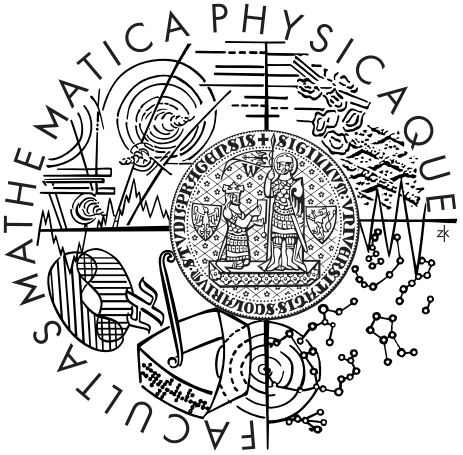
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Otakar Trunda

Monte Carlo Techniques in Planning

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: prof. RNDr. Roman Barták, Ph.D.
Study programme: Informatics
Specialization: Theoretical Computer Science

Prague 2013

I would like to thank profesor Roman Barták for supervising this thesis and for his inspiring suggestions and advice.
Also I'd like to thank my girlfriend Kamila for her support and patience with me during writing this thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date ..... signature of the author

Název práce: Monte Carlo Techniques in Planning

Autor: Otakar Trunda

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: prof. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Algoritmus Monte Carlo Tree Search (MCTS) v nedávné době prokázal, že dokáže úspěšně řešit těžké problémy v oblasti optimalizace i v oblasti hraní her. Pomocí tohoto algoritmu byly vyřešeny i některé problémy, které dlouho vzdorovaly konvenčním technikám. V této práci zkoumáme možnosti aplikace MCTS v oblasti plánování a rozvrhování. Problém zkoumáme z teoretického pohledu a snažíme se identifikovat případné potíže při použití MCTS v této oblasti. Navrhujeme řešení těchto problémů pomocí úpravy algoritmu a pomocí předzpracování plánovací domény. Představujeme techniky které jsme pro tyto účely vyvinuli a integrujeme je do funkčního celku. Výsledný algoritmus specializujeme na konkrétní typ plánovacích problémů - plánování přepravy. Vzniklý plánovač experimentálně porovnáváme se současnými plánovacími systémy.

Klíčová slova: plánování, Monte Carlo Tree Search, učení HTN, logistické domény

Title: Monte Carlo Techniques in Planning

Author: Otakar Trunda

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The Monte Carlo Tree Search (MCTS) algorithm has recently proved to be able to solve difficult problems in the field of optimization as well as game-playing. It has been able to address several problems that no conventional techniques have been able to solve efficiently. In this thesis we investigate possible ways to use MCTS in the field of planning and scheduling. We analyze the problem theoretically trying to identify possible difficulties when using MCTS in this field. We propose the solutions to these problems based on a modification of the algorithm and preprocessing the planning domain. We present the techniques we have developed for these tasks and we combine them into an applicable algorithm. We specialize the method for a specific kind of planning problems - the transportation problems. We compare our planner with other planning system.

Keywords: planning, Monte Carlo Tree Search, HTN-learning, transportation domain

Contents

Introduction

In this thesis we explore the ways how to use a stochastic optimization algorithm Monte Carlo Tree Search in the field of automated planning and scheduling. Most of the techniques currently used for planning scheduling are based no logic and reasoning combined with a deterministic backtracking search algorithm enhanced with many heuristics.

In this thesis we try to replace the deterministic backtracking search by a stochastic non-backtracking optimization algorithm based on a random sampling which is known as Monte Carlo Tree Search (MCTS)

We describe the algorithm and specify the planning task we want to solve. Then we analyze the problem of using MCTS for that task. Based on this analyzis we develop a planning algorithm which we implement and test on several classical planning and scheduling problems.

Motivation

Our interest in MCTS was motivated by its recent successes in solving many kind of difficult optimization problems [1] and in game playing [2].

We have recently tryied to use MCTS technique for planning to solve the Petrobras domain (see appendix A.4) proposed at *The International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)* conducted under *The 22nd International Conference on Automated Planning and Scheduling (ICAPS)* in 2012. We created an ad-hoc planning algorithm based on MCTS designed specifically for the Petrobras domain. The algorithm is described in related published paper [3].

