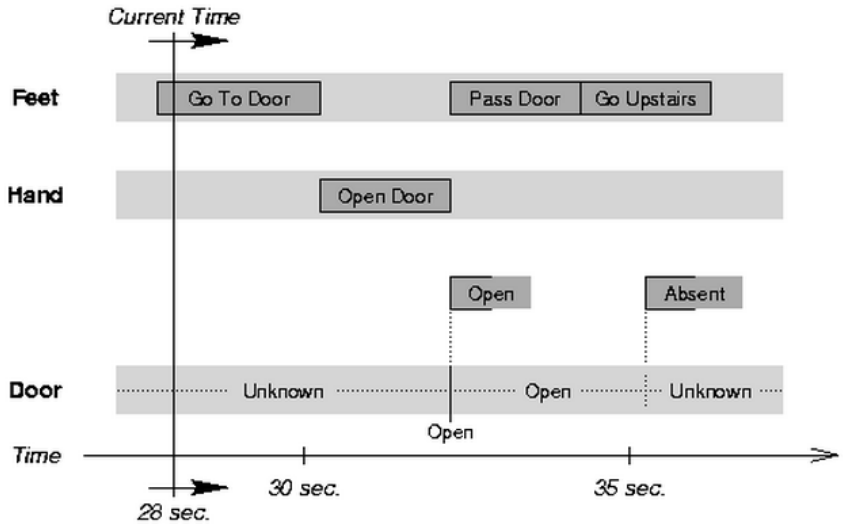


Figure 4: A Plan Example in Excalibur (taken from [3])

Excalibur uses explicit time representation, i.e., actions have start times and durations. Actions are projected by the timeline but they are not explicitly connected. As can be seen from the example, the action “Open Door” makes another action “Pass Door” possible to occur. However the causal relation between those two actions is not explicitly represented; it can only be acquired in a non-straightforward way that analyzes the state projection of the door. Searching for causally relevant actions in this kind of plan structure is inefficient.

Plan structures that have the following basic features are defined as ***loose plan structures*** in this thesis:

1) All Actions in the plan are temporally ordered;
2) There is no explicit connection between actions in plan structure. Supposing A is a set of actions, an explicit connection p is a tuple $<a_i, a_j>$ where a_i, and $a_j \in A$. A data structures that can be directly translated to a p, is also regarded as an "Explicit connection" between actions. For example, causal links in POP [5] and a hierarchical relationship between a high-level action and a low-level action in HTN planner [1] are some forms of explicit connection;
3) An action has preconditions and effects.
4) There should be a specific representation of preconditions and effects that can be used to easily analyze a causal relation between a precondition and an effect. For example, a variable "Whether John owns an apple" has only two possible states: "John has an apple" or "John doesn't have an apple". Thus, the variable can be formally represented as a Boolean attribute variable "John.hasApple" that has "true" or "false" value referring to above two states respectively. Besides, the order between preconditions and effects is also important for analyzing the causal relation because it is impossible that a precondition has causal relation between effects that occur after it. Suppose the time representation is used to address the ordering problem, a state can be represented as "John.hasApple == true @ t_1". Using this representation, the causal relation between an effect and a precondition can be analyzed in terms of the following three requirements: they have the same value on the same variable and they belong to different

actions; secondly, suppose the effect and the precondition are states at time t_1 and t_2 respectively, then "$t_1 < t_2$" should be satisfied; finally, there is no action that occurs during time t_1 and t_2 and changes the value of the effect.

Planners that have robust plan structures where actions are explicitly connected can be converted to the above general loose plan structures, but not vice versa. If during the converting no p is dropped, then the planner can also be regarded as having loose plan structure. For example, Excalibur system has feature 1), 2) and 4), but it uses concepts of "condition" and "contribution" instead of "precondition" and "effect". Similar to a precondition, a condition is related to states of an attribute variable. But a contribution is a state transformation behavior to an attribute variable. An effect is a result of a contribution in Excalibur system. Thus, Excalibur can be regarded as this type of planning system.

2.1.3.2 Total Ordered Planning

Early planning systems constructed plans in a total order [3]. The total-ordered planning paradigm originated from the earliest planning system, STRIPS [13], is roughly synonymous with the notion of "classical planning" as described in subsection 2.1.1.

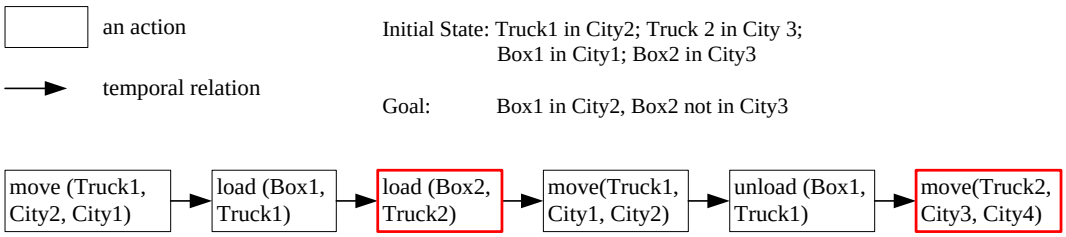


Figure 5: A Total Ordered Plan Example

A total ordered planning paradigm searches in a state space. State transition can be achieved by actions. A *plan* in a total ordered planning system is defined as a sequence of actions corresponding to a path from initial state to goal state. Figure 5

shows an example of a plan in total ordered planning. All actions in the plan are ordered by temporal constraints.

Note that all states along the path are explicit. Although early state-space search algorithms work in low efficiency due to a lack of good techniques to guide the search, the state-of-the-art state-space planners have been able to significantly benefit from this "explicit" feature by making very efficient use of domain-specific heuristics and control knowledge (refer to Bonet and Geffner's Heuristic Search Planner (HSP) [14] and its later derivatives for more details if interested). This makes state-space planning capable of scaling up to very large problems and quickly generating plans which are optimal or near optimal in length [5]. Besides, strong domain-independent heuristics can also be derived automatically by defining a relaxed problem which is easier to solve (if interested more details can refer to [1], section 10.3). Furthermore, other techniques, such as Goal-Oriented Action Planning architecture (GOAP [15]), enable total ordered planning to handle a restricted open world (online planning) problem by adding some extensions to classical STRIPS. With those extensions, GOAP can handle partial observability, non-determinism and extended goals. However, these are not intrinsic advantages of total ordered plan structure, i.e., they are ensured by extra domain-specific information, or by relaxing problems, or by adding extensions to domain representations.

Moreover, there are still some restrictions of classical planning that haven't been addressed yet, such as implicit time (actions which have no duration), sequence plan, and finite state space. Furthermore, total ordered planning is still not capable of handling multi-agent problem environments, because multiple objects will cause the state space to increase exponentially and make planning very slow. For example, "Truck1" in Figure 5 is an object that has the capability of moving between two cities

and load/unload boxes. In the example, the state space increases exponentially with the increasing amount of trucks.

Another disadvantage of total ordered planning is that its representation makes it impossible to produce an optimal plan for the case that some sub-problems are independent and sub-plans are allowed to run in parallel. For example, the two sub-problems in Figure 5 are independent. They can be solved by a sequence of actions that are operating on one of two trucks. The total ordered planner needs two subsequences of actions (each of them is to move one specific package) to run in sequence. The optimal plan in this example is that these two sub-plans run in parallel (Figure 6).

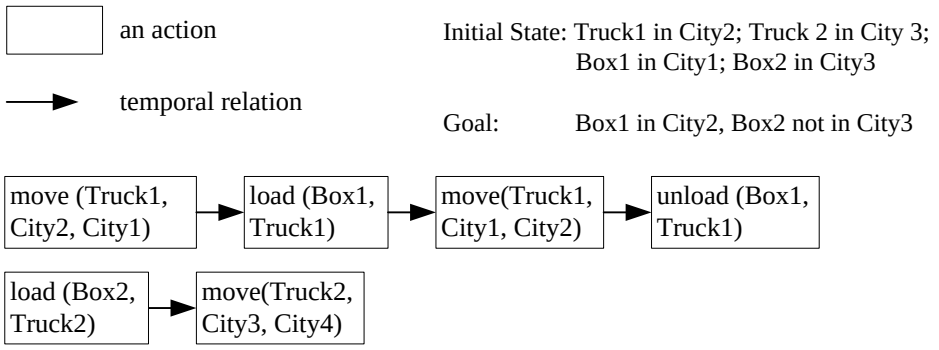


Figure 6: An Optimal Plan Example Runs in Parallel

Furthermore, re-planning is costly in the dynamic domain. For example, in Figure 5, if the sub-problem that "Box2 not in City3" is changed or removed from the domain, the planner cannot get information that "load(Box2, Truck2)" and "move(Truck2, City3, City4)", highlighted with red border line in Figure 5, are the subsequence of actions required to solve the sub-problem from its plan structure. The planner needs to do lots of backtracking with respect to the change in domain.

Therefore, temporal constraints between actions in total-ordered plan structure are not helpful for domain problems that have real-world environment features.

2.1.3.3 Partial Ordered Planning

Partial ordered planning (POP) searches through plan space. In POP paradigm, a *plan* is composed of exactly four ingredients: a set of actions, ordering constraints, causal links and variable binding constraints [5]. Actions in a plan can be partially or totally ordered. Thus, as compared to planning in state-space, POP has more general and looser plan structures. Some famous POP planners are UCPOP [16] and RePOP [17].

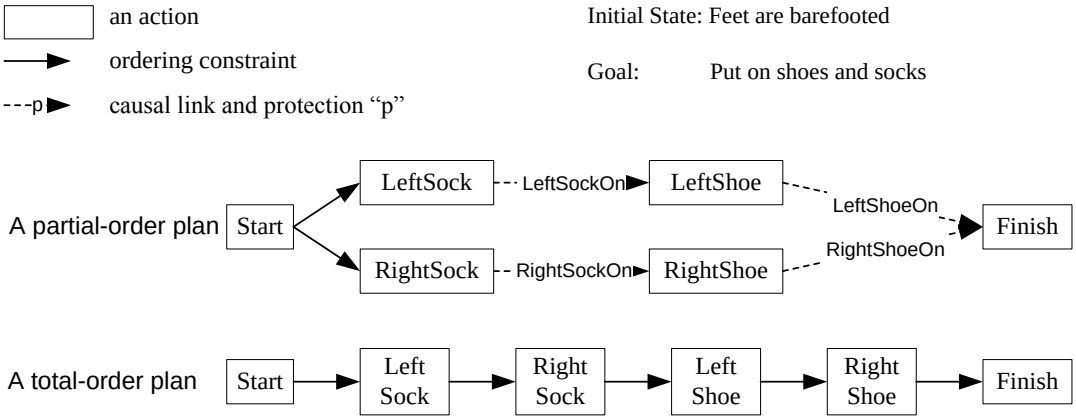


Figure 7: A POP Plan for Put on Shoes and Socks Problem (Figure is evolved from [1])

Figure 7 illustrates a POP plan example for "put on shoes and socks" problem [1]. The total ordered plan in the example is one of six total-order plans which can be generated by linearizing the partial-order plan. The linearization cannot violate ordering constraints and causal links in the partial-order plan. POP starts with an empty plan consisting of the initial state and goals and uses refinement search to find a plan solution. One key point of POP is the usage of causal links in the plan.

A *causal link* is added when establishing an open condition (that is, an unsatisfied goal/precondition) by adding an action into the plan. It links between two actions, stating that one precondition of the latter action is achieved by the former action. For example, "leftSockOn" is not only precondition of action "LeftShoe" but also effect of action "LeftSock". The precondition mentioned above is a protection

that cannot be negated when adding a new action between the two linked actions. Thus, a causal link has more meaning than an ordering/temporal constraint. It not only has an implicit order between two actions but also keeps the rationale of the order [5,18]. A partial-order plan may have ordering constraints without causal links, but not vice versa. If action A and B are linked by an ordering constraint, it indicates that A should be executed before B, but not "immediately before" B. Causal structures contain vital information that is obscured by classical STRIPS representation [19] in state-space planning paradigms.

Compared to state space planners, plan-space planners such as POP have the following advantages:

- They contain fewer constraints on partial plan.
- They keep all the advantages of refinement search, like high efficiency and great reduction of the overall size of search space. (But the refinement cost increases concurrently, which makes re-planning very slow.)
- Plan structures are more general. Different types of plan-merging techniques can be easily defined and handled because of partial plan structures. This feature ensures POP can handle multi-agent planning.
- More expressive and flexible. Because of causal links, the rationale for plan's components is explicit and easy to understand.
- They can handle some extensions to classical planning, such as time, resources using temporal and resource constraints.

On the other hand, some excellent domain-specific heuristics improve planning efficiency in state space, but they reveal low efficiency in plan space, because plan-space planners such as POP represent implicit states. Furthermore, the search space is more complex in the plan space than in states [5]. Thus, as of now

POP planners are not competitive enough in classical planning with respect to computational efficiency, and there is no related work of adding real-time extensions to POP.

2.1.3.4 HTN Planning

Hierarchical decomposition is one of the most pervasive ways for dealing with complexity. A planning method based on hierarchical task networks (HTNs) is called HTN planning, in which the plan is refined iteratively by applying action decompositions. The process of HTN planning can be viewed as iteratively replacing abstract actions by less abstract actions or concrete actions [1]. Thus, HTN planning is based on refinement search on plan space. The main difference between HTN planning and POP is the primary refinement techniques they use: POP planners use establishment refinement, while HTN planners use task reduction refinement.

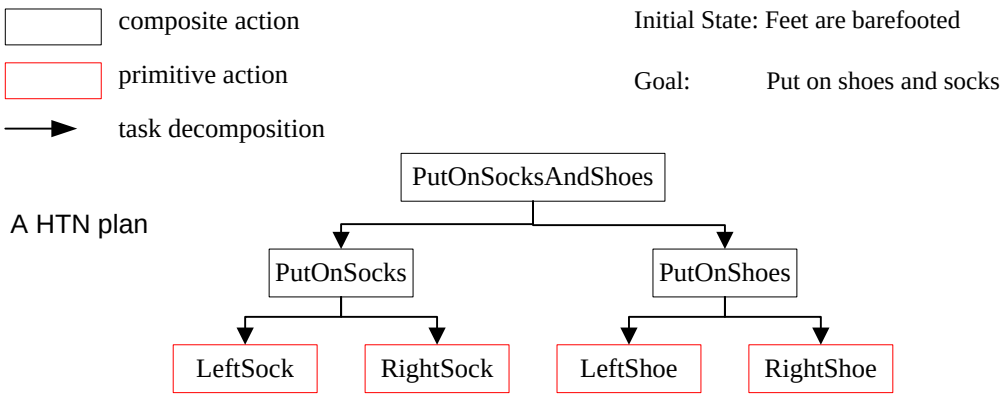


Figure 8: A HTN Plan for Shoes and Socks Problem

Figure 8 shows a HTN plan example for solving shoes and socks problem. There are two kinds of action representations in HTN planning: composite actions and primitive actions.

A *composite action* is an abstract action which cannot be directly executed by an agent. It needs to be decomposed into simpler and lower-level actions or primitive actions. For example, "PutOnSocks" is a composite action that operates on both feet

and it is not simple enough for executing. Thus, this action needs to be decomposed into two simpler and lower-level actions ”LeftSock” and “RightSock”. On the other hand, a *primitive action* is a ground action which can be directly executed by an agent. For example,”LeftSock” is such a primitive action in the above example.

In a HTN planner, an *initial plan* which contains only problems and goals is viewed as a very high-level composite action. Moreover, a *solution plan* contains only primitive actions.

The advantage of HTN planning can be regarded into the following categories:

- *Flexibility*. The knowledge representation makes it more flexible to model planning domains and problems.
- *Expressiveness*. Hierarchical structure is easy for humans to understand.
- *Efficiency*. The efficiency can be greatly ensured by first searching for abstract solutions by exponentially pruning the search space.
- *Facilitating online planning*. It is possible for HTN planning to expand only some portions of a planning which needs to be executed immediately-that is, the interleaving between planning and execution is possible [18].

These advantages are guaranteed by HTN planner’s partial plan structure, sophisticated knowledge representation (not just in the action sequences specified in each refinement but also in the preconditions for the refinements) and good reasoning capabilities [5]. One can refer to Kambhampati’s comparative analysis report on POP and HTN planning [20] for a detailed comparison of the two algorithms. Due to these advantages, HTN planner can solve a variety of classical and nonclassical planning problems with magnitude more quickly compared with classical or neoclassical planners.

On the other hand, to implement the HTN approaches, a set of planning operators together with a set of decomposition methods are necessary for the domain modeler, greatly increasing its workload. Furthermore, HTN planning has difficulty in accommodating extended goals that require infinite sequences of actions, making HTN planning unsuitable for solving real-time domains with large state spaces.

Another disadvantage of pure HTN structure is that action interleaving between different branches is not represented. A low-level action might achieve the precondition of another action that is in a different branch, but the causal information between these kinds of actions is not represented in the plan structure. For example, “LeftSock” achieve the precondition “LeftSockOn” of the action “LeftShoe” and there is not data structure used to explicitly store the relationship. According to the hierarchical relations, all low-level actions are to be removed with respect to removal of one of high-level actions along its branch, even if some of low-level actions are still useful in the plan. It will increase planning costs. Thus, removing a set of actions connected hierarchically is less reasonable compared to removing those connected by causal links in POP.

In summary, HTN planner has the advantage of ensuring planning efficiency by using abstract actions to greatly prune search space. In contrast, it requires a lot of domain modeling works before planning starts, and the HTN relationship is less useful than causal relation which is explicitly represented in POP.

Another key to HTN planning is the construction of a plan library containing known methods for implementing complex, high-level actions [1]. The methods which are either knowledge-based or learnt from good problem-solving experience can be stored in the library and retrieved to be used as a high-level action. Similar techniques are also used in case-based Planning (CBP) [21]. Since using learning

techniques on causal explanation will be one of the future lines of work in this research, analyzing these library construction methods might be helpful.

A well-known HTN planner is SHOP2, which is derived from SHOP [22,23]. SHOP2 performed well in International Planning Competition (IPC)-2002. It is domain-independent and can be configured to work in many different planning domains, including real world temporal or dynamic planning domains.