In conducted experiments the ad-hoc planner outperformed the standard planning software [3],[4] and contributed to winning the *Outstanding Performance on the Challenge Track* award on ICKEPS 2012 (together with the other authors).

Based on these promissing results we have decided to investigate the possibilities of MCTS in planning further.

Goal

In this thesis we would like to extend our work done on Petrobras and generalize the techniques used there to create a planner capable of solving a larger variety of planning and scheduling problems.

The ad-hoc planner described above is strongly tied with Petrobras domain. It uses many domain dependent search and pruning techniques as well as domain dependent knowledge representation. (These techniques form a substantial part of the planning process - MCTS is used in rather straightforward way only as a final step over an already preprocessed problem.) Therefore it was only able to solve problems from single domain.

Our goal is to develope a domain indepenedent planner such that

1. it would use a standard knowledge representation (based on logic - predicates as state-variables and instantiated operators as state-transitions)
2. it would be usable in larger variety of domains (described in PDDL)
3. it would use MCTS as a main search algorithm (possibly combined with other techniques and enhancements)
4. it would be competitive with standard domain independent planners (at least at some domains)

We would also like to contribute to a theoretical research about MCTS and its usability in the field of planning and scheduling. We will try to find out which planning domains are best suited for MCTS and which are not (and why), and where are the limits of MCTS approach in planning.

Structure

In the first chapter we describe the MCTS algorithm in its basic form as well as some enhancements and modifications that we reffer to later in the thesis. We also mention a few theoretical properties of MCTS which are important with respect to our modifications to the algorithm described later.

This chapter is rather theoretical and contains mostly already known facts about the algorithm which we present with connection to the task of planning. It stands as an introduction to the topic.

The second chapter contains an analyzis of the problem we're solving. Here we define the planning problem and analyze how MCTS could be used in the field of planning and scheduling. We describe some other domains where MCTS was successfully used and compare them to planning. Main goal of this chapter is to identify possible obstacles that we have to overcome in order to use MCTS efficiently for planning.

In this chapter we try to sort the known facts about both MCTS and planning and possibly discover new interesting connections between these fields.

The third chapter is the most important part of this thesis. We present here our contribution to the topic. In this chapter we present our solutions to the problems described in section two, if there were more alternatives we present all of them and defend the way we choosed. We describe here the planner we developed.

In the following chapter we present the results of experiments we performed to compare our planner with other standard domain-independent planning software. In the experiments we also try to determine the optimal values of the parameters of the algorithm.

In the last section we summarize the results and mention other related work and possible future work on this topic.

A the end we provide the list of bibliography we reffered to in the thesis, the list of abbreviations we used, and the appendices. The first appendix contains description of the planning domains we used in the thesis (as examples or in the experiments). Second apendix describes the content of the DVD attached to this thesis.

1. Monte Carlo Tree Search algorithm

Overview In this chapter we briefly introduce the Monte Carlo Tree Search algorithm (MCTS). We focus on describing the basic principles, specific details can be found in cited bibliography. We also present some theoretical features of MCTS that might be affected by our modifications to the algorithm as described in the chapter 3.

1.1 Introduction

Monte Carlo Tree Search (MCTS) is a stochastic optimization algorithm that combines classical tree search with random sampling of the searched space. The basic concept is very simple but many modifications and enhancements have been devised [5]. We will first describe the predecessor of the MCTS where we will show the basic concepts.

1.1.1 Motivation for MCTS

Monte Carlo methods have been successfully used to solve various problems, especially numerical [6]. Therefore it has been attempted to use it in different areas, mostly in computer Go, where classical approaches didn't perform well. The most basic idea of using Monte Carlo technique in game playing is as follow:

Consider we are given a game position and are asked to find the best move for an active player. We will evaluate every possible move and then select the most promising one. The evaluation will be done (based on the Monte Carlo technique) by random sampling of the space of all possible developments of the game from the given state to the end. This random sample is implemented as a simulation starting with the move we want to evaluate and then making random moves until the game ends. (This simulation is called a *playout*.) When reaching the end the playout is evaluated according to the result of the game.