2.1.3.5 Graphplan Planner

A planning graph is a special data structure that works with propositional planning problems that contain no variables [1]. It is a directed graph organized into levels [1]. Each level i is composed of a state level S_i and action level A_i. S_i contains all literals that could hold after the i^{th} step, while A_i contains all actions whose preconditions could be satisfied by some of literals in S_i. S_0 presents the initial state. S_{i+1} is a union of all literals in S_i and literals which can be achieved by effects of all actions in A_i. Figure 9 shows an example of the planning graph for the "have cake and eat cake" problem (We won't discuss about "mutex links" in this thesis, because relations that they represent are at the same levels and we are interested in relations that are between different levels).

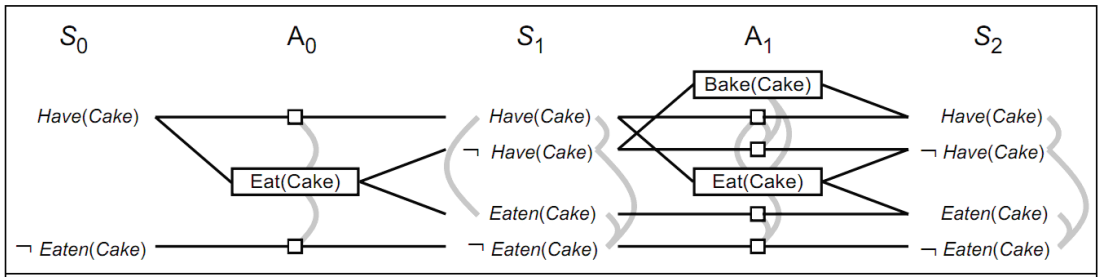


Figure The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

Figure 9: A GraphPlan Example (taken from [1])

Planning graph has following properties:

- Literals increase monotonically with levels.
- Actions increase monotonically. Preconditions of actions satisfied by a level are also satisfied by the next level according to the first property.

Because actions and literals are finite in the classical problem, there must be a level that is the same as its previous level. When this level is reached, the incremental planning graph construction can be terminated. Similarly, planning graphs are of polynomial size and can be computed in polynomial time. A *plan* output by a Graphplan planner is a sequence of sets of actions in a planning graph, following which goal can be found in one of the levels in the graph. As described in [5], Graphplan algorithm is sound, complete and always terminates.

As can been seen from the structure, the planning graph contains a rich source of information about the problem. First, the planning problem is solvable if and only if the goals can be found by *reachability analysis* in one of levels. Next, the count of levels from the initial state to the level containing all goals can be used as a cost value for evaluating heuristics. Because there can be multiple actions in each level, this way of evaluation is reasonable for estimating actual costs. Planning graph has been proven to be an effective tool for generating accurate heuristics and solving hard planning problems [1]. Those heuristics can be applied to almost all search techniques, like local search. LPG [2], the winner of 2002 IPC, is a fast planner that searches planning graphs using local search techniques (so-called Walksat procedure). It is important to note that, LPG can produce good quality plans by handling action costs. Although there are limitations due to the problem representation, LPG showed that local search works well in planning graph.

Planning graph is less general than a POP plan but more general than a total ordered plan. Actions in planning graph are also causally connected. For example

"Eat(Cake)" in level A0 and Bake(Cake) in level A1 are connected via state "not Have(Cake)" in S1 in Figure 9. This relationship is represented in every continuous two action levels, that is, greatly increasing redundancy in the plan structure. This is one disadvantage of planning graph structure.

Despite its advantages, the classical representation used in planning graph makes it unable to scale well in problem size (it has trouble in domains with many objects, which means large amount of actions need to be created), and cannot solve practical real-world problems.

2.1.3.6 Summary

The planning paradigms described above all have their advantages and limitations; it is so easy to say research on one of them is worthwhile and others are not. Recently, researchers in planning showed great interest in using combinatorial planning techniques to solve more complex larger problems. RePOP[17] planner (a partial ordered planner) and FF [24] (a state-of-the-art fully automatic planner in state space) are two good examples. They scale up better than Graphplan by using accurate heuristics derived from a planning graph. Thus, besides representational issues, research on combinatorial issues and developing useful heuristics is a promising way to derive good planning techniques to move the field of planning forward. Our research is headed this way.

2.2 Explanation Concepts

Explanation plays a key role in understanding, controlling and finally improving our environment. From Heider's seminal study on interpersonal relations, explanation of actions allows people "to give meaning to action, to influence the actions of others as well as themselves, and to predict future actions" [25]. Leake, D.

later on pointed out that explanation has similar effect on events by explaining their material causes [26].

2.2.1 Explanations Concepts in Other Areas

Explanation is widely researched in many fields, such as psychology, philosophy, and AI. Psychologists and philosophies have long studied and well developed explanation theories in natural science. More recently, explanation theories have been developed in Artificial Intelligence to facilitate learning and generalization. One example is applying explanation in case-based study of expert systems [27] to guide learning and searching. Another technique is having explanations help planning achieve good performance by explaining encountered failures or anomalies [28] [29][30]. The performance can be on speed, quality of a solution, etc. Besides these, some explanations are able to provide failure information to normal users [31].

The concept and methods of developing and using explanation in the following of the thesis are different from that in the studies above. Although planning is a field involved in AI, previous studies of explanations in other fields of AI are mainly focused on learning and generalization. On the other hand, its application in planning or CSP are to predict the failure in the future, thereby helping developers by pruning impossible searching branches, or to give important causal information of failures to users who are interested in what caused the failures [32][33].

2.2.2 Explanation Usages

As described in the above subsection, explanation can be seen as a tool or data structure for storing useful information and providing them to developers and users.

Planning domains are problematic if they have many dead ends in the search space and there is insufficient information for backtracking and dead ends detecting,

which are very common for complex and large real-world domain problems, since it is very hard for the domain modeler to systematically model all the information. This holds true for many real-world problems which are classified as contingency problems, where dead ends are very likely to be created dynamically in an unforeseen state, by unpredictable external events even they doesn't not exist initially. Local-search-based planning will probably be inefficiency for solving this kind of domain problems.

As analyzed in [19], the causal structure of domain contains vital information for heuristic search. The *explanation* hereinafter is to explain causal information in plan structure, like causal relation between actions. The information contained in explanation structures can be used to do more intelligent search in planning which can yield to better planning performance (either on planning speed or plan quality). We will provide more detailed research on our explanation-based techniques in Chapter 3.

At the first step of our research, we propose the usage of causal explanation structures. The idea of its usage in planning is inspired by POP, in which causal relation between some plans are modeled, and represented by causal links. Typically, causal explanations are used to explain the causality between actions or events. They can be in physical or mental aspects, but the utility of explanation to be discussed in this research is for a different purpose. Their usage in this research is focused on explaining causal relation between actions, regardless of what type of reasons.

In other words, we proposed an approach that is totally different from previous theories or applications, and the purpose is to speed up local-search-based planning by using causal explanations.

2.3 Macro-Actions Analysis

One reason of developing explanation structure for planning system is that by using explanations, causal relation between actions can be acquired automatically during planning, after which a planner can then make same or similar changes to a set of actions synchronously the same way as making changes to a normal action. In a way, some actions in a plan are put together. There are lots of planners using other combining techniques, such as macro-actions.

A macro-action [34][35] is a group of actions selected for application at one time like a single action. Learning and using macro-actions is promising in achieving significant improvement in planning. MacroFF [36] using forward chaining heuristic based on a relaxed Graphplan algorithm and Macro–AltAlt [35] which evolved from AltAlt [37], are two recent planners that performed well using macro-actions.

Macro-actions are well suited for use in classical planning systems. Most planning systems using macro-actions have two subsystems: macro learning system and planning system. Macro learning system are focused on and specialized to exploiting particular properties of the planners and the domain through off-line learning. Macro-actions generated by the learning system can be added into the original domain to get an augmented domain, and some of them can improve planning either in either speed, or length of plan.

Explanation-based techniques to be introduced in the following chapters uses combination concepts, because by using explanation structure a more significant change can be made to the plan in a single iteration, has same result as making a set of single changes in several iterations. It is similar to macro-actions, but the combination

is made and used temporally, and it is not to be learned and evaluated through or after the whole planning process.

Chapter 3 Using Causal Explanations in Planning

Having established the necessary background in Chapter 2, we can move on to detailed contents of the explanation-based approach in planning. The research is carried out based on the general loose plan structure that is analyzed in Subsection 2.1.3. The content in this chapter is organized as follows: First, we present a case example on Logistics domain in Section 3.1 for giving a better understanding of what information deserves an explanation, use the example to uncover some potential problems in the planning, and then characterize the problems into some cases. Next, inspired by POP, *causal links* are used to explicitly represent the causal relations between actions in this research. After adding causal links to the plan, actions are connected like a network. Thus, the network is called *causal network*. In Section 3.2, four types of causal networks are defined and are differentiated according to some restrictions of keeping these causal relations. After giving a comparison between them, we show that the usage of Multiple-In-Single-Out (MISO) causal network is more reasonable and less costly. Finally, we expand our research based on MISO causal networks. The detailed contents include the explanation data structures and the algorithms for updating and exploiting MISO causal networks. After that, we integrate these two algorithms into the general local-search-based planning process and highlight them in the revised planning process.

3.1 Case Study on Logistics Domain

In this section, we do not aim to discuss complex logistics domain, but to do our case study based on a simple logistics-type domain (illustrated in Figure 10).

The purpose of a *logistics-type* domain problem can be to obtain and move supplies and equipments in a timely fashion to some locations where they are needed, at a reasonable cost. Thus, logistics-type domain problems are very practical and it is encouraging if they can be efficiently solved. Several issues that are very common in logistics-type domains are listed as follows:

- *Storage capacity*. For the case that a facility has limited storage space.
- *Transportation efficiency*. For saving facility resources.
- *Safety stocks*. It is insurance as unexpected the high demands can be met despite unexpected events (trucks breaking down).

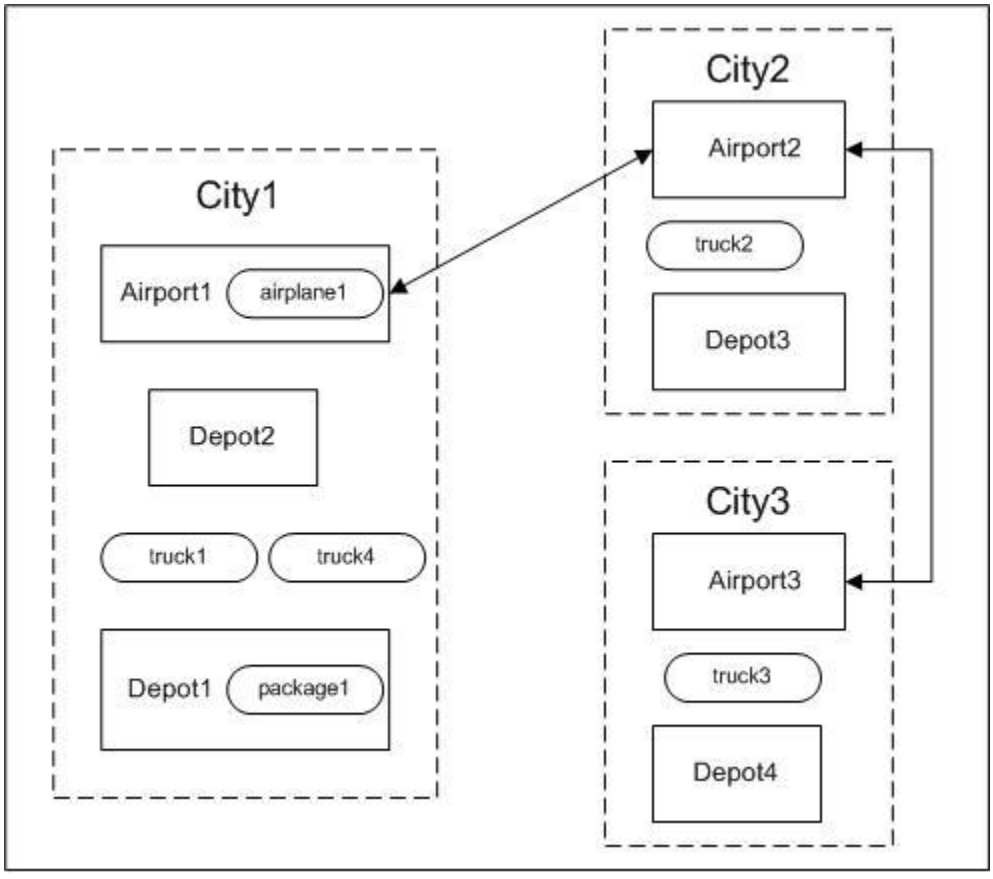


Figure 10: A Logistics Domain Example

Logistics domain planning problems are often modeled in the way like delivering some packages to some locations by utilizing various forms of transportation, such as driving truck or flying airplane [38,39]. The constraints on the

problem may include storage capacity, transition reachability by different types of vehicles, etc. In this research, the notation "object.attribute == value" is used to represent a state of the object's attribute "object.attribute". The state can be a precondition or an effect of an action, or a goal.

3.1.1 Case Example on Logistics Domain

In the example, "package1.location == Depot1" and "package1.location == Depot4" are the initial state and the goal state of "package1.location", respectively. Trucks can travel between depots in a same city, but cannot travel across different cities. Airplanes are able to be located only in airports, and fly across different cities; but they cannot fly without holding a package. Available actions in the domain are in the following: "load/unload a package to/from a truck" (e.g. load(package1, truck1), this action representation is used in this research)), "move a truck from one depot to another" (e.g. moveTruck1(Depot1, Airport1)), "fly a airplane from one airport to another" (e.g. flyAirplain1(Airport1, Airport2)). Furthermore, the airports are specific depots.

To reach the above goal, the planner might construct a plan with the following steps: First, add the following five actions in sequence: load(package1, truck1), move(truck1, Depot1, Airport1), unload(package1, truck1), load(package1, Airport1) and fly(airplane1, Airport1, Airport3). At next step, the planner finds that the action fly(airplane1, Airport1, Airport3) is infeasible, because there is no direct transition between City1 and City3. Thus, "flyAirplane1" is removed from the current plan, and the action load(package1, Airport1) is then removed because it reduces plan quality. At this point, *package1* is in *Airport1*, and all trucks are empty. To repair the plan, the planner might add two new actions: "load(package1, truck1", and "move(truck1, Airport1, Airport3)". However, the new plan would definitely fail again since the

constraints state that a truck cannot travel across different cities. Moreover, the planner might repeatedly add/remove same actions into/from the current plan in the rest of the iterations.

3.1.2 Characterizing Causes of Inefficiencies

The above planning process is naive and is inefficient. The inefficiency might be because of the following reasons:

- When a action (e.g. fly(airplane1, Airport1, Airport3)) is to be removed from the plan, some actions (e.g. load(package1, Airport1)) which exist only to satisfy this action can also be removed. This is an obvious choice by human beings, because they know the causal relation between actions, and can use this causal relation to make an intelligent plan repair. However, a local-search-based planner that has the loose plan structures needs to take a lot of time for analyzing and searching due to a lack of straightforward causal structures.
- In the case that reverse actions (e.g. move(truck1, Depot1, Airport1) and move(truck1, Airport1, Depot1)) exist, it is possible that the planning might go into a loop that is composed of a set of states (any attribute value is a state of the attribute.), and repeatedly explore some visited states in the loop. In the above example, the states in the loop are "*truck1.location == Depot1*" and "*truck1.location == Airport1*". The looping can greatly slow down the planning. Although the low efficiency problems that are caused by the looping might be able to be avoided by using tabu-lists to store recently explored states, it is still intractable if there are too many states in a loop and the tabu-list is too short.

- Actions that have same type but operate on different objects are different and the experiences on different actions cannot be simply shared. For example, there was a loop that is composed of three actions that move *truck1* between three different locations and this loop was solved by some method. Another similar loop might occur, the only difference is that the actions in this loop operate on *truck4* (*truck4* has the same initial state as *truck1*). Even though the new loop is similar to the solved loop, the planner cannot avoid it by using the experience of the actions in the solved loop. A potential way to address this problem is to abstract general information from the experience of some actions of a type and share this information for all actions of this type.

In a word, a planner that has a loose plan structure suffers from a lack of useful information, and a great overhead might be caused by useless, repeating or unintelligent planning operations. According to the above case study, we conclude that some information, like the causal relations between actions, bad/good planning experiences, can be used to guide more efficient search. However, the way of retaining and using different kinds of information should be different. The planning efficiency will also depend on heuristics or algorithms that are based on the enhanced plan structures by using those kinds of information. In the current stage, the explanation usage in this research is focused on addressing the first inefficiency.

3.1.3 Analyzing Causal Information

Let's give a more detailed analysis on causal information in order to acquire the proper information. Causal information herein means causal relations.

Supposing *S* is a set of states, *c* and *e* are two states that belong to *S*, a causal

relation between a cause *c* and an effect *e* is a pair denoted by <*c*, *e*>, meaning that *c* caused *e*. Causes and effects are typically related to changes or events.

In planning, operations that can change a plan can also have causal relations. These operations are called *plan changes* hereinafter. The basic plan changes can be adding/removing an action/object into/from the plan, etc. Taking Figure 10 as an example, *c* and *e* can be plan changes that adds the action "load(package1, truck1)" and that adds the action "move(truck1, Depot1, Airport1)", respectively. This pair <*c*, *e*> is due to a causal relation $<c_a, c_e>$ between the above two actions. However, the lifetime of those plan changes is only one iteration, i.e., they are used to update a plan but not stored in the plan. Note that, a plan change can also be composed of several basic plan changes, which might cause the amount of plan changes to increase exponentially relative to that the amount of actions. If utilizing causal relation between plan changes, the planners might then need to store a significant amount of extra information, i.e., the plan changes and their relationships, and therefore may require more time for evaluating successor plans. Thus, it is not promising to use causal information between plan changes.

Instead, using causal information between actions is far less costly because actions themselves are a part of the plan. Furthermore, the causal relations between actions are easier to detect and can be acquired during planning, i.e., they are not necessary to be modeled in domain definitions. Thus, it is promising to explicitly represent causal relations between actions rather than between plan changes.

Symbolic Attribute	Value Transitions
airplane1.location	Airport1 ⇔ Airport 2
	Airport 2 ⇔ Airport3

Table 1: Transition Table of a Symbolic Attribute

On the other hand, although the purposes of adding actions to a plan are the same (i.e., to improve plan by resolving inconsistencies), the reasons of adding them might be different. A detailed analysis for these cases will be given below, and the analysis is based on symbolic attributes. Similar to the approach of the encoding in CSP [40], symbolic attributes can be regarded as variables range over symbolic domains. For example, the value of "airplane1.location" as shown in Figure 10 can be any of "Airport1", "Airport2" and "Airport3". The value transitions of the symbolic attribute "airplane1.location" are shown in Table 1. These transitions can be regarded as a set of constraints over the attribute. The attribute's next state is dependent on its current state, like the next value is possible to be "Airport3", only if the current state is "airplane1.location == Airport2". Now, let's move on for more details of general causal information in planning.

In planning, an action can contain multiple preconditions/effects. An action cannot be executed until all of its preconditions are satisfied. The planner might add a new action into the plan due to the following reasons:

- ***Direct causal relations between actions.*** The new action has an effect that fully achieves a goal or a precondition of an existing action in the plan. For example, if the action "fly(airplane1, Airport1, Airport2)" is in the current plan, and one of its preconditions "airplane1.state != empty" is not satisfied, i.e., it is an inconsistency. Another action "loads(package1, airplane1)" can achieve this precondition. In this case, there is a *direct causal relation* between these two actions.
- ***Indirect causal relations between actions.*** The new action has an effect that can somehow *reduce* a distance to the state of an unsatisfied precondition/goal. For example, if the goal is "airplane1.location ==

Airport3", and currently "airplane1.location == Airport1", then the action "fly(airplane1, Airport1, Airport2)" can be added into the plan because it is on the half way to achieve the goal (referring to Table 1). The causal relation in this case is *indirect*.

Grouping a set of actions that have indirect causal relations is reasonable.

In summary, the way of using causal information between actions is very promising, and two types of causal relation can be classified. Techniques on retaining and using this information will be discussed in the following sections. Now let's move on to the next section to discuss to the way of using the smallest possible data structure to represent as much causal information as possible.

3.2 Classification of Causal Networks

To better describe our work, we would first introduce a concept named *causal network*. A causal network herein is an action network *N* that can be denoted by a tuple <*A, C*>, where *A* and *C* are a finite set of actions and directed causal links, respectively. Each action $a_i \in A$ is a node in *N*, while each causal link $c_i \in C$, is a directed link between two nodes, and all of them have the same direction if time representation is used. Being inspired by the representation of causal relation between plans in POP [41][42][16], using causal links to explicitly represent the causal relations in planning is expressive and intuitive. A *causal link* represents a causal relation between two actions. Furthermore, the direction of causal links means that one action achieves a precondition of another action.

The motivation of using causal explanations is that the actions which are closely dependent on each other can be grouped as a macro action and operating on this kind of macro-actions like on a normal action is reasonable.

In this thesis, the size of *N* is measured by the addition of the sizes of *A* and *C*. Since an action might have multiple preconditions or multiple effects, the network size will be very large if all causal relations are kept. For example, action "unload (Package, Truck, City)" has two preconditions: "Package.location == inTruck" and "Truck.location == City", that can be achieved by two different actions. Similarly, an action that has multiple effects can achieve multiple preconditions that are in different actions. To have a better understanding of causal networks and how to keep causal information is beneficial, we characterize causal networks into four types and give a brief comparison. They are named Multiple-In-Multiple-Out (MIMO), Multiple-In-Single-Out (MISO), Single-In-Multiple-Out (SIMO) and Single-In-Single-Out (SISO) networks, respectively, and are differentiated according to some restrictions of keeping causal information. Figure 11 and Figure 12 illustrate the four types of causal networks. The "MI" property in MISO and MIMO causal networks herein indicates that an action in the casual networks is allowed to have multiple causal link inputs. Similarly, the "O" means an action's casual link output, and the "S" property restricts the amount of causal link inputs/outputs. Taking action a2 and a6 MISO as an example, a2 is called "linking action" and a6 is called "linked to action" in causal networks.

An action in MIMO causal networks is allowed to be connected with multiple linking actions and linked to actions in terms of "MI" and "MO" properties, respectively. Its advantage is that the planner can keep all of the causal information, while its disadvantage is that maintaining robust causal structures is costly. A MIMO causal network that has *n* actions needs *O(n!)* memory to keep all of the causal relations in the worst case (Actions are ordered in sequence and each action has causal relations with all of actions that are ordered after it). Moreover, it is not easy and is

costly to find out which subset of actions deserves to be grouped together as a temporal macro-action in a strongly connected MIMO causal network.

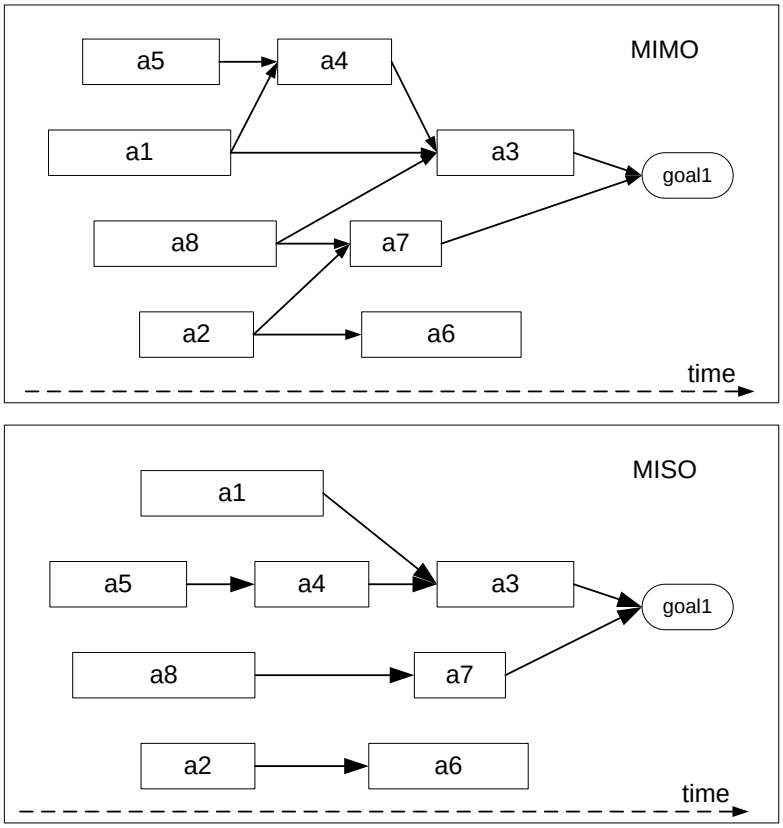


Figure 11: Illustrations of MIMO and MISO Causal Networks

In MISO, the "SO" property restricts that each action has at most one causal link output, indicating that there is only one major purpose for the planner to add this action into the plan. The "SO" property is reasonable even if the action might have multiple effects. For example, the action "John buys an apple" achieves the following two states: "John owns an apple" which is a precondition of the action "John eats apple", and "John's money is used up", which is a precondition of another action "John earns money". However, only the first state is the purpose for John buys an apple. Therefore, only using one causal link to explicitly represent this purpose is reasonable. In terms of the "SO" property, a MISO causal network that contains *n* actions needs only *O(n)* extra memory to keep the allowed causal links, since the set of causal links go out of all actions in MISO is also the set of causal links in MISO.

Similarly, the memory usage for keeping causal links is *O(n)* in SIMO and SISO causal networks that contain *n* actions in terms of the "SI" property. Nevertheless, the "SI" property is not as reasonable as "SO" property, because all the preconditions of an action are prerequisites for the action's occurrence.

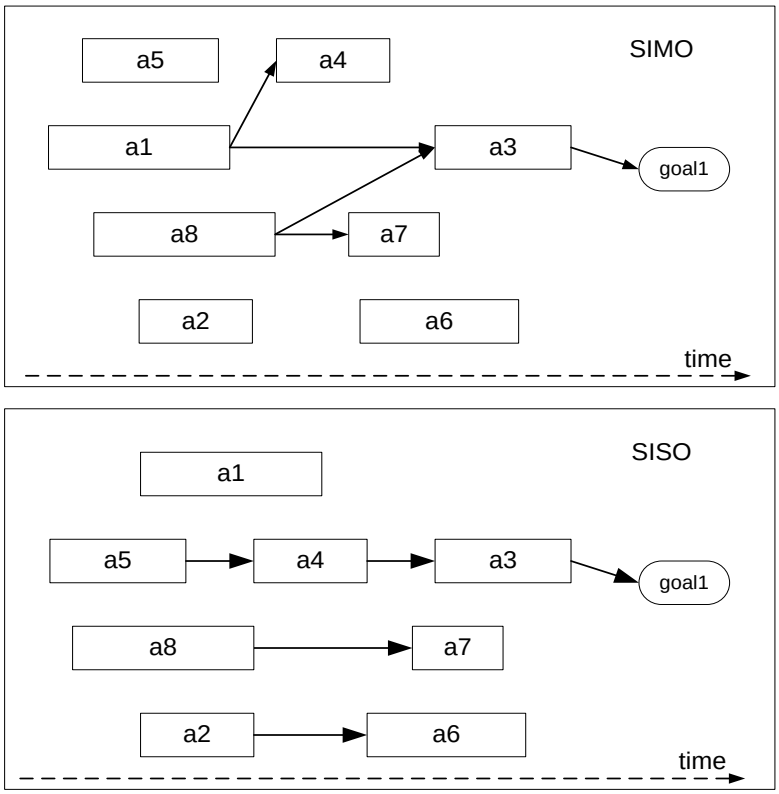


Figure 12: Illustrations of SIMO and SISO Causal Networks

In summary, to get a balance between the extra time and memory costs and the time costs reduction due to the causal information, constructing MISO causal network is promising and reasonable. The current research is focused on the MISO structure. The details on the MISO causal networks and two algorithms for updating and exploiting them will be expanded in the following sections. To give an overall picture of when the algorithms will be used, we will first introduce a general local-search-based planning process in the next section, and then highlight the use of the above two algorithms in the planning process. Evaluations of the other three types of plan structure are our future work to give a performance comparison with MISO.

3.3 Integrating Explanation-based Algorithms to the Overall Planning Process

Figure 13 illustrates the general process of the planning that uses local search to iteratively repair the plan. Referring to the local search paradigm illustrated in Figure 3, the whole circle can be regarded as a plan space, and every point is a plan. There is an objective function for evaluating plan quality in plan space. The initial plan only contains the initial states and goals. In every iteration loop, the planner first uses local search to search for successor plans that are located in the neighborhood of the current plan, using the objective function to evaluate them. *Successor plan* herein means a plan that can be achieved by making some changes, like adding/removing an action to/from plan, to the current plan. If there are better successor plans in terms of value of objective function, then the current plan is replaced by the successor plan. If not, the current plan is a local optimal plan, and some local search based heuristics can be used to jump out of the local optima (refer to Subsubsection 2.1.2.3 Discussion).

The step "Evaluate successor plans" in Figure 13 might be inefficient, due to a lack of explicitly represented causal information for searching better successor plans. To address this potential inefficiency, we developed two algorithms and integrate them into the above general planning process (highlighted in Figure 14). The "Evaluate successor plans" step includes two sub-steps: searching for successor plans and evaluating them. Exploiting algorithms are used in the "searching" sub-step, by using causal information between actions to facilitate local search. Updating algorithms are used for maintaining the MISO causal structures. The details of updating and exploiting algorithms will be introduced in the following two sections, respectively.

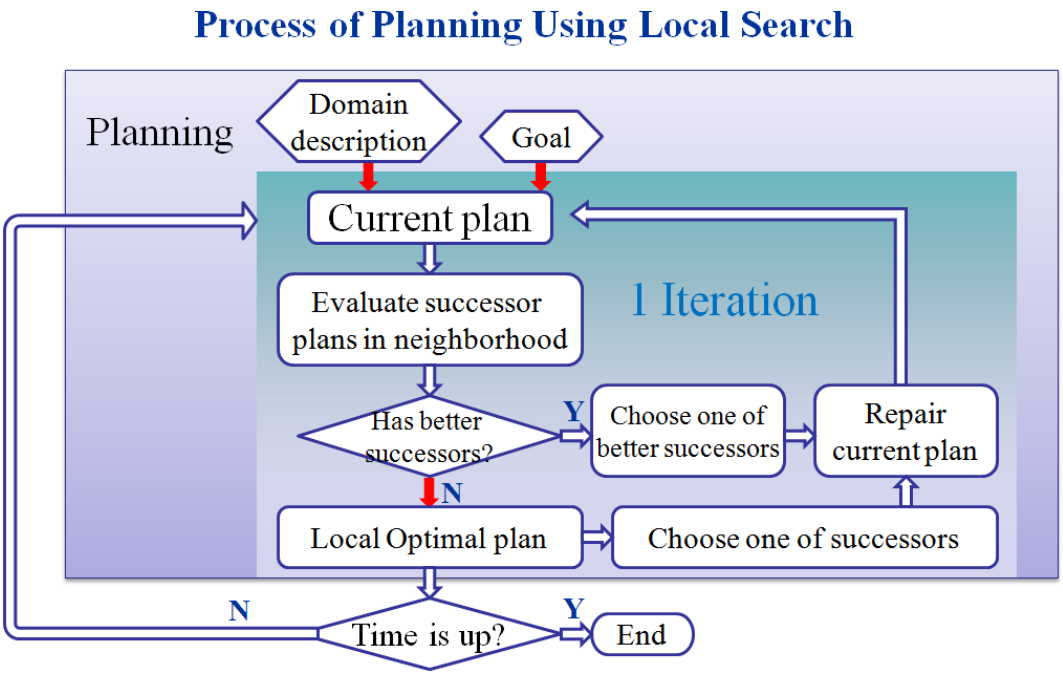


Figure 13: General Local-Search-based Planning Process

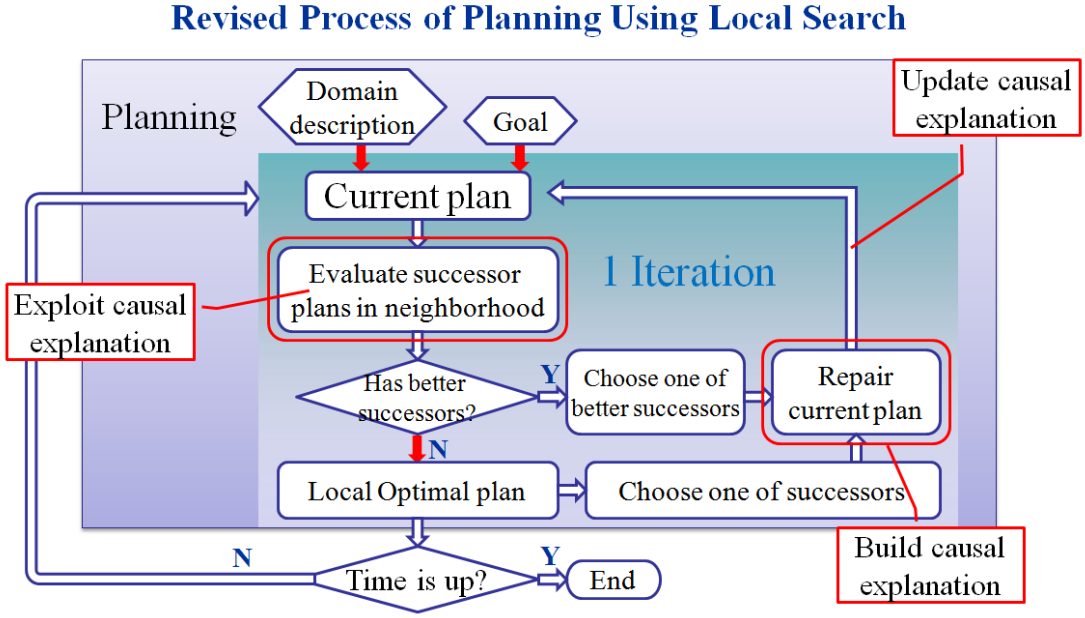


Figure 14: Local-Search-based Planning Process Using Causal Explanation Algorithms

3.4 Constructing MISO Causal Networks

Algorithms and data structures are inherently related, that is, algorithms can be measured by their efficiency in processing the data. Thus, designing a good plan structure is very important for improving the planning performance. We will first

introduce causal explanation data structure, and then introduce the updating causal explanation algorithms for constructing the MISO causal networks.

3.4.1 Explanation Data Structures

Causal explanation data structures can be divided into two parts. The explanation elements and their functionalities are listed in Table 2. Causal links are stored outside actions and used to connect two actions. While in every action, there is a specific data structure that contains a counter and a set of pointers. More details are in below.

(1) Causal Link

To retain as much information as possible, let's have a look at general action structures in planning system.

Explanation Element		**Functionality**
Outside actions	*causal links*	A causal link connects an effect and a precondition that are in two different actions.
Inside an action	*EoutCounter*	Count the number of preconditions inside other actions that are satisfied by the action.
	A set of pointers to causal links	One or two of pointers point to causal links that go out of the action; At most one is direct/indirect; The other pointers are stored in a container. They point to causal links that go into the action.

Table 2: Element of Causal Explanation and Their Functionalities

From the perspective of the action's level (taking an action as a whole), a causal relation is represented by a causal link between two actions. It indicates the existence of the causal relation. More detailed causal information, like why they are causally related, is not indicated at this level. For example, suppose an action "move(truck1, Depot1, Airport1)" in logistics-type domain has one causal action predecessor: "move(truck1, Depot2, Depot1)" (suppose truck1 is initially located in Depot2), and the actual reason of adding the latter action is to achieve a precondition

"truck1.location == Depot1" of the first action. If a causal link is just added between these two actions at the action's level, planners cannot get above detailed reason. These kinds of reason might be very useful for exploiting and updating plan structures.

To address this problem, we choose to make a causal link connect between two components, i.e., an effect and a precondition, that belong to two different actions, respectively. Compared to using causal link at the action's level, our way of connection doesn't increase memory expense, but represents more information, like why two actions are connected for example. Figure 15 illustrates two types of causal relations. A causal link that represents a direct causal relation between two actions (refer to 3.1.3) are called a ***direct causal link.*** It is illustrated by a solid line between two actions in Figure 15 (a). On the other hand, an ***indirect causal link*** is defined as a causal link that represents an indirect causal relation between two actions. It is illustrated by a dashed line in Figure 15 (b). An indirect causal link $<a_i, a_j>$ can be updated by a direct causal link $<a_k, a_j>$ when a_k is inserted between a_i and a_j for satisfying a precondition in a_j. Since the direct causal relations are more straightforward than indirect causal relations, planners should have higher priority to exploit direct causal links than to exploit indirect ones when it has multiple causal links to exploit during planning.