The basic algorithm called *Flat Monte Carlo* would run a fixed number of simulations for every possible move and then it would select the move with the best average outcome. There are two major drawbacks of this approach:

1. The algorithm doesn't converge i.e. there are positions where it will never find the optimal move even if the number of simulations goes to infinity. (The formal definition of convergence will be given later in this chapter.)

2. The algorithm allocates the same number of playouts (and therefore the same computation time) to every move. This leads to ineffectiveness since the most promising moves should be searched much more throughly than least promising ones.

MCTS is a modification of Flat Monte Carlo that doesn't suffer from these defects. It enhances the search process with a tree that is consequently built in an asymmetric manner based on results of the playouts. The most promising parts of the searched space are modeled more precisely (by larger tree) while in the less promising parts the tree is shallow. Moreover with a proper *tree policy* (will be explained) the algorithm can be shown to converge.

1.1.2 Basic description of MCTS

The basic concept of the MCTS can be described as follow. The algorithm repeatedly performs these steps:

1. Selection
    - The tree built so far is traversed from root to a leaf using some criterion (called *tree policy*) to select the most urgent leaf.
2. Expansion
    - All applicable actions for the selected node are generated and the resulting states are added to the tree as successors of the selected node. (Sometimes different strategies are used - like adding only one successor every time.)
3. Simulation
    - A pseudo-random simulation is run from the selected node until some final state is reached (the state of the searched space that doesn't have any successors). During the simulation the moves are selected by a *simulation policy* (also called a *default policy*).
4. Update
    - The result of the simulation is propagated back in the tree from the selected node to the root and statistics of the nodes on this path are updated according to the result.

The schema can be seen in the picture 1.1 taken from [7].

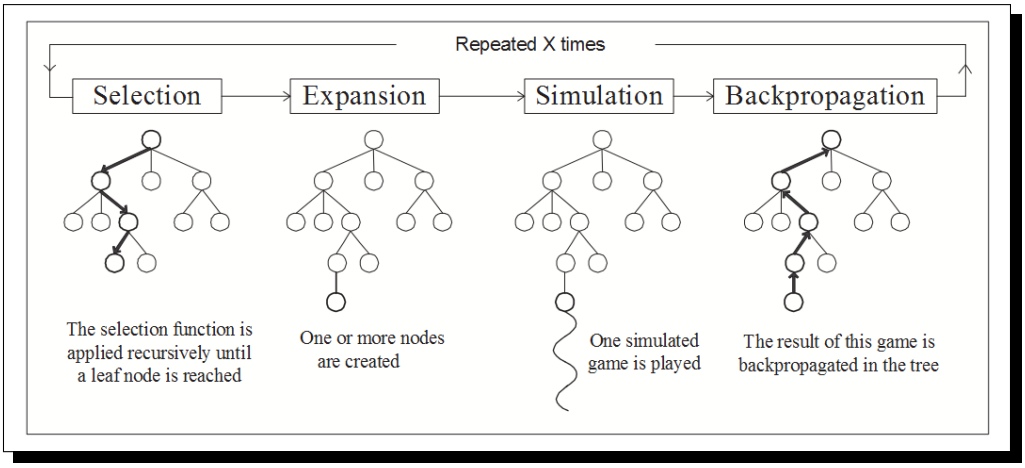


Figure 1.1: Basic schema of MCTS

1.2 Detailed description

We describe the individual phases in greater detail since they form the core of the MCTS algorithm and hence the core of our planner.

1.2.1 Selection criterion

The selection criterion (or a tree policy) is one of the most important parts of the algorithm. It determines which node will be expanded and therefore it affects the shape of the resulting tree. The purpose of tree policy is to solve the exploration vs. exploitation dilemma. We need to search the promising parts of the search space to achieve good results (exploitation); as well as search the unknown parts to determine whether they are promising or not (exploration).

Commonly used policies are based on a so called *bandit problem* and *Upper Confidence Bounds* [8],[9] which provide a theoretical background to measure quality of policies. We will not discuss it here further. We only present the most popular tree policy and its important properties. Before doing that we shall define the necessary terminology. Let n be the currently selected node and let $n_1, ..., n_k$ be it's successors in the tree. We need to select the most urgent successor (and then continue with the selection recursively).