By using indirect causal links, the "SO" property holds only regarding one type of causal links in MISO rather than regarding both types, regardless of the other type, that is, an action is allowed to have at most one causal link of any type (direct/indirect) going out of it. However, it can link to a direct causal link and an indirect causal link at the same time. On the other hand, although there can be multiple causal links going into an action, for a precondition that is related to a

symbolic attribute, there are only two possibilities: it is fully satisfied by an effect or not. Thus, a precondition can link to at most one causal link.

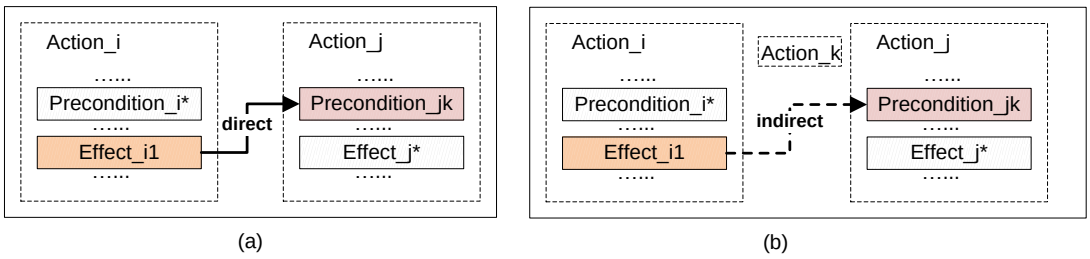


Figure 15: Illustrations of Causal Links

(2) *EoutCounter*

For the purpose of reducing complexity of causal network, in MISO, an action links to at most one action even if its effects might satisfy more than one precondition. The "*EoutCounter*" inside an action is a mechanism that is used to somehow compensate for the loss of large amount of causal information, and can be used by exploiting algorithms to judge whether it is worth further exploiting other causal links. Take note that the counter only counts the number of direct causal relations going out of the action.

For example, if action "John goes to school" achieves state "John.location == school", its main purpose is satisfying a precondition of "John takes exam". Meanwhile, the state is also a precondition of action "John has lunch at school", "John does experiment" "John talks to his supervisor". "*EoutCounter*" inside the action "John goes to school" is assigned by value "4" according to the number of preconditions it is satisfying. If John suddenly gets notification that the exam is rescheduled to another day, then it is not reasonable to reschedule "John goes to school" to the same day as the exam, because this action is the prerequisite for the other three action. Thus, it is better to stop exploiting other causal links when the

algorithm exploited an action that has a large value of *EoutCounter*. The way of utilizing EoutCounter will be introduced in the next section.

(3) Pointers inside an Action

Exploiting algorithms should be able to exploit interleaving actions and causal links. It is easy to access the two actions when given a causal link, but not vice versa, because a causal link doesn't belong to any action.

There are two ways to solve the problem when exploiting algorithms encounter an action. One of methods is searching the library of causal links to find out which causal links are linking to the action, and heuristically select one of causal links to exploit. The other one is allowing actions to have access to its connected causal links. The first one increase time expenses for searching all causal links. It needs $O(n^2)$ time to exploit a MISO causal network that has n actions. The second method takes more memory relative compare to the first method to ensure action's access capability to its connected causal links. The extra memory is $O(n)$, because only two of the actions need to access one causal link. We choose to use the second method. Some pointers are used to address the problem.

3.4.2 Updating MISO Causal Network

In this section, we will introduce when and how to iteratively update MISO causal networks according to the explanation data structure described in the previous subsection.

The plan structure needs to be updated when some changes happen. Changes occur when an action is added/removed into/from the plan, or an action is moved to other time intervals in planners that use action projection representation, and so on.

3.4.2.1 Updating when Adding an Action

Now let's have a look at how MISO causal actions networks are constructed.

As mentioned in subsection 3.4.1 causal links are directed. That is, a causal link connected to an action can either go into / link to the action or go out of / be linked by the action. Thus, to decide whether to add causal links between the new action and an existing action in the plan, we have to consider the following generalized cases:

I) Looking forward, currently there is no causal link connected to an unsatisfied precondition, and the new action is added to directly reduce/remove the inconsistency of the precondition;

II) Looking forward, currently there is no causal link connected to an unsatisfied precondition, and the new action is added to indirectly reduce/remove the inconsistency of the precondition;

III) Looking forward, currently there is an indirect causal link connected to an unsatisfied precondition, and the new action is to directly/indirectly reduce/remove the inconsistency of the precondition;

IV) Looking backward, some precondition of the new action is satisfied by an existing action, and there is no causal link going out of the existing action;

V) Looking backward, some precondition of the new action is achieved by an existing action, and there is a direct causal link going out of the existing action;

VI) Looking backward, some precondition of the new action is achieved by an existing action, and there is only an indirect causal link going out of the existing action;

Before going through the above cases, let's introduce some notations to give a clear and simple description. In the examples, a_i means the i^{th} action; ef_i means the i^{th} effect; c_i means the i^{th} precondition; and $a_i.ef_j$ means the j^{th} effect is in the i^{th} action. The "*new_*" and "*existing_*" are temporally used in examples only to indicate whether an action is in the plan before current planning iteration, like new_a_i means a_i is a new action to be inserted into the plan, while $existing\_a_i$ means a_i is an action already existing in the plan. The subscripts used in these notations are arbitrary and only used to discriminate different actions or action components. On the other hand, symbol "$\Rightarrow$" and "$\rightarrow$" are used to represent direct and indirect causal relations in formulas, respectively.

If an action contains only one precondition/effect, then it is easy to add a causal link connected to/connected by the action. However, take note that one precondition can be satisfied by a sequence or set of actions. For example, one effect might also satisfy multiple preconditions/goals, which can be in different actions (denoted by "$a_1.ef_1 \Rightarrow a_2.c_1$", and "$a_1.ef_1 \Rightarrow a_3.c_2$"). Similarly, effects ef_1 and ef_2 belong to action a_1 and a_2 respectively, and precondition *c1* might be satisfied by the combined effect of ef_1 and ef_2 ("$a_1.ef_1 + a_2.ef_2 \Rightarrow a_3.c_1$").

On the other hand, there can be several options if an action has several preconditions to be satisfied, or when adding an action which has multiple effects being able to satisfy multiple preconditions in other one or multiple actions. More than two actions are actually causally related between each other, so the situation is more complex. For example, an action "move(truck1, Depot1, Depot2)" has two unsatisfied preconditions "truck1.location == Depot1" and "package1.location == truck1", and both of them are inconsistent.

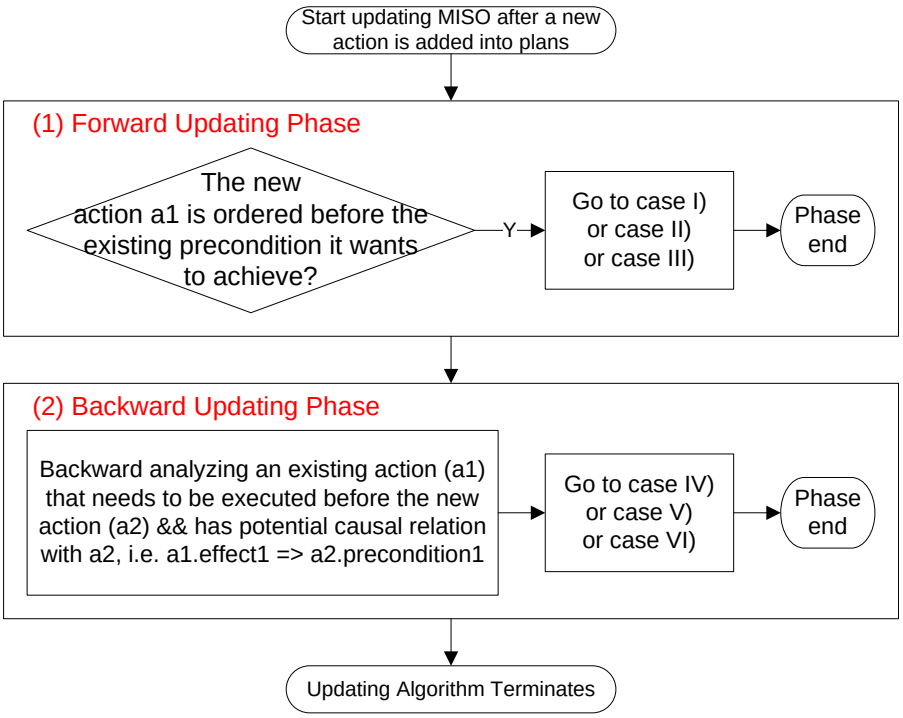


Figure 16: Overall Process of Updating MISO Algorithm after Adding a New Action

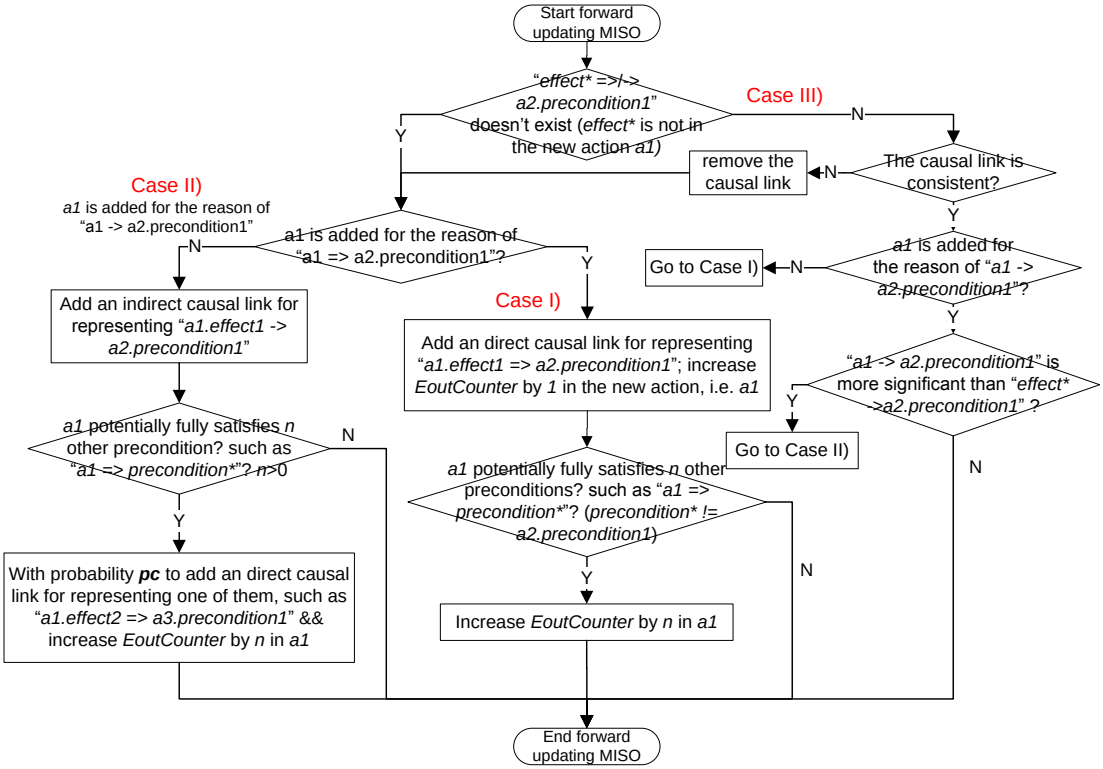


Figure 17: Process of Forward Updating MISO

Now, let's analyze each case in detail. Figure 16 illustrates the overall process of updating MISO algorithms after adding a new action into plan. The proposed algorithms are composed of two updating phases: forward updating and backward

updating. They are described in (1) and (2) as follows respectively. Furthermore, both of phases are classified into several cases.

(1) Forward Considering Causal Information

Case I) ~ case III) listed above need to look forward, in order to add a causal link that goes out of the new action: new_a_1. The forward updating processes are illustrated in Figure 17. Take note that the updating algorithms need not consider whether the other preconditions of $existing\_a_2$ are connected to some causal links or not, since an action can have multiple preconditions that are inconsistent and all of them should be satisfied.

I) Case of Explaining an Direct Causal Relation to an Unexplained Precondition

The case I) is simple. Figure 18 shows a concrete example of forward updating causal structures in this case.

The case can be generalized as follows: If $new\_a_1.ef_1$ is suggested by a planner to fully satisfy $existing\_a_2.c_1$ and $existing\_a_2.c_1$ is not connected to any causal link, then the causal information of "$new\_a_1.ef_1 \Rightarrow existing\_a_2.c_1$" is retained inside the new action, after the new action is added into plan, a direct causal link representing the causal information is created and connected from $new\_a_1.ef_1$ to $existing\_a_2.c_1$ and "EoutCounter" in new_a_1 is accordingly increased by 1.

The next step is to propagate explanations. For example, new_a_1 might fully achieve n other unsatisfied preconditions in the plan, such as $new\_a_1.ef_2$ *(*or $new\_a_1.ef_1$*)* also fully satisfies $existing\_a_3.c_2$, i.e., there are no other actions that changes the state of $new\_a_1.ef_2$ occurs between new_a_1 and the preconditions. In terms of "SO" property of MISO causal network, the planner won't add direct causal

links to represent the potential causal relations. Instead, it increases "EoutCounter" by *n* to record their existence. The updating algorithm is based on the agreement on the assumption that the first account for adding an action (i.e., "$new\_a_1.ef_1 \Rightarrow existing\_a_2.c_1$") is the most important reason of the addition, even if the new action potentially resolves other inconsistencies, because the inconsistency that is suggested to resolve by the planner is regarded as the most significant one, thus the reason of resolving it deserves higher priority to be stored and the reason is also the first consideration of the addition.

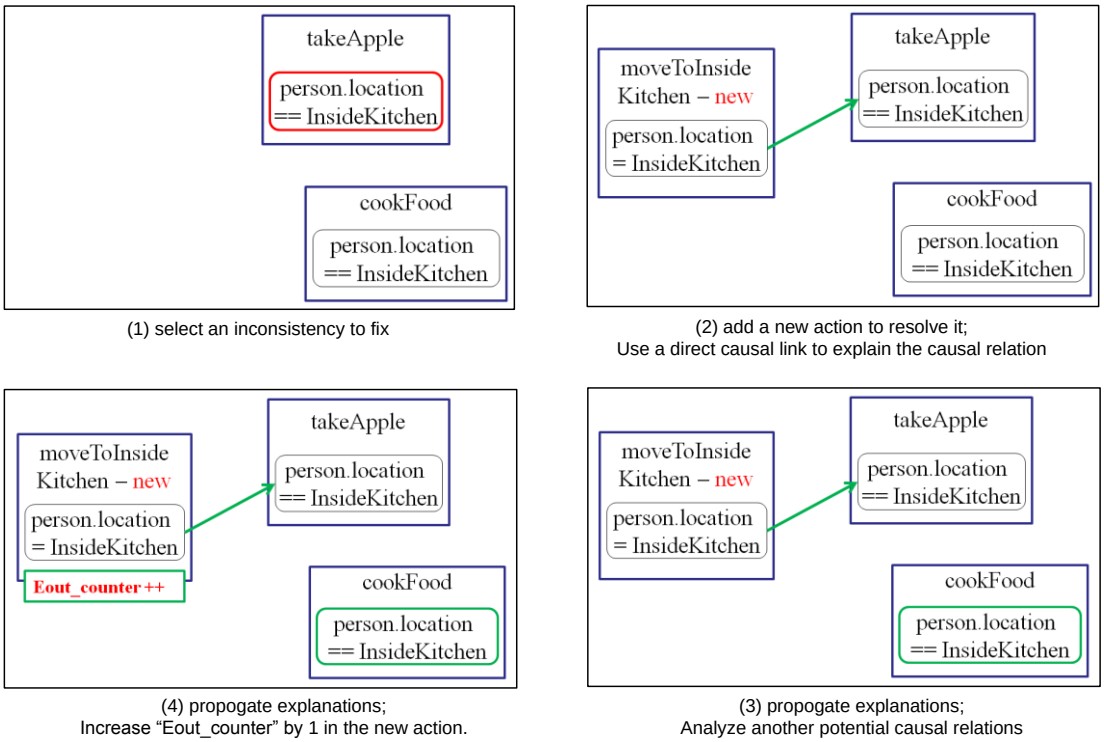


Figure 18: An Example of Updating Causal Explanation Structures When Adding a New Action that Directly Resolves an Inconsistency that is Totally not Resolved

II) Case of Explaining an Indirect Causal Relation to an Unexplained Precondition

Figure 19 shows a concrete example of forward updating causal structure in four steps in case II). The updating algorithm is as follows. It is more complicated than case I).

Firstly, the updating algorithms add an indirect causal link connected from *new_a*$_1$.*ef*$_1$ to *existing_a*$_2$.*c*$_1$ (denoted by "*new_a*$_1$.*ef*$_1$ → *existing_a*$_2$.*c*$_1$").