- Let $Visits(m)$ be the number of simulations that passed through the node m and
- $Reward(m)$ the average reward of the simulations that passed through the node m

1.2.2 Upper Confidence Bounds for Trees (UCT)

Using the previous notation the UCT value of node n_j is

$$UCT(n_j) = Reward(n_j) + c\sqrt{\frac{2\ln(Visits(n))}{Visits(n_j)}}$$

where $c > 0$ is a constant. The node is then selected to maximize the UCT value.

The formula has two parts. The first component ($Reward(n_j)$) is called the *Expectation* and the second ($\sqrt{\frac{2\ln(Visits(n))}{Visits(n_j)}}$) is called the *Bias*. Sum of these components is called the *Urgency* of the node. UCT tree policy selects the most urgent node (i.e. maximizes Urgency). The purpose of the two parts is as follow:

- The Expectation component increases if the simulations that pass through the node have high reward. This component supports the exploitation.
- The Bias component of node n slowly increases every time the sibling of n is selected (that is every time the node enters the competition for being selected but is defeated by another) and rapidly decreases every time the node n is selected i.e. it prefers nodes that haven't been selected for a long time. This supports the exploration.
- Constant c - the parameter that determines the *exploration vs. exploitation* ratio. Its optimal value depends on the domain and on other modifications to the algorithm, in computer Go the usual value is about 0.2 (supposing that some AMAF heuristic is used - will be described later).

1.3 Theoretical features of MCTS

We will mention several theoretical properties of the algorithm that will be discussed later on with the connection to our modifications to the algorithm.

1.3.1 Convergence

Since MCTS is a probabilistic algorithm, we cannot guarantee that some fixed performance will always be achieved, instead we can make statements about the probability of achieving the result. Furthermore the performance depends on the number of simulations that the algorithm has performed (i.e. on the number of steps). We use the following notation:

- Let A be a probabilistic online optimization algorithm
- X be a set of its possible inputs

- $opt(x)$ be the optimal output for the input $x \in X$
- $A_n(x)$ be a random variable which represents the output of the algorithm A after n steps on the input $x \in X$

We say that an algorithm A converges to an optimal solution (or simply converges) if

$$\lim_{n\to\infty} Pr(A_n(x) = opt(x)) = 1$$

for all $x \in X$ (where Pr denotes probability).

To ensure that MCTS converges it suffices to guarantee that no tree node will be infinitely many times skipped during the selection (i.e. that every node will be selected and expanded at some point). This property implies that for every k there exists m such that after m steps of the algorithm all the nodes in depth $\leq k$ will be expanded and thus if the optimal solution lies in depth k it will be found in at most m steps. *UCT* has this property since the Bias component of continually skipped nodes goes to infinity (and we assume that the Expectation is always bounded).

1.3.2 Exploration vs. exploitation

UCT tree policy is proven to provide asymptotically optimal exploration vs. exploitation ratio [8]. However good asymptotical estimation doesn't imply good practical performance. (Practical performance is greatly affected by the choice of parameter c) Several other policies have been proposed - one of the most popular is *UCB-Tuned*, which performs well in some practical problems but asymptotical optimality has not been proved. In practice there are indeed situations when UCT doesn't perform well. We will describe one of these situations here to provide insights into behavior of the algorithm.

We have conducted a simple experiment to understand how *UCT* policy allocates computation time to the nodes (based on their Expectation). Consider the following example: we have 30 nodes and we want to analyze how many times every particular node will be selected by *UCT* (i.e. how many simulations will pass through the node which represents how much effort the algorithm will make to explore the node).

Initial situation is as follow: We have 30 nodes $n_1, .., n_{30}$ with a common predecessor (which we call the root), each is assigned a fixed Expectation value (which doesn't change during the experiment). The Expectation values are assigned uniformly from interval $(0, 1)$, in decreasing manner - the first node has its Expectation close to 1, while the last one has value close to 0. (More formally the node n_i has its Expectation set to approximately $1 - \frac{i-1}{29}$). We simulate 100 000 cycles of selection by *UCT* from the root and measure how many times every node is selected. Picture 1.2 shows the results.