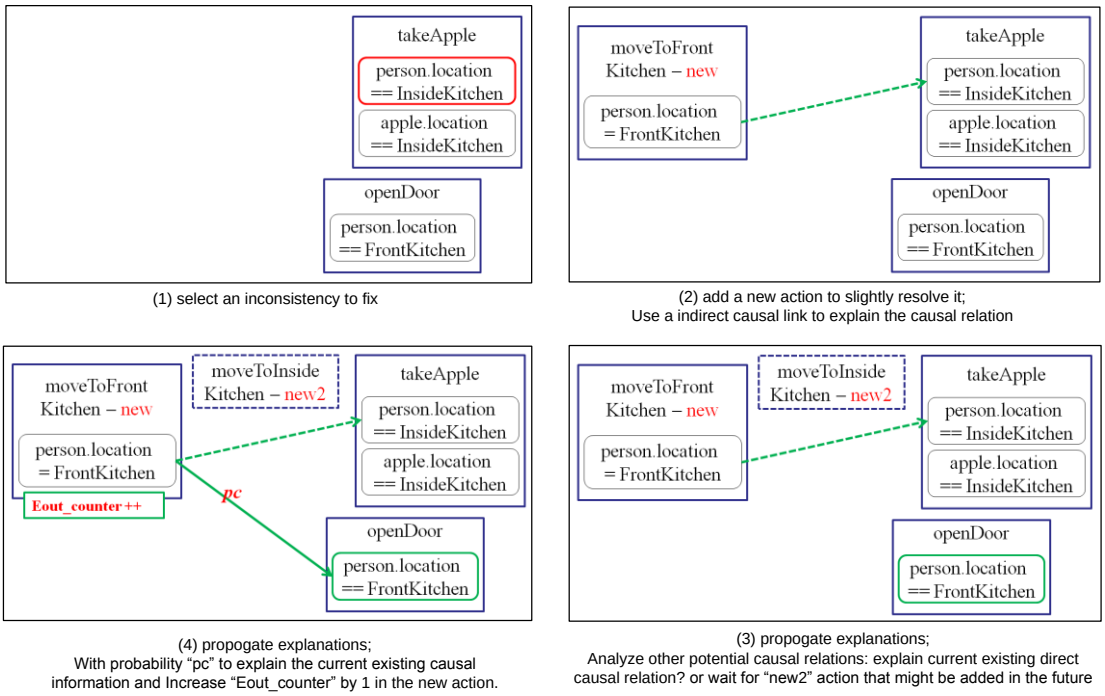


Figure 19: An Example of Updating Causal Explanation Structures When Adding a New Action that Partially Resolves an Inconsistency that is Totally not Resolved

Next, there is a key point needs to be considered when propagating explanations. Besides the initial reason of adding *new_a*$_1$, *new_a*$_1$ might also fully resolve *n* other inconsistencies (*n*>0), such as the potential causal relation denoted by "*new_a*$_1$.*ef*$_2$ ⇒ *existing_a*$_3$.*c*$_2$"), then it means *new_a*$_1$.*ef*$_2$ is significant to *existing_a*$_3$.*c*$_2$. Whether to add a direct causal link between *new_a*$_1$ and *existing_a*$_3$ or not, needs to be considered, because an action *new_a*$_4$ that satisfies "*existing_a*$_1$.*ef*$_1$ ⇒ *new_a*$_4$.*c*$_3$" and "*new_a*$_4$.*ef*$_3$ ⇒/→ *existing_a*$_2$.*c*$_1$" might be added into plan in the future, that is, *new_a*$_1$ and *new_a*$_4$ are in the sequence to achieve *existing_a*$_2$.*c*$_1$. Thus, the updating algorithms encounter a choice point once again. The latter scenario is initially expected, thus the causal relation between a_1 and a_4 is stronger and more reasonable than that between a_1 and a_3. However the scenario also might not occur.

Since MISO allows at most one direct causal link going out of a_1, only one of causal relations respectively in the above two scenarios can be explicitly represented.

Thus, we proposed a **strategy** that uses a probability ***pc*** to randomly represent a direct causal link in first case, that is, with probability of *1-pc* to wait for the occurrence of the second scenario. The probability *pc* might affect the performance of the planning, because it can affect the decision of explaining which causal relation. Thus, the usage of *pc* is an important technique, which might be problem dependent. It is a strategy for heuristically updating causal network when the planner encounters a new action that is added for the sake of an indirect causal relation. The evaluation of the affect of *pc* on can be our future work.

III) Case of Explaining a Causal Relation to an Explained Precondition

Case III) takes charge of the scenarios that the precondition that is to be achieved by the new action is already connected to an indirect causal link before adding the new action, or the previously connected direct causal link become inconsistent due to addition of the other actions.

In the second sub-case, the inconsistent causal link needs to be removed from the plan, and the precondition is then not connected with any causal links. Thus, the algorithm will jump to either case I) or II).

In the first sub-case, the existing indirect causal link might need updating. As illustrated in Figure 17, the forward updating algorithms will go into the other three sub-branches according to different scenarios. If the new action new_a_1 is added to fully achieve the precondition, then the process is the same to that of case I). Otherwise, the reason of the addition of new_a_1 is an indirect causal relation to the precondition. If the new causal relation is more significant than the existing one, then

the algorithms do the same process as that of case II), otherwise, it is unreasonable to explain a weaker casual relation, the forward updating phase will thus terminate.

Take note that, up to this point, the precondition might be connected to two casual links, at least one of which is an indirect causal link. Each precondition is only allowed to be connected to one of them, that is, the first indirect causal link needs either to be removed or to be updated by replacing this precondition with another one). This part of updating will be done in the backward updating phase because backward propagations will occur in this sub-case. More details are in (2). Case VI) in the following and Figure 22 illustrates a concrete example of the updating process.

In summary, the value of "EoutCounter" in the newly added action needs to be calculated after analyzing the all related actions in plans.

(2) Backward Considering Causal Information

Case IV) ~ VI) listed above need to search backward, in order to find causal relations going into the new action: new_a_1. It is the explanation's backward propagation in nature, because adding an action just because that some of its preconditions can be satisfied by the current plan, but without a good contribution to the plan is unreasonable.

Figure 20 shows the process of backward updating MISO, while Figure 21 and Figure 22 illustrate three concrete examples of the cases IV) ~ VI) respectively. The process will go into one of the three branches at somewhere (i.e., cases IV) ~ VI)), and then goes back to a common sub-process at the end. The sub-process is to forward check whether all the preconditions that are to be achieved by new_a_1 is currently connected to a redundant indirect causal link, since a precondition on a symbolic attribute only allowed to be connected with at most one causal link in MISO. If yes,

then remove/update it. Take note that the backward updating process should be done for all preconditions in *new_a*$_1$.

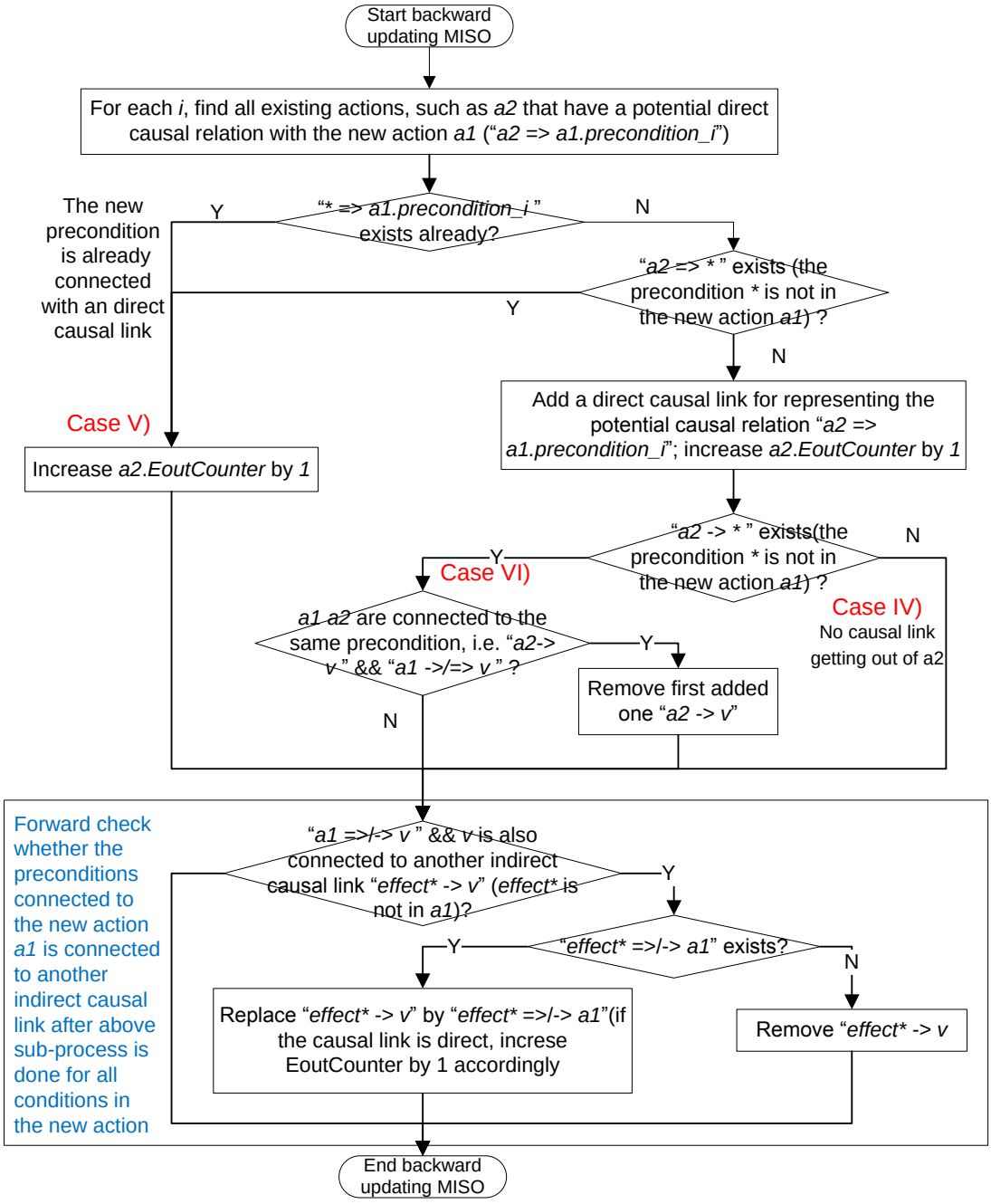


Figure 20: Process of Backward Updating MISO

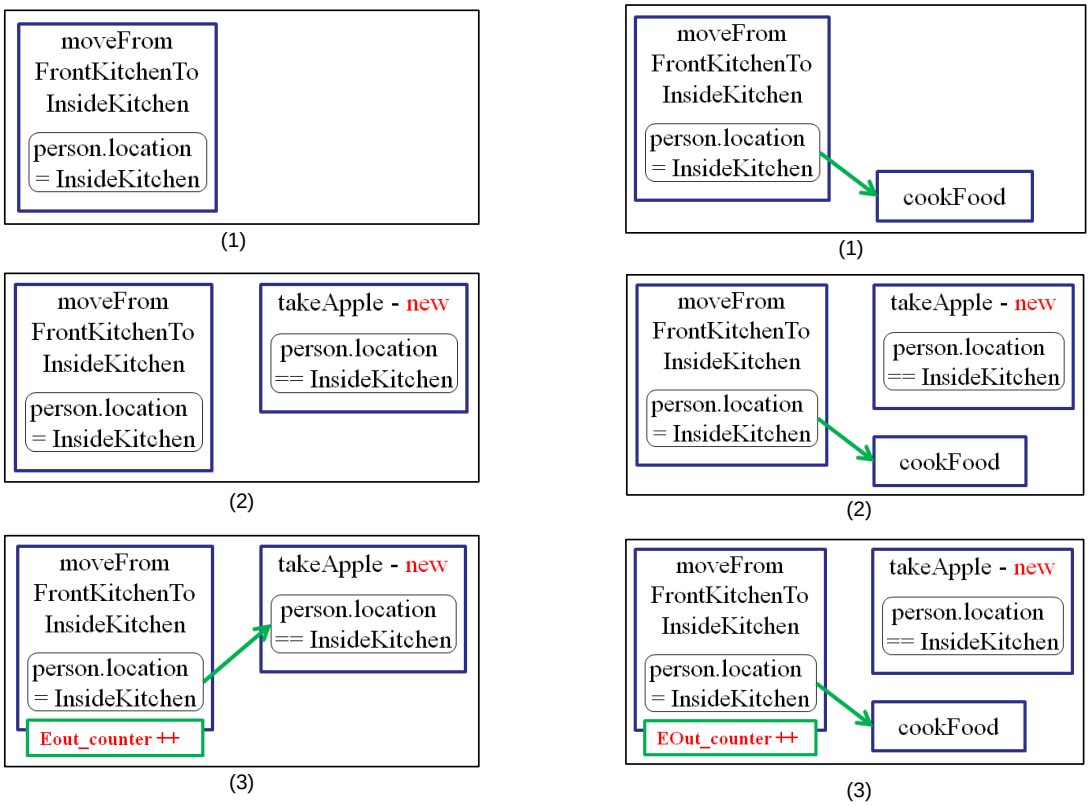
Take note that the backward updating is for all the preconditions in the new action. Besides, the "Single-In" property should hold for every precondition. Thus, although there might be more than one existing actions that have potential direct

causal relations to a precondition in the new action, only one of them can be explicitly explained.

IV) Analyzing Existing Actions without any Causal Link Getting Out

Case IV) is trivial. Let's assume that the found causal relation is "$existing\_a_2.ef_1 \Rightarrow new\_a_1.c_1$", and there is no causal link going out of a_2. Planners can just add a direct causal link connected from $existing\_a_2.ef_1$ to $new\_a_1.c_1$. Next, "EoutCounter" in $existing\_a_2$ should be accordingly increased by 1.

$existing\_a_2$, that is, there is a direct/indirect causal link between new_a_1 and $existing\_a_3.c_2$ (denoted by "$new\_a_2 \Rightarrow\!\!\!\!/\rightarrow existing\_a_3.c_2$"). In this case, there is direct or indirect causal relation between $existing\_a_2$ and new_a_1.



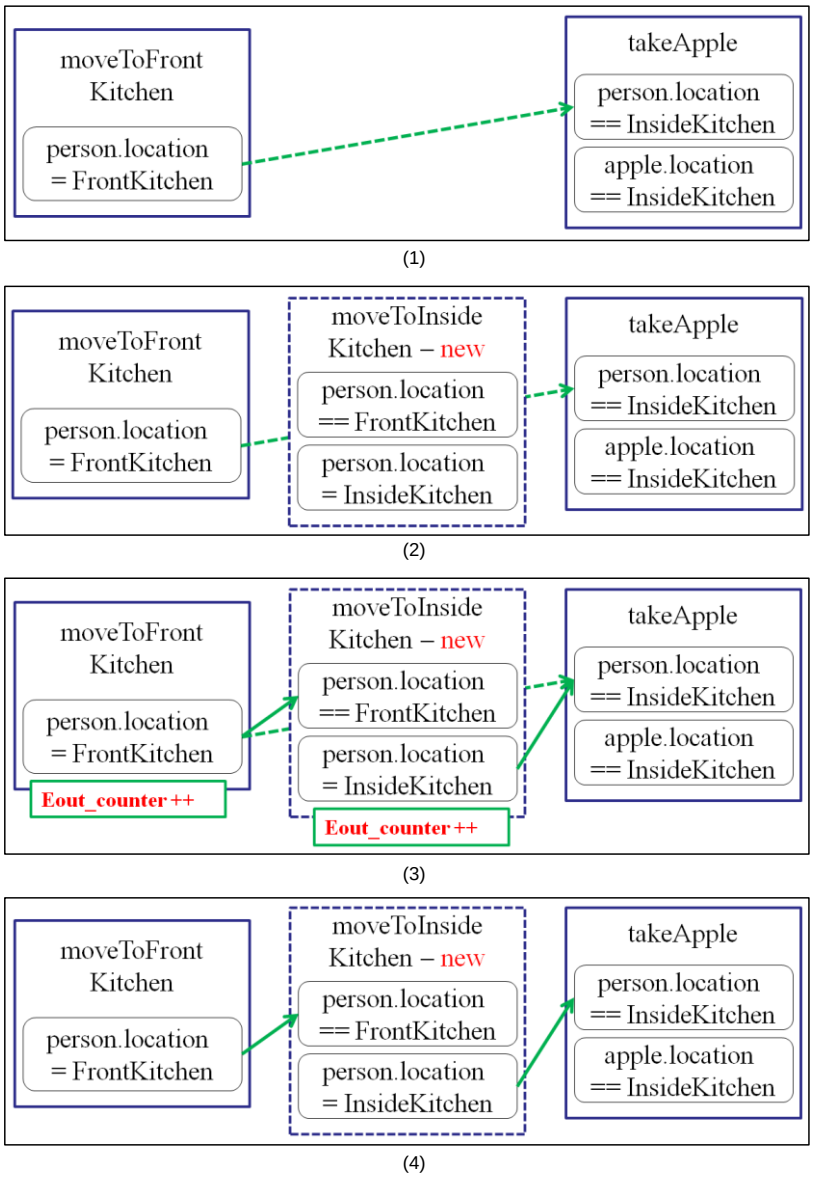
IV) Analyzing Existing Actions without any Causal Link Getting Out

V) Analyzing Existing Actions with a Causal Link Getting Out

Figure 21: Two Backward Updating Examples of Adding a New Action

V) Analyzing Existing Actions Having a Direct Causal Link Getting Out

In case V), since there is already a direct causal link going out of *existing_a*$_2$, as per "SO" property of MISO causal network, i.e., only one causal relation is allowed to go out of one action, no new causal link is added in this scenario but "EoutCounter" in *existing_a*$_2$ should be increased by 1.



VI) Analyzing Existing Actions Having only an Indirect Causal Link Getting Out

Figure 22: Another Backward Updating Example of Adding a New Action

VI) Analyzing Existing Actions Having only an Indirect Causal Link Getting Out

If there is only one indirect causal link going out of the *existing_a*$_2$, then *existing_a*$_2$ is in an action sequence to resolve an inconsistency. Let's assume the

inconsistency is because of $existing\_a_3.c_2$, then the indirect causal link can be denoted by "$existing\_a_2.ef_2 \rightarrow existing\_a_3.c_2$". If new_a_1 is added in the action sequence after If the causal relation is direct, i.e., $existing\_a_2 \Rightarrow new\_a_1$, then it can be found during the sub-process that is common for case VI) and case IV) (refer to Figure 20). After a direct causal link is added for representing it, the indirect causal relation "$existing\_a_2.ef_2 \rightarrow existing\_a_3.c_2$" will become redundant. Thus, the indirect causal link representing it needs to be removed.

If the causal relation is indirect, i.e., $existing\_a_2 \rightarrow new\_a_1$, then it can only be found during the sub-process that is common for all of the three cases (refer to Figure 20). The indirect causal link needs to be either removed or replaced by another direct/indirect causal link.

(3) Other Special Cases

Another case to be considered is "$a_1.ef_1 + a_2.ef_2 \Rightarrow a_3.c_1$". All of them are related to the same attribute. The case is very likely to occur on a numerical attribute. Thus, the research on this part can be our future work.

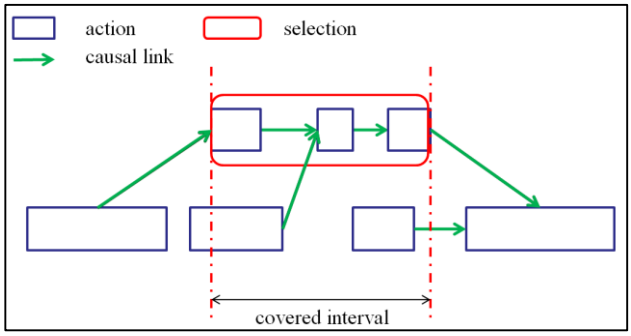
In summary of the above six cases, causal explanation in the "from" end of causal links need to be updated when plan structures change. The update process includes updating value of "EoutCounter" and analyzing whether to add a causal link between an existing action and the newly added action.