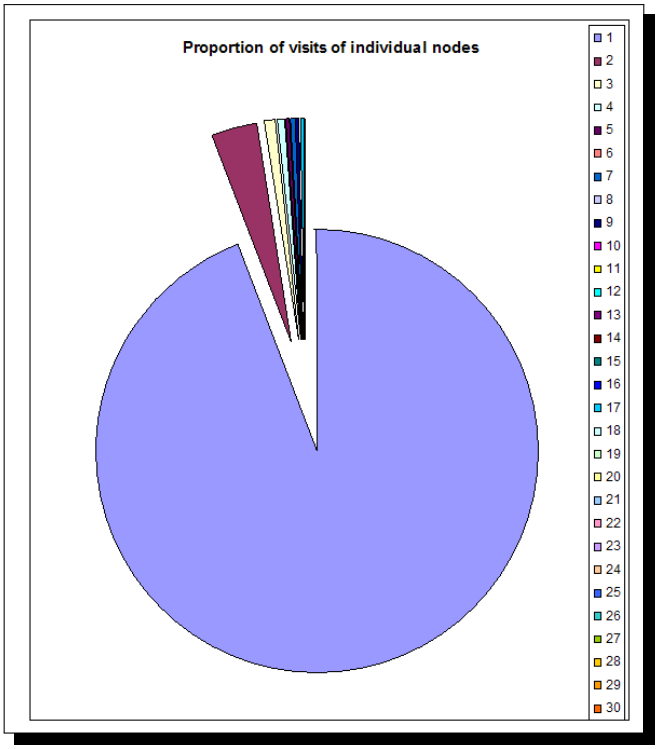


Figure 1.2: Number of visits to the nodes. (The whole intreval $(0, 1)$ was covered by the Expectation values)

Out of total number of 100 030 cycles, the first node was selected 94208 times, the second 3228 times, the third 980 times and so on, last node was visited 6 times. We can see that promising nodes are selected exponentially more often than the others, which causes UCT to provide asymptotically optimal exploration vs. exploitation ratio. In the following experiment we will analyze how the number of visits depends on the size of interval from which the Expectation values come.

We modify the previous experiment in the way that the Expectation values are now uniformly distributed in the interval $(0, 0.5)$, decreasing order is preserved (node n_1 has highest value). As in the previous experiment we visit each node once and then run 100 000 cycles of UCT selection. Results are shown in the picture 1.3. The number of visits of nodes n_1, n_2, n_3 are $82264, 8804, 3146$ respectively. The least promising node n_{30} was visited 23 times.

We try two more interval lengths 0.25 and 0.1. The results are depicted in the pictures 1.4 and 1.5. In the case of length 0.25 the nodes n_1, n_2, n_3, n_{30} were visited $57145, 16568, 7756, 86$ times respectively, in the case of length 0.1 these values were $21667, 14771, 10712, 428$ respectively. Picture 1.6 shows the ratio of visits for 100 intervals of lengths from 0.01 to 1.

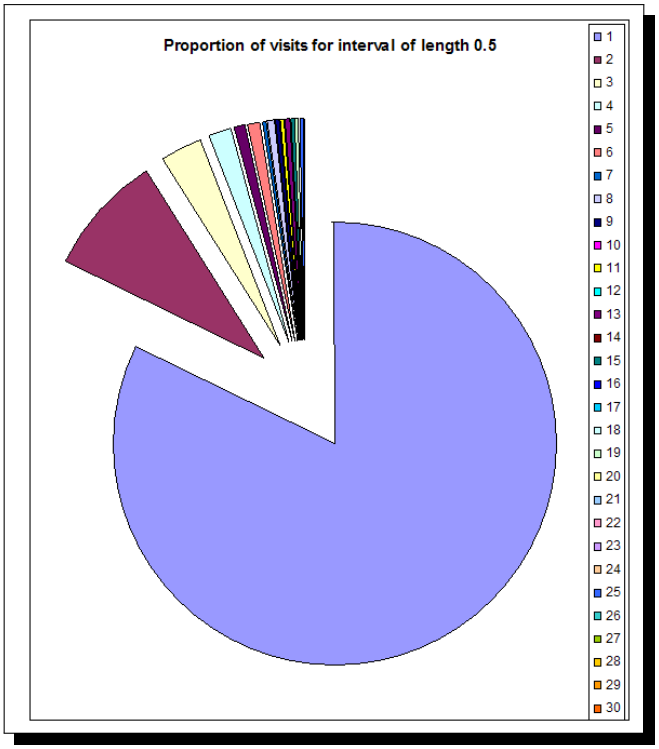


Figure 1.3: Number of visits to the nodes. (The Expectations are from the interval $(0, 0.5)$)