3.4.2.2 Updating when Removing Actions

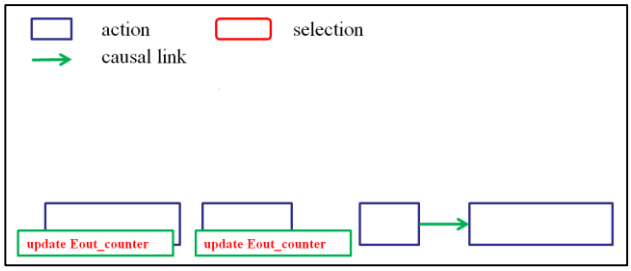
Figure 23 illustrates a scenario of updating MISO after removing a set of actions. All causal links connected to the selected set of actions also need to be removed and "EoutCounter" in their related actions that are still kept in the plan needs to be updated by reanalyzing the number of direct causal relations.

One should note that whenever an action is removed from the plan, the causal explanation in its connected action that is in the set of selected actions also needs updating. The updating in the set of selected actions is unnecessary. Thus, to further improve planning speed, the set of selected actions are grouped into a macro-action, and only actions that are not contained in the macro-actions needs updating.

Update MISO after Removing a set of Actions



(1)The selected set of actions are to be removed



(2) Update related Eout_counters after removing the selected actions and their connected causal links

Figure 23: Illustration of Updating MISO after Removing a set of Actions

3.4.2.3 Updating when Moving Actions

Cases of moving one or a set of actions can be regarded as a combination of removing them from the original covered interval and adding them to the new interval. Similarly, the set of actions that need to be moved, are grouped into a macro-action for reducing time cost for explaining causal relations inside the macro-action.

(4) Summary

The MISO causal networks are partially connected and can be generated from the enhanced plan structure. In MISO, there can be multiple causal links going into an

action, but at most one direct/indirect causal link going out of the action. Besides, all preconditions in MISO are connected with at most one causal link. Up to now, all cases that can be explained have been analyzed. The proof of correctness is introduced in the next subsection.

3.4.3 Proving Correctness of the Updating MISO Algorithms

The correctness of algorithms is very essential. In this subsection, we will prove that all properties of MISO causal network hold after using the above updating algorithms in any cases, by way of mathematical induction. As mentioned in Subsection 3.4.2, adding/removing/moving an action are the basic planning operations that can change the plan structure. Furthermore, a moving operation is a combination of an adding operation and a removing operation. Thus, we can narrow down the proof to updating algorithms after adding or removing an action. Referring to Subsubsection 3.4.4.2, the removing operations don't add any new causal links to the plan, i.e., the number of causal links that are connected with an action or a precondition will never increase by a removing. Thus, the removing operations will never violate the MISO properties under any circumstances, since all the properties of MISO (refer to Subsubsection 3.4.1 – (1)) are constrained by two upper bounds of the number of causal links related to actions and preconditions. In summary, only the updating MISO algorithm after adding a new action into plan needs to be proved.

The first step is to prove that all properties of MISO hold after the first run of updating algorithms. Since a causal link is added between two actions or between an action and a goal, the initial plan only contains initial states and goals and no action is included, and thus, no causal link is included. The only planning operation on the initial plan is adding a new action before the goal that is to be satisfied. Thus, only the operations in the forward updating MISO phase might be executed. Since there is no

causal link in the initial plan, the algorithm will follow case I) or II). In case I), only a direct causal link will be added, thus, both the "Single-Out" property of the new action and the "Single-In" property of the goal hold after running the updating algorithm. In case II), there will be only one indirect causal link added into the plan, and might be a direct causal link added between the new action and another goal. Thus, the properties also hold. In summary, the properties hold after the first run of updating algorithms.

Assuming that all the properties hold after the k^{th} run of updating algorithms, we should prove that they still hold after the $k+1^{th}$ run.

Let's first consider the "Single-In" constraint for the existing preconditions. Only the three cases that are in forward updating phase have the operation of adding new causal links and connecting it between the new action and an existing precondition. If there is no causal links connected with the precondition that is to be satisfied, then the algorithm will follow the branches for case I) or II). One new direct/indirect casual link will be connected with the precondition. Thus, the property holds for the precondition. However, in case II), the algorithm might also add a new direct causal link between the new action and another precondition that might already has an indirect causal link "In". Thus, the indirect causal link is redundant and the property doesn't hold for this precondition after forward updating phase. Furthermore, the backward updating phase also doesn't change the state. However, there is a "consistency check" phase after backward updating (refer to the last common sub-process in Figure 20) that removes the redundant indirect causal link, the property then holds again for the precondition. Moreover, case III) doesn't contain any adding causal links operations before the updating algorithm goes into either case I) or case II). Since the property for all existing precondition holds after running the updating

algorithm both in case I) and case II), it also holds after running the updating algorithm that goes into branch for case III). In summary, the property holds for all existing preconditions in the plan after running the algorithm.

Next, let's consider the "Single-In" constraint for all the preconditions in the new action. Only the backward updating phase will affect these preconditions. Furthermore, the first choice point in Figure 20 ensures that the "Single-In" property always holds for every precondition in the new action after the backward updating phase.

In summary, the "Single-In" property of all preconditions still holds in the new MISO after the $k+1^{th}$ run of the updating algorithms. The next property needs to be checked is the "Single-Out" property of all the actions in the new MISO.

The "Single-Out" property of the new action needs to be checked only in the forward updating phase, while the property of the existing actions needs to be checked only in the backward updating phase. In forward updating phase, all branches contain at most one operation for adding a new direct causal link and at most one operation for adding a new indirect causal link that are out of the new action. Thus, the property of the new action holds after the $k+1^{th}$ run of the algorithm. On the other hand, only in case IV) and VI), the updating algorithm has one operation of adding a direct casual link out of an existing action under the condition that there isn't any direct causal link goes out of the existing action, i.e., there is at most one indirect causal link out of the existing action since all properties hold before adding the new action. Thus, the "Single-Out" property of all existing actions also holds after the $k+1^{th}$ run of the algorithm.

In summary, by the mathematical induction, we have proved that the updating MISO algorithm is correct. The way of utilizing explanations to facilitate searching will be introduced in the next subsection.

3.5 Exploiting Causal Explanations

Adding or detecting the explanation is not an end in itself, and the real reason for the planners to use explanations is to improve their performances.

When searching for the successor plans, if the planner finds that changing an existing action can somehow repair the plan, and the existing action is connected by some causal links, then planners can exploit causal links to go further, and analyze the next successor plan. The planner faces three problems when exploiting causal explanations: 1) whether to exploit all causal links connected to the selected action; 2) which causal link to follow; 3) when to stop going deeper.

The first problem is easy to answer. For a complex problem or if the problem size is large, it is not promising to exploit all causal links. Exploiting only parts of the search space to reduce time expense is also a reason for planners to use local search.

The second problem is due to the fact that every action can have multiple causal links coming in, and one direct causal link and one indirect causal link going out. Based on the consideration in the first problem, we can choose some of the branches to go deeper. Some heuristics are developed to address the problem. The heuristics are to be introduced later.

Furthermore, if causal networks are strongly connected, it is unwise to go too deep, thus the searching has to stop at some point. To address this problem, we can define some stopping criteria, and will be introduced in the second subsection.

3.5.1 Exploiting Heuristics based on Causal Explanation

Three heuristics are developed to solve the second problem described above.

(1) Forward Exploiting Heuristic

Figure 24 illustrates forward exploiting heuristic in MISO causal network. It exploits MISO in the same direction as direction of causal links.

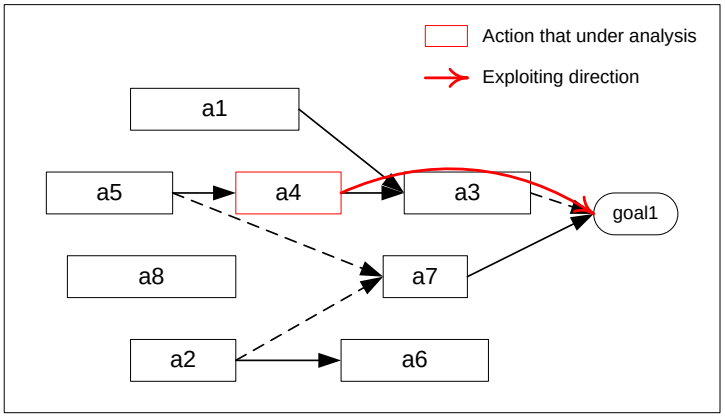


Figure 24: An Example of Forward Exploiting Causal Explanation Heuristic

Suppose action "a4" (in red border) is under analysis, one of its preconditions is unsatisfied and removing the precondition from current plan can improve the plan quality. Action "a3" and "goal1" is sequentially followed by "a4" along causal links. Then we can make such an understanding that removing "a4" (in order to remove its unsatisfied precondition) will make a currently satisfied precondition in "a3" become inconsistent, that is, "a3" and "goal1" will become infeasible subsequently. This subsequent result will reduce the plan quality. Thus, it is reasonable to remove the set of actions that are followed by "a4" and subsequently connected by causal links. Similar exploiting and operation can be done on the set of actions, if moving "a4" to another time points can improve plan quality. In summary, it is promising and reasonable to do forward exploiting.

For forward exploiting, there are three cases that might be encountered when an action is exploited, in terms of number of causal links going out of the action. The

first case is when an action that has no causal link coming out is exploited, like “a6” in Figure 24. The exploiting algorithm stops when this case is encountered. The second case occurs when an action that has only one causal link going out, like “a4” and “a3” in Figure 24. This case is trivial, since there is only one branch that can be further exploited. The final one is like action “a5” in Figure 24. Our choice is to exploit the direct causal link according to an assumption that direct causal relations are stronger than indirect ones, because indirect ones might be broken and substituted by a set of direct causal links associated with a set of actions (refer to updating algorithm in the previous section).

(2) Backward Exploiting Heuristics

Backward exploiting heuristics exploit MISO in the opposite direction of causal links. Figure 25 illustrates the heuristic.

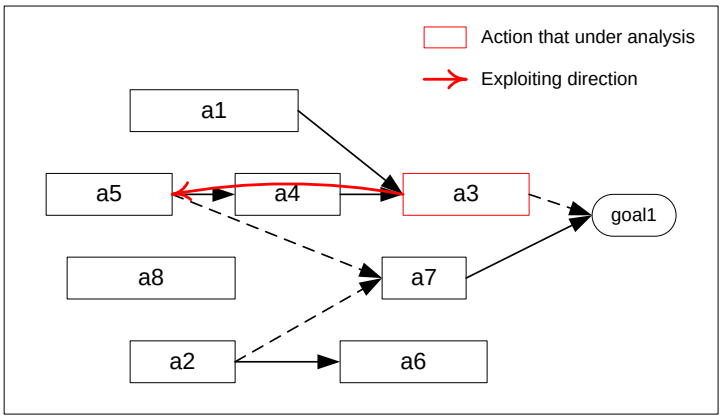


Figure 25: An Example of Backward Exploiting Causal Explanation Heuristic

Suppose action “a3” is currently under analysis. Action “a1” and “a4” are both connected to “a3”. If “a3” has a precondition unsatisfied, then removing the precondition can improve the plan quality, and it can be achieved by removing “a3”. As can be seen in Figure 25, “a1” and “a4” exists to ensure occurrence of “a3”. If “a3” is removed, then it is meaningless for them to exist in the plan. Thus, it is reasonable

to remove the set of meaningless actions that would occur before “a3”. Similarly in case “a3” is supposed to moving to another time point.

A key problem in this case is that there can be multiple causal links going into an action, like “a3”, to decide which causal link to exploit is a problem for the planners when such an action is currently exploited. Planners can exploit all of branches or heuristically select one of branches to exploit. By comparison, far more actions and causal links will be exploited by using the first method than using the second one, in terms of MISO’s tree structure. Thus, the second method is preferred in this research.

Two selecting branch heuristics are used in backward exploiting. One of them is to randomly exploit one of causal links coming in the action. Another one is based on an assumption that the worst inconsistency is to be repaired first, the first action links to action “a3”, let’s say “a4”, has the most significant effect on “a3” than effects of other actions. Thus, exploiting the first branch is more reasonable. To use this selecting heuristics, pointers inside an action can be stored according to order of causal links linking to the action. Note that, in any case, direct causal links has higher priority for exploiting than indirect ones.

(3) Hybrid Exploiting Heuristic

As described in the previous two cases, both forward searching and backward searching are reasonable. When exploiting forwards and encounter an end, then the planner can also turn back and search another branch coming into the end, as shown in the illustration of hybrid exploiting in Figure 26, “a5” is such an “end”.

On the other hand, hybrid exploiting heuristic can be implemented in two ways. First go forward and then go backward, or vice versa.

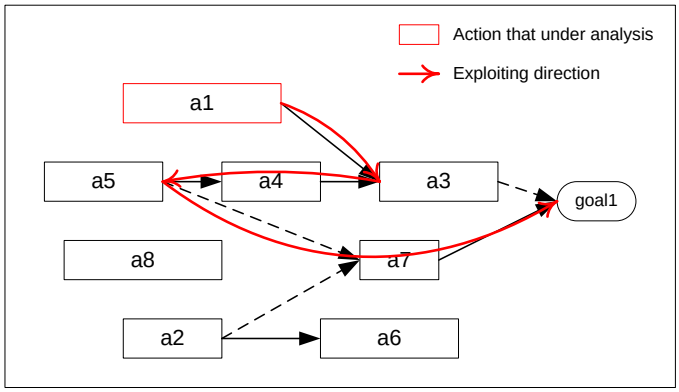


Figure 26: An Example of Hybrid Exploiting Causal Explanation Heuristic

3.5.2 Stopping Criteria

It is easy to imagine that causal networks can be very huge and complex, because lots of real-world domains are complex. Intuitively, if the number of actions connected by causal links is large, it is unwise to suggest the same change to the whole set of actions in the same tree, when one action in the tree is suggested to be removed or moved to another time point. Thus, setting proper stopping criteria is essential for improving planning performance.

We propose some ideas about when to stop and they are listed as follows:

1) # of traced actions < nA_ub
2) # of traced action levels < nAL_ub
3) *EoutCounter* >= e_lb
4) *Time limitation*. In real-time applications, timely responsiveness is a key satisfaction/optimization criterion.

The criteria 1) and 2) respectively set upper bounds for the number of actions and levels that heuristics can exploit. The upper bounds can be initially set. Moreover, it can be a fixed value or can be dynamically adjusted during planning. In the current stage, we only consider the first case. The dynamically adjusting case requires the usage of online learning techniques and would be our future work. Note that, the

bounds might be domain dependent. Criteria 1) and 2) have different effects on exploiting algorithms only in hybrid exploiting case. Stopping criterion 3) is another strategy that making use of "*EoutCounter*" in explanation structures. The counter shows that the action is satisfying more than one precondition, and this indicates that it is not reasonable to make a synchronous change to this action as the change to the first exploited action. The last criterion is to set time limitation to exploiting algorithms.

In summary, all of above stopping criteria can be used by exploiting algorithms, and exploiting algorithms stop when any of criteria is satisfied.

3.6 Summary

Up to now, the systematic introduction of the proposed approach is finished. The implementation and evaluation related issues will be introduced in Chapter 4.

Chapter 4 Prototype Implementation

Our experimental setup is implemented on a planning system named Crackpot. Crackpot is a local-search-based planning system evolved out of the Excalibur system [43]. It can be applied in many applications, such as storytelling, game development, etc. We will introduce its implementation in detail in this chapter. Firstly, we give an overview of Crackpot and show its workflow. After that, we introduce some structures and concepts closely related to explanations, such as action structure.

4.1 Crackpot Overview

In this section, the high level structures of Crackpot will be introduced.

4.1.1 Crackpot Architecture

Crackpot architecture is modeled as shown in Figure 27. Six components are designed to manage different kinds of tasks. Table 3 lists their functionalities.

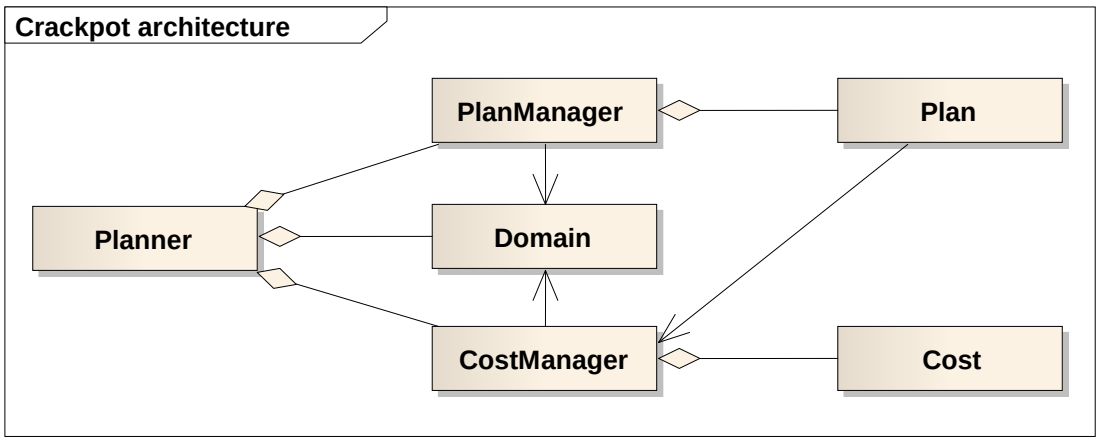


Figure 27: Architecture of Crackpot

Plan and *Cost* in Crackpot keep essential data structures. Plan contains actions, causal structures, objects associated with their components attributes and actuators, and so on. For example, “package1.location” is an attribute of the object “package1” in time, such as “package1.location == Depot1” in duration [2, 7). The usage of a

truck is an actuator. If a condition is "package1.location == Depot2" during [3, 5), then the condition is inconsistent, a cost will be added to the unsatisfied condition. Anything that can compute costs is called *CostCenter* in Crackpot, like the attribute "package1.location" in the above example. Actuators can also compute costs if their usages are overlapped. The functionality of CostCenters is computing costs and then reporting them to *CostManager*. Because cost computation in different CostCenters might be different, CostManager generates costs that come from different CostCenters in a normalized way and make use of them to guide planning improvement. Similarly, plan structure is managed by *PlanManager*.

Component	Functionality
Planner	Given a description of the initial state of the world and the desired goals, and domain description, planner takes charge of the overall planning flow to iteratively repair the plan.
Domain	Domain contains a description of the environment, like a set of possible actions, a set of constraints, objects, and so on.
Plan	It contains all plan related information, such as projected actions, attribute values, actuator usages, and causal structure. Plan quality can be evaluated in terms of costs.
PlanManager	It makes changes to the plan and maintains plan structure after realizing change.
Cost	contain cost instances and cost related structures, and they can be used to guide the repairing process
CostManager	manages cost related things

Table 3: Components of Crackpot and Their Functionalities

Domain contains a description of the environment. *Planner* controls overall planning process. It controls CostManager and PlanManager to make use of domain information to iteratively repair plan. The initial plan only contains the initial state and the goals given. When some plan components, such as actions, objects, are added into the plan or are deleted from the plan or get updated, the plan structure needs to be updated and a new plan will be generated. The changes above are called *planChange* in Crackpot.

There are 3 kinds of essential planChanges closely related to explanation and its related methods. They are named AddActionChange, MoveActionChange and DeleteActionChange respectively, and their names are intuitive.

4.1.2 Overall Planning Workflow using Causal Explanation

The detailed repairing process of Crackpot is shown in Figure 28. In every iteration, the planner delegates the CostManager to select one of the costs to repair. Next the corresponding CostCenter, where the selected cost comes from, would be asked to suggest a set of changes to the plan (plans that can be generated by making the changes the current plan are called successor plans of the current plan). The successor plans might be better than the current plan according to a cost function. After receiving those plan changes, the costManager selects one or multiple of them, and delegates the PlanManager to update the plan using the selected plan changes. At this step, projection of some plan components in the current plan will be updated. The projection updating will result in inconsistencies of some related costs, thus the CostManager will be delegated to update the corresponding costs. If the new plan quality is not good enough and the planning time is not used up, then the planner will start another iteration to repair the newly generated plan.

The exploiting algorithm is used by CostCenters for suggesting better plan changes, while the updating algorithm is used by the PlanManager after corresponding CostCenters realize their related components changes. They are highlighted in red color in Figure 28. Take note, Crackpot has a heuristic framework that it is easy to be configured to enable using an exploiting heuristic or not. Those exploiting heuristics are integrated into this framework and are enabled when using causal explanations. This makes the usage of hybrid heuristics that can also include non-explanation-based heuristics. This hybrid scheme is currently used in Crackpot.

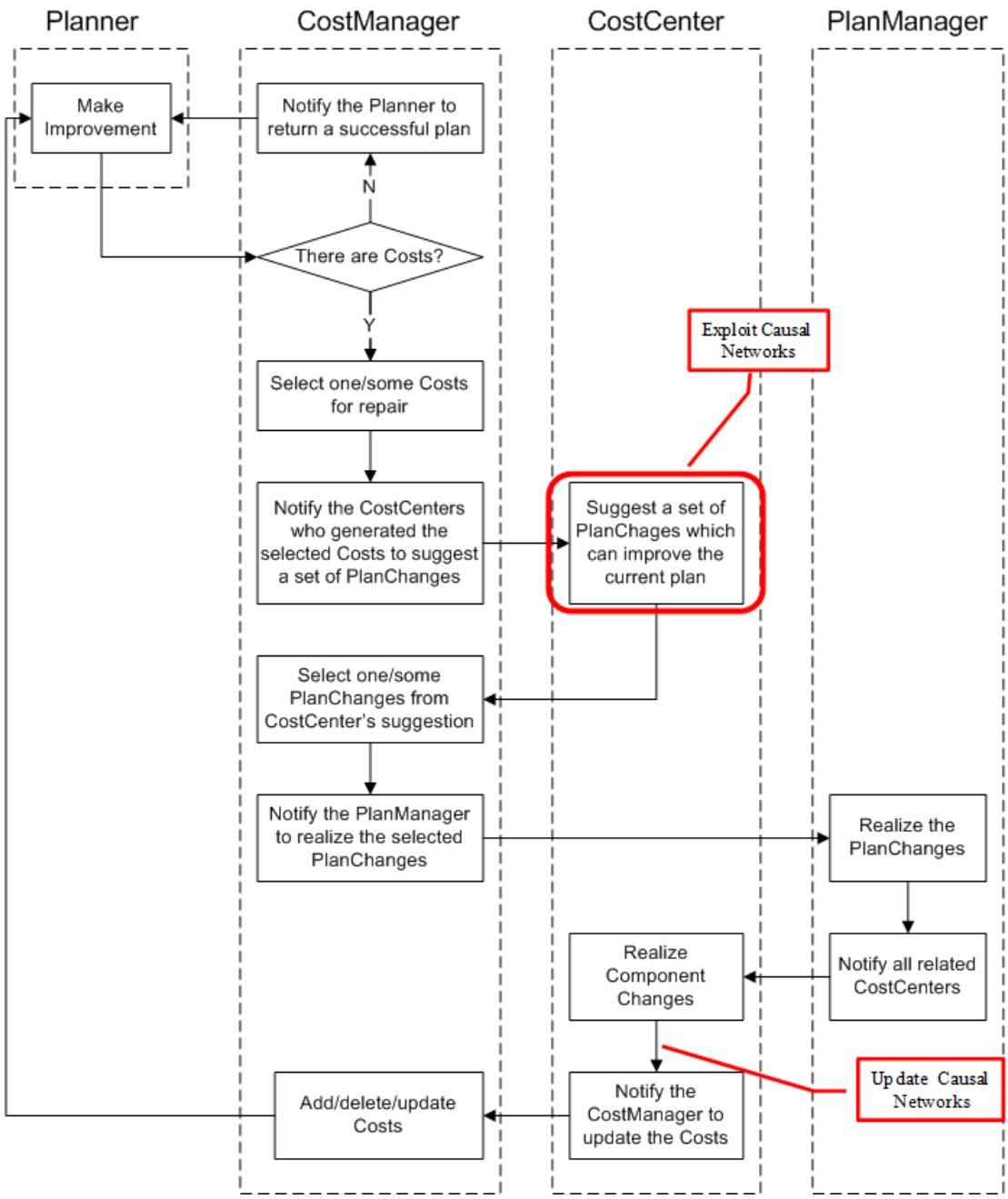


Figure 28: Overall Flow of Planning in Crackpot

4.2 Introduction of Action Compositions

Costs introduced by different costCenters have different weights to represent their significances comparing to costs from other costCenters. For example, for the planner, the cost due to a resource's overlapping might be more significant than another cost due to the difference between a numerical attribute's current value and its

desired value. To differentiate different kinds of costs, the structure of an action in Crackpot is designed as shown in Figure 29.

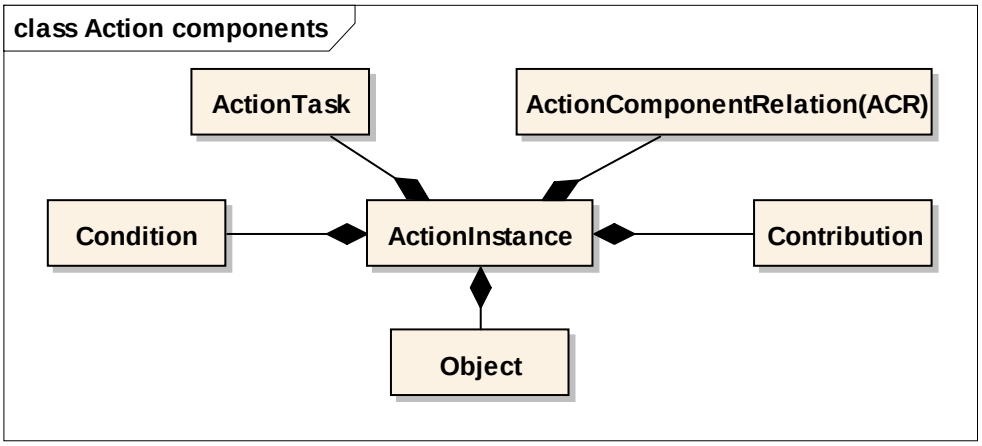


Figure 29: Action Structure

Action is composed of five main components. In every action there is a set of conditions contributions and ACRs (Action Component Relations). Each condition or contribution is related to an attribute, while an ACR is about the relationship of multiple attributes.

4.2.1 Condition

Take attribute "package1.location" as an example, a condition related to the attribute can be "package1.location == Depot1" or "package1.location != Depot1" during time [2, 7). If the condition is unsatisfied in the duration, there will be costs accordingly. Crackpot currently has the above two type of conditions. They are only related to one specific condition and called classical conditions in this research.

4.2.2 Action Component Relation

Take action "load(package1, truck1)" as an example. If "package1.location" and "truck1.location" are represented as not ground attribute variables, then the action's condition can be modeled as "package1.loation == truck1.location @ t" or "package1.location" is contained by "truck1.location". This kind of conditions

describes relationships (or constraints) between different attribute variables. Those attribute might belong to different objects (like “package1” and “truck1”). To differentiate them with classical conditions, this kind of condition is called *ActionComponentRelation* (ACR) in Crackpot. Similar to classical condition, when ACR is not satisfied, there will be cost.

4.2.3 Contribution

A *contribution* is a state transformation related to an attribute, like “truck1.location == Depot1 @ $t_1 \Rightarrow$ truck1.location == Depot2 @ t_2”. If the value of the attribute at t_1 equals to the preceding value in the transformation (contribution), then the contribution will be applied. The value of the attribute will be updated subsequently at t_2 by realization of state transformation. A contribution inside an action is used to satisfy a/an condition/ACR in another action. For example, “truck1.location == Depot2 @ t_2” is a condition of action “unload(package1, truck1, Depot2) @ t_2”, which can be satisfied by an action having the contribution in the above example, like “move(truck1, Depot1, Depot2) @ t_1”.

4.2.4 Other Components in Action

Object is the component to operate an action or to be operated in an action. For example, the objects related to action “eat(apple)” are a person (player) and an apple.

ActionTask is to be added to an actuator. For example, “person.hands” is an actuator, and action task in “eatApple” is one of action tasks needs to be done by “person.hands”. If “person.hands” is currently in use, there will be overlapping cost because the “person.hands” can be used by only one action task at one time.

We won't introduce these 2 parts in detail, because they are not directly related to explanation related work.

4.2.5 Summary

An action might have only one or multiple conditions/contributions. For example, the action "open(door)" has following conditions: "the door is locked", "the player is outside the door", and "the key of the door is in hand of the player", and the following contributions: "make the door from locked to unlocked", and "make the door from closed to open". In the example, the lock state and the open state are two values of attribute "door.state", and "player.location" and "key.location" are the attributes of the player and the key, respectively.

In summary, some of general knowledge about local-search-based planning is introduced based on Crackpot planning system. So up to this point, we can have an overall picture of local-search-based planning process.

4.3 Explanation Structure related to Actions

Figure 30 illustrates the relationships between action, causal explanation and causal link. Causal links are stored in plan and used to connect two of actions, while the causal explanation structure is stored in every action and serves for causal network exploiting algorithm and updating algorithm. Figure 31 is a screenshot of the GUI of Crackpot with updating and exploiting MISO algorithms integrated.

4.4 Evaluations

Some tests were run on a computer with the following configuration:

- Hardware: Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.4GHz; 4GB RAM;
- Software: Windows Vista™ Business (SP2);

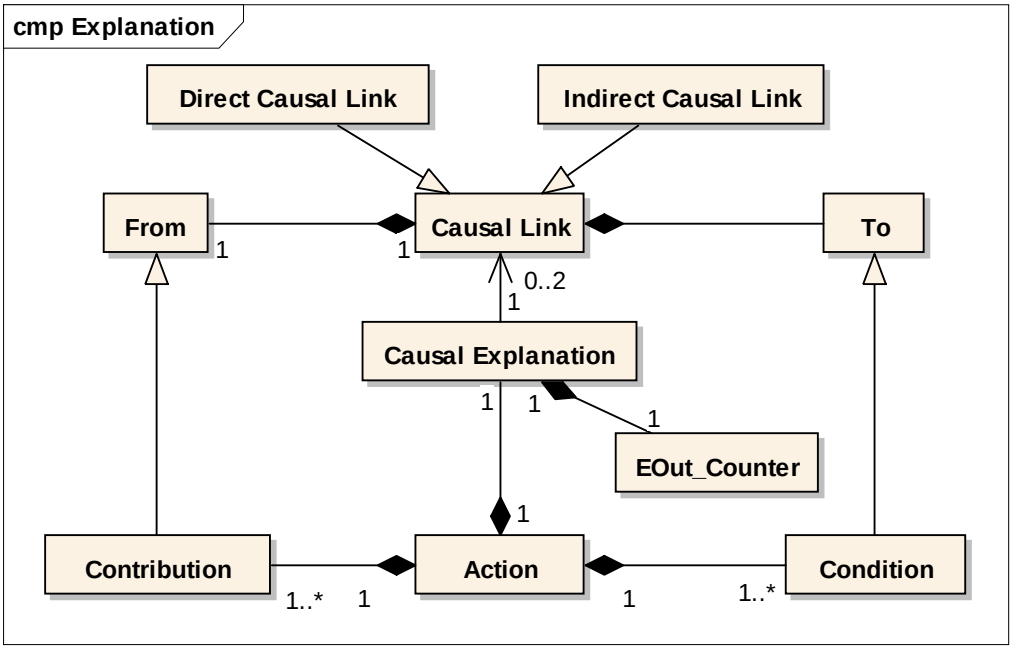


Figure 30: UML Model of Causal Link, Causal Explanation and Action

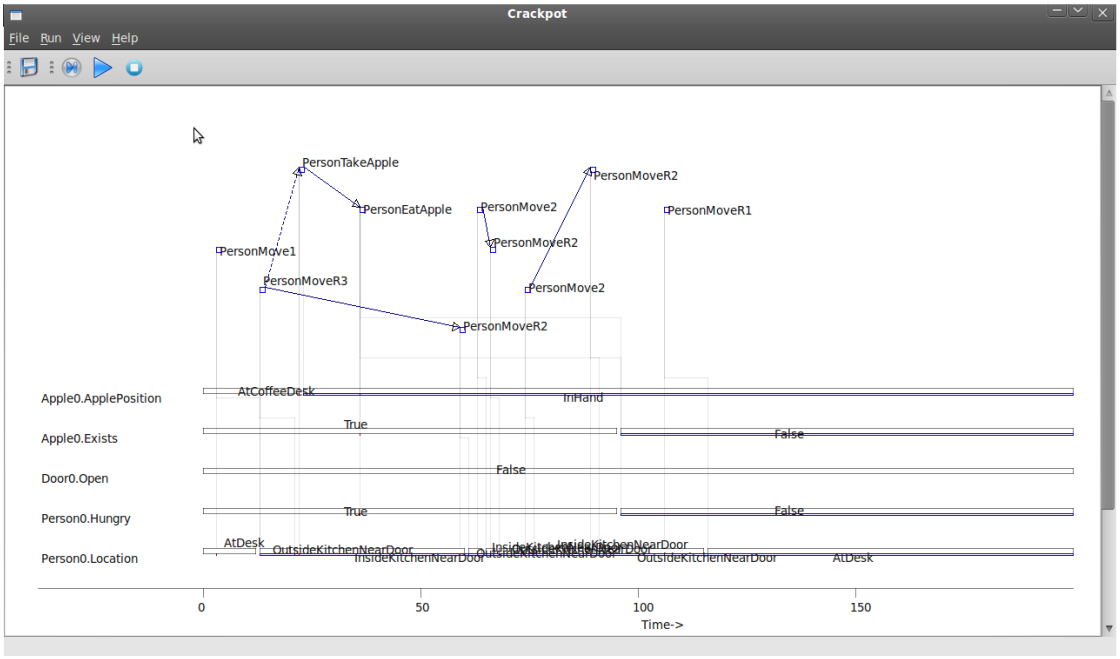


Figure 31: A Screenshot of the Enhanced Plan Structure in Crackpot with Updating and Exploiting MISO Algorithms Integrated

On the other hand, the testing was run on a simple BlockWorld domain that the planner can do the following four types of actions: picking up a block from the table, putting it on the table, stacking one block onto another one and unstacking one block that is on another block. Initially, there are four blocks named A, B, C and D on

the table. The planner is required to reach the state that B is on A, C is on B and D is on C. Furthermore, there is also an optimazition requirement that the planner should get an optimal plan that has lowest costs. The cost is computed according to the total number of actions and whether the goal state is reached. The planner was set to run 180 seconds every time. Based on the above experimental environment, three planning cases were tested: 1) without causal explanation; 2) using explanation and exploiting maximum 2 actions if our exploiting heuristics are used in one iteration; 3) using explanation and exploiting maximum 3 actions if exploiting heuristics are used in one iteration. Take note that, in case 2) and 3), the planner used hybrid heuristics, including forward and backward exploiting heuristics that are introduced in subsection 3.5.1 (refer to subsection 4.1.2 for how they are integrated into the planning), and non-explanation-based heuristics.

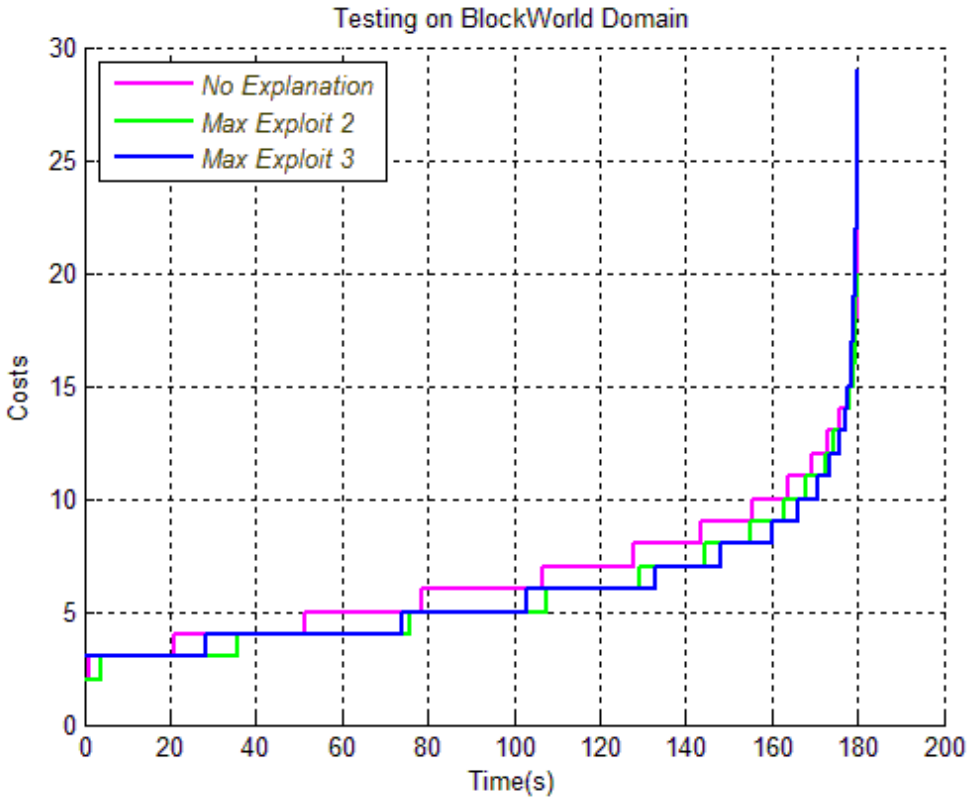


Figure 32: MISO Performance on BlockWorld Domain

In this research, total costs in every iteration are used for evaluation. Figure 32 and Figure 33 illustrates performance comparison of the three cases.