We can see that with the decreasing size of the interval the UCT policy is having hard time distinguishing promising nodes from those that are not (even though the ratio of Expectation values of any fixed pair of nodes is equal in all the experiments - i.e. $\frac{Expectation\ of\ n_1}{Expectation\ of\ n_{30}}$ in the first experiment is the same as in the last one). It allocates the computation time much more evenly than it should. Note that it still visits the promising nodes exponentially more often - that is if we increased the number of steps, the differences would increase as well - however the exponents of these exponential functions are so small that it would take a huge number of steps for the effect to take place. In practice we always have a fixed number of simulations that can be performed.

(A note to the methodology used to design the experiment.) We assume that the Expectations of the nodes doesn't change during the experiment and that the number of simulations that pass through the node is predetermined and fixed, yet none of these assumptions hold in practice. Further more we stated that increasing the number of simulations would increase the differences of the visits to the nodes (even for smaller interval). However the mothod we used is correct since we want to observe the dependence of visits on the size of the interval (i.e. partial derivative with respect to the size of the interval) therefore the other variables has to be fixed.

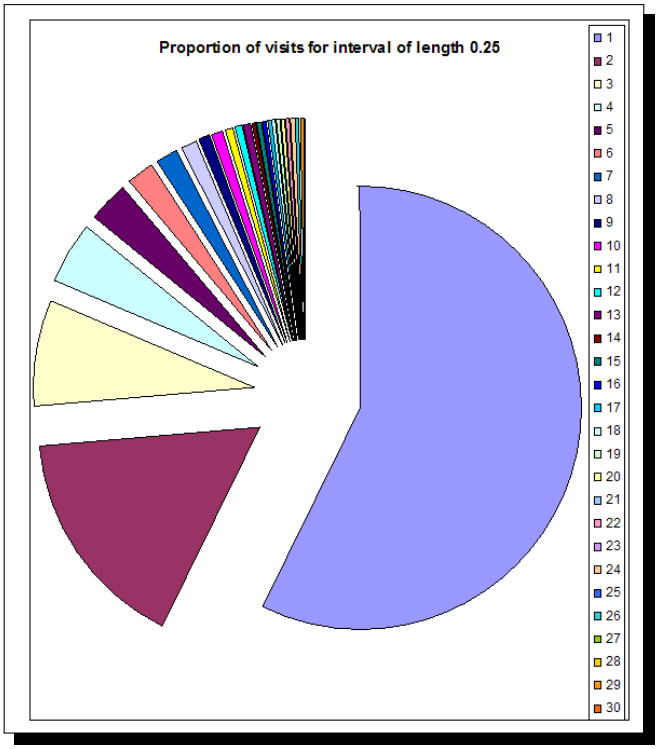


Figure 1.4: Number of visits to the nodes. (The Expectations are from the interval $(0, 0.25)$)

This problem was encountered in computer Go as well - it emerges in so called handicap games (when one player is given advantage at the beginning of the game) [11]. Handicap causes the simulations to be biased in favor of the advantaged player that is the Expectation of the nodes doesn't cover the whole interval $(0, 1)$ but rather only a part of it. Size of the interval depends on the magnitude of the handicap (the bigger handicap the smaller interval). In these situations MCTS-based Go engines were observed to perform poorly [11].

The problem can be analyzed in terms of *signal to noise ratio* - when the absolute difference of the Expectation values of two nodes is small UCT will select both nodes almost evenly. Another way to look at this is that the two components of UCT formula (Expectation and Bias) has to be of the same magnitude otherwise it makes no sense to sum it. In this point of view the problem can be solved by modifying the c constant of the formula so that the two components were of the same magnitude again.