To do a statistical evaluation, the planner did 30 runs for each case. The two explanation-based exploiting heuristics were used 674 and 710 times in average in case 2 and 3, respectively; while the other heuristics were used around 23600 times in total. In every iteration, we recorded the total cost of the current plan and how long the current iteration took. In order to have a better comparison, costs are sorted in ascending order. Next, we sum up the total time duration when the planner has the same costs for every run, and get average time duration for every cost value. As can be seen from Figure 32, in cases that use explanations, we can easily compare that how long it totally took when the plan had costs less than a cost value. For example, if we want to know how long the planning was taken when the cost of the plan was less than 5. The case 1) took only 20 seconds while the other cases took longer. It means the planning took longer time making lower quality plans in case 1). Thus, we can conclude from Figure 32 that by using explanations, the planning can get better performance. Furthermore, the third case has better performance than the second one. It means that the maximum number of actions being exploited by using causal explanations also affects the planning performance. However, this research is concentrated on performance comparison between planning with and without using causal explanations. More detailed research on how the maximum exploiting numbers affect the planning performance is our future work.

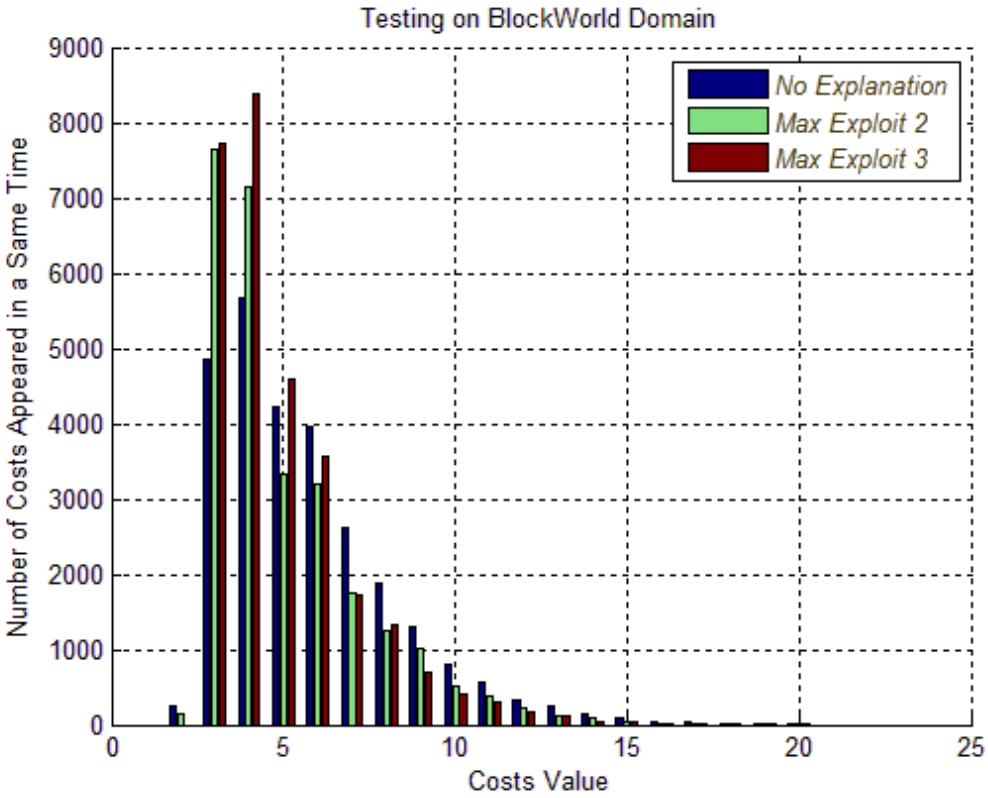


Figure 33: Bar Graph of MISO Performance on BlockWorld Domain

Figure 33 shows a different view of the experimental results that is the same as in Figure 32. The number of iterations that has the same cost values is counted for a set of cost values. All the three cases can be easily compared for each cost value. As can be seen from Figure 33, most of iterations got total cost value between 3 and 5. Furthermore, the smaller the cost value is, the more iterations were encountered in cases that using explanations. This means the planning fluctuated in sub plan spaces that have higher plans when using explanations.

Finally, evaluations on other kinds of domains are our future work.

Chapter 5 Conclusion and Future Work

The key idea of the thesis is to keep some causal information in a straightforward way, such that the planners can easily utilize this information to facilitate the search in order to improve their planning performances. In this thesis, we proposed a novel technique using causal explanation structures to retain some causal information acquired during planning. To improve the planning performance by utilizing this causal information, we designed an explanation structure for composing Multiple-In-Single-Out (MISO) causal networks, and developed some updating and exploiting MISO algorithms. The updating algorithms update the MISO causal networks whenever the plan is changed, and they were proved to be correct in this research. The exploiting algorithms exploit the MISO causal network to dynamically group some of the actions into a macro-action, and the planners can then operate on this macro-action like on a normal action. Our approach is promising to speed up planners that have loose plan structures using local search approaches to create plans, due to the potential benefits from the usage of macro-actions.

5.1 Future Work

Although we have successfully fulfilled our initial objective, there are still some limitations which can be improved or extended. In the thesis we have developed two algorithms for updating and exploiting MISO causal networks by using causal information between actions, and the current research is based on symbolic attribute. Besides, the prototype implementations were based on Crackpot. There are several directions we can focus on in the future:

- In terms of empirical evaluation, our approach is to be evaluated on more planning systems.

- Extend explanation theory to rules. Some planning systems have the capability of handling rules, like Crackpot. Similar to actions, rules have conditions and contributions as well. There will be causal relations between rules or between rules and actions.
- Numerical attributes are more complicated than other attributes, such as symbolic attributes and Boolean attributes. Contributions on numerical attributes can be a range of numerical value. Besides, multiple contributions might collectively achieve a condition on numerical attribute. Thus, causal information related to numerical attributes is more complex than that related to other attributes. It is common that a real-world planning domain contains numerical attributes related actions and rules. Crackpot also can handle numerical things.
- Furthermore, currently every causal explanation inside an action will be removed if the action is removed from the plan. However, some of them might be reused for similar actions. Thus, the planning performance might be further improved if learning techniques can be used to generate those useful causal explanations.

5.2 Schedule of Master Study

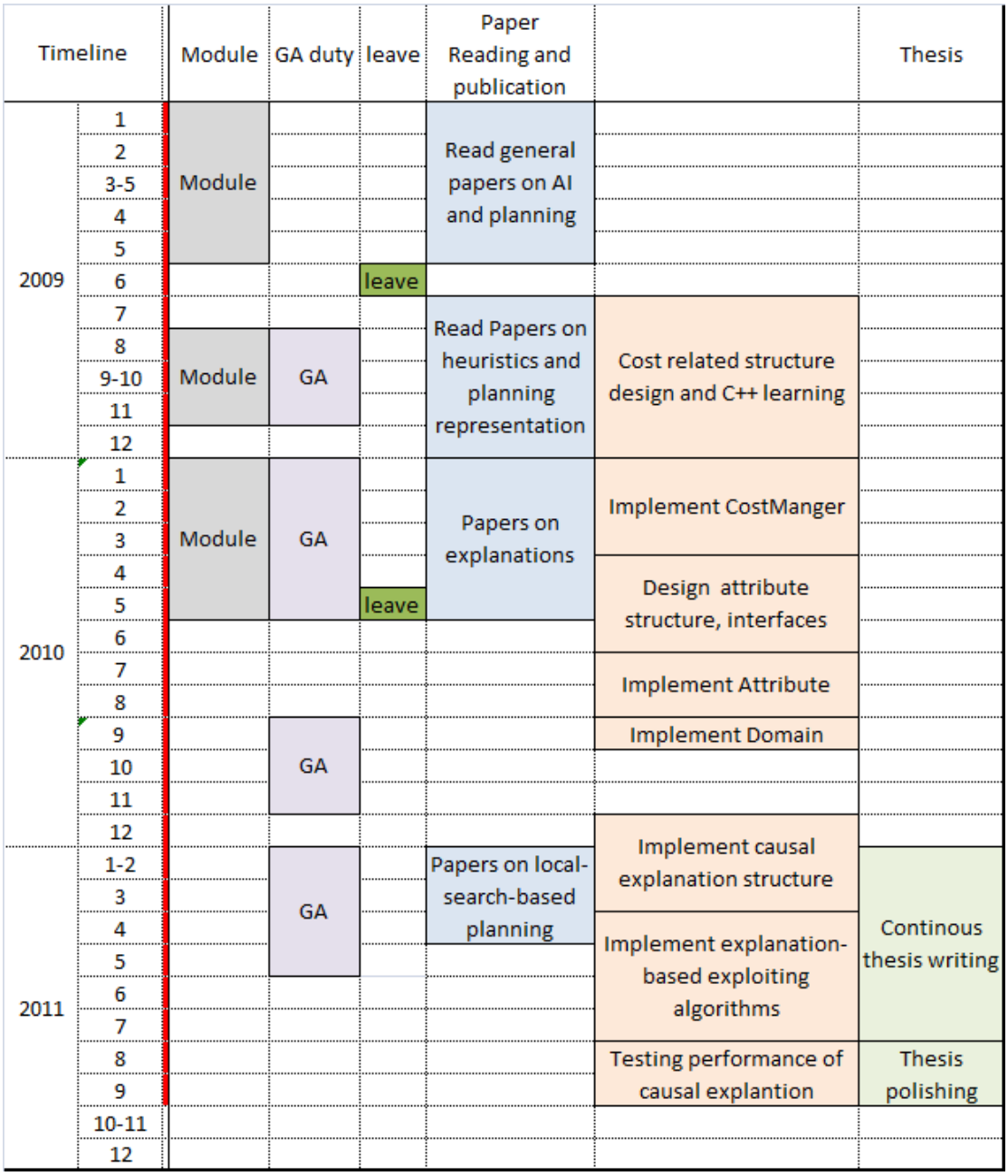


Figure 34: Schedule of M.Eng Study

Bibliography

[1] Stuart J. Russell; Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed.: Prentice Hall, 2009.

[2] Alfonso Gerevini and Ivan Serina, "LPG: A Planner Based on Local Search for Planning Graphs with Action Costs," in *International Conference on Automated Planning and Scheduling/Artificial Intelligence Planning Systems (ICAPS/AIPS)*, 2002.

[3] Alexander Nareyek. (2005, May) EXCALIBUR: Adaptive Constraint-Based Agents in Artificial Environment. [Online]. http://www.ai-center.com/projects/excalibur/documentation/

[4] Emile Aarts and Jan K. Lenstra, Eds., *Local Search in Combinatorial Optimization*. New York, NY, USA: John Wiley & Sons, Inc., 1997.

[5] Malik Ghallab, Dana Nau, and Paolo Traverso, *Automated Planning Theory and Practice*, 1st ed., Denise E. M. Penrose, Ed.: Morgen Kaufmann Publishers, 2004.

[6] Judea Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984.

[7] Subbarao Kambhampati, "Planning as Refinement Search: A unified framework for comparative analysis of Search Space Size and Performance," Technical Report 93-004 1993.

[8] José Luis Ambite and Craig A. Knoblock, "Planning by Rewriting," *Journal of Artificial Intelligence Research*, pp. 207-261, Sep. 2001.

[9] B. Selman, H. Kautz, and B. Cohen, "Noise Strategies for improving local search," in *Proceedings of AAAI*, 1994.

[10] Stefan Voss, "Meta-heuristics: The state of the art," in *Local Search for Planning and Scheduling*.: Springer-Verlag, 2001, vol. 2148/2001 of Lecture Notes in Computer Science, pp. 1-23.

[11] Alexander Nareyek, "Using Global Constraints for Local Search," in *Constraint Programming and Large Scale Discrete Optimization, American Mathematical Society Publications, DIMACS Volume 57*, 2001, pp. 9-28.

[12] Mohamed TOUNSI and Philippe DAVID, "Local Search Algorithm to Improve the Local Search," in *Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence(ICTAI'02)*, 2002.

[13] Fikes, R. E.; Nilsson, N., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.," in *Artificial Intelligence*., 1971, pp. 189-208.

[14] Patrik Haslum, "Improving Heuristics Through Relaxed Search - An Analysis of TP4 and HSP*a in the 2004 Planning Competition," *Journal of Artificial Intelligence Research (JAIR)*, vol. 25, pp. 233-267, 2006.

[15] J. Orkin, "Three States and a Plan: The A.I. of F.E.A.R.," in *Game Developers Conference (GDC)*, 2006.

[16] Scott Penberthy Ibm, J. Scott Penberthy, and Daniel S. Weld, "UCPOP: A Sound, Complete, Partial Order Planner for ADL," in *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, Cambridge, MA, 1992, pp. 103-114.

[17] Xuanlong Nguyen and Subbarao Kambhampati, "Reviving Partial Order Planning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[18] R. Michael Young; Martha E. Pollack; Johanna D. Moore, "Decomposition and Causality in Partial-Order Planning," in *2nd International Conference on AI Planning Systems (AIPS)*, 1994, pp. 188-193.

[19] Malte Helmert, "A Planning Heuristic Based on Causal Graph Analysis," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2004, pp. 161-170.

[20] Subbarao Kambhampati, "Comparing Partial Order Planning and Task Reduction Planning: A preliminary report," in *AAAI-94 Workshop on Comparative Analysis of Planning Systems*, 1994.

[21] Santi Ontañón, Kinshuk Mishra, Neha Sugandh, Ashwin Ram, "ON-LINE CASE-BASED PLANNING," *Computational Intelligence*, vol. 26, no. 1, pp. 84-119, Feb 2010.

[22] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, J. William Murdock, Dan Wu, Fusum Yaman, "SHOP2: An HTN Planning System," *Journal of Artificial Intelligence Research 20*, pp. 379-404, 2003.

[23] (2005, Feb) SHOP. [Online]. http://www.cs.umd.edu/projects/shop/description.html

[24] Hoffmann, J. ; Nebel, B., "The FF Planning System: Fast Plan Generation Through Heuristic Search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253-302, 2001.

[25] F. Heider, "The Psychology of Interpersonal Relations," in *volumeXV of Current Theory and Research in Motivation*. New York, 1958.

[26] David B Leake, *Evaluating Explanations: A content Theory*. Bloomington, Indiana: Lawrence Erlbaum Associates, Inc., 1992.

[27] Peter Jackson, "Designing for Explanation," in *Introduction to Expert System(3rd Edition)*.: Addison Wesley Longman Limited 1999, 1998, pp. 294-315.

[28] K. Surech and K. Subbarao, "Learning Explanation-Based Search Control Rules For Partial Order Planning," in *Proc. 12th Natl. Conf. on Artificial Intelligence(AAAI-94)*, 1994.

[29] Kristian J. Hammond, "Explaining and Repairing Plans that Fail," *Artificial Intelligence*, pp. 173-228, 1990.

[30] Kristian J. Hammond, "Learning to Anticipate and Avoid Planning Problems through the Explanation of Failures," in *National Conference on Artificial Intelligence (AAAI)*, 1986.

[31] N Jussien and S Ouis, "User-friendly Explanations for Constraint Programming," in *Proceedings of the Eleventh International Workshop on Logic Programming Environments(WLPE'01)*, 2001.

[32] Guillaume Rochart, Eric Monfroy, and Narendra Jussien, "MINLP Problem and Explanation-based Constraint Programming," in *4th Workshop on Cooperative Solvers in Constraint Programming (COSOLV'04)*, Toronto, 2004.

[33] Narendra Jussien and Vincent Barichard, "The Plam System: Explanation-based Constraint Programming," in *Proceedings of Techniques foR Implementing Constraint programming Systems (TRICS), a post-conference workshop of CP*, Singapore, 2000, pp. 118-133.

[34] M.A. Hakim Newton, John Levine, Maria Fox, and Derek Long, "Learning Macro-Actions for Arbitrary Planners and Domains," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2007.

[35] I. Murugeswari and N. S. Narayanaswamy, "Tuning Search Heuristics for Classical Planning with Macro Actions," in *Proceedings of the twenty-second International FLAIRS Conference*, 2009.

[36] Adi Botea, Markus Enzenberger, Martin Muller, and Jonathan Schae, "Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators," *Journal of Artificial Intelligence Research 24*, pp. 581-621, 2005.

[37] Romeo Sanchez Nigenda, XuanLong Nguyen, and Subbarao Kambhampati, "AltAlt: Combining the Advantages of Graphplan and Heuristic State Search," Technical Report 2000.

[38] R. M. Simpson, "GIPO Graphical Interface for Planning with Objects," in *Proceedings of the International Conference for Knowledge Engineering in Planning and Scheduling, Monterey Workshop*, 2005.

[39] Owens, Richard C., Jr., and Timothy Warner. (2003) Concepts of Logistics System Design. [Online]. http://www.phishare.org/files/1888_Concepts_of_Logistics_System_Design_final_11_18_03.pdf

[40] Roman Barták, Daniel Toropila , "Reformulating Constraint Models for Classical Planning," in *Proceedings of the Twenty-First International FLAIRS Conference* , 2008, pp. 525-530.

[41] A. Barrett and D. Weld, "Partial-order Planning: Evaluating Possible Efficiency Gains," *Artificial Intelligence*, pp. 71-112, 1994.

[42] G. Collins and L. Pryor, "Representation and Performance in a Partial Order Planner," Technical report 35, 1992.

[43] Alexander Nareyek, *Constraint-Based Agents - An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds*.: LNAI 2062. Springer, 2001.

[44] M. A. Hakim Newton, John Levine, Maria Fox, and Derek Long, "Learning Macro-Actions Genetically from Plans,".