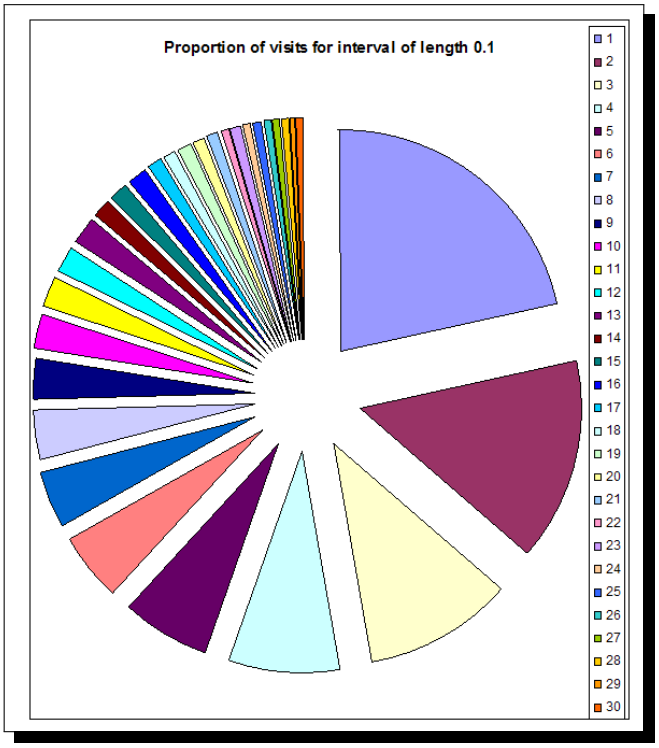


Figure 1.5: Number of visits to the nodes. (The Expectations are from the interval $(0, 0.1)$)

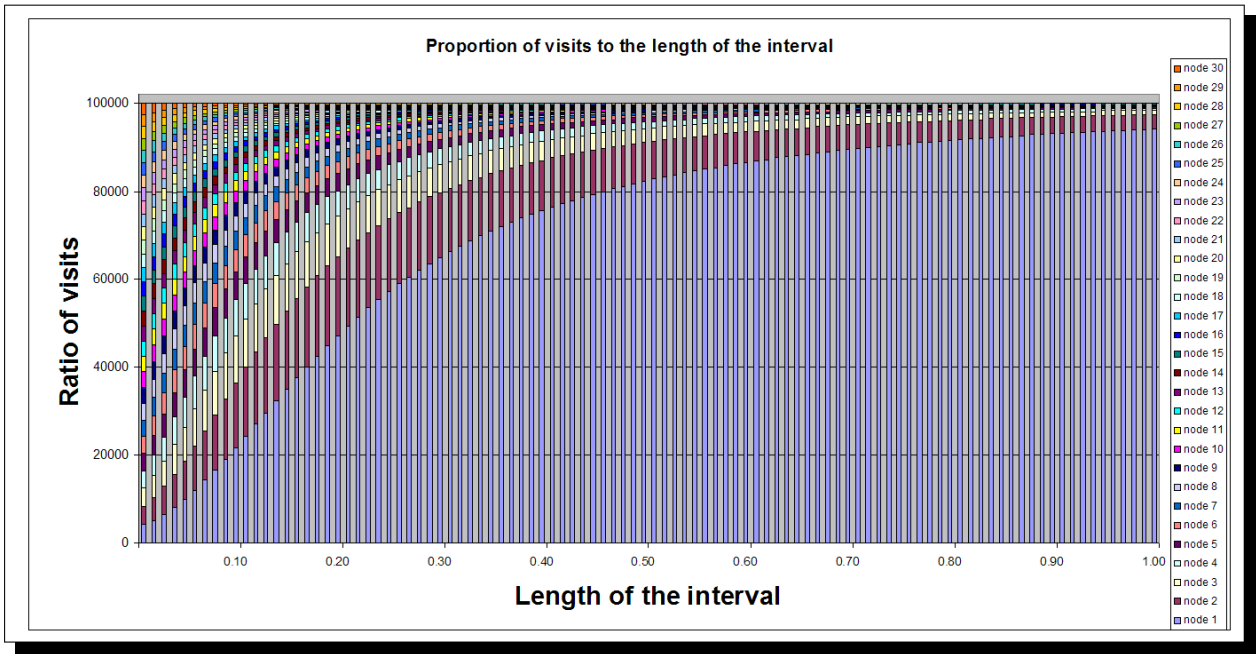


Figure 1.6: Number of visits to the nodes for different sizes of the interval.

1.4 Single-Player MCTS

Single-Player MCTS (SP-MCTS) is a variation of MCTS designed specificaly for single-player games (optimization problems, puzzles) was introduced by Schadd et. al. [12].

The main difference between SP-MCTS and the standard algorithm is that another component is added to the UCB formula which represents the standard deviation of Expectation value. The standard deviation component is *added* to the formula hence nodes with *higher* variance are preffered in the selection phase. (During the selection the nodes that maximize the formula are chosen.)

Reason for this is that simple average cannot distinguish the situation when all simulations that passed through the node were of average quality from the situation when there were some very high rewarded playouts and some low rewarded ones. The latter case is preferable since we are interested in nodes that lead to *some* good solution not in nodes where *all* solutions are good (or average in this case). The deviation term allows us to distinguish these situations since in the former case the variation is low while in the latter it is high.

Authors of SP-MCTS also suggested that a simulation heuristic should be used as a default policy in this case. As was previously showed by Gelly and Silver [10], in standard MCTS a strong simulation heuristic doesn't necessarily lead to better performance. (It provides more accurate estimations than random simulations, but the tree policy is only interested in the *difference* between the values rather than the values themselves. If simple random simulation can estimate the differences well enough - inaccuracies are smoothed by mutual cancelation of errors - then stronger heuristic will only slow the process.)

In Single-player games, however, we are very interested in obtaining the precise value of the simulation for we can directly use the value as a solution to the problem. In classical two player MCTS this is not possible for a high simulation outcome is usually caused by a poor move-selection for one of the players therefore we can't guarantee that this result can in fact be achieved in the game. (The second player will probably choose stronger moves).

In Single-player games this is not the case and the result of any simulation can be considered a solution of the problem. Hence a strong simulation heuristic will provide better solutions.

This fact is also exploited by another modification used in single-player version. Bjornsson and Finnsson [13] suggested that the best simulation result should in addition be stored for each node and used in the selection phase to prefer nodes with higher *Personal Best*. This strategy is motivated by the same principle as the standard deviation term described above.

Both these modifications work well in Single-player games, in multiplayer games however it is not used since the *Personal Best* value might be misleading (as described above).

1.5 Comparison to other algorithms

1.5.1 A^*-algorithm

A^* and its variations are commonly used for solving puzzle problems (Single-player games) of optimization type (like shortest paths, 15-puzzle, Rubbics cube and so on). Its advantages against MCTS are that it can naturally handle domains that allows infinite paths in state-space (see section 2.2.1 for details) and that it guarantees finding an optimal solution and when used with an admissible heuristic, it can provide a theoretical upper bound on the maximum steps needed to find a solution.

Disadvantages would be the need of a heuristic function and a narrow field of usage (MCTS can be used for both Single-player and Multi-player games and for both optimization and satisfacing problems).

1.5.2 $\alpha - \beta$ pruning

$\alpha - \beta$ pruning is a classical algorithm for two player games. Unlike MCTS it is well suited to handle state-spaces with trap-states (see section 2.2.3 for details) where there are many forced moves and therefore it works better in tactical situations.

Drawbacks are again the need of a heuristic evaluation function, a narrower field of usage than MCTS and poor performance in domains with the large branching factor.

1.5.3 Genetic algorithms

Genetic algorithms are a robust optimization technique used successfully for many types of problems. They are more similar to MCTS than previously mentioned algorithms since both these are meta-heuristics with many modifications and a wide field of usage. Moreover both are stochastic any-time algorithms and don't guarantee finding an optimal solution.

The difference between MCTS and genetic algorithms is in the means used to gain information about the searched space. MCTS explores the space by pseudo-random sampling (*Default policy*) which is targeted to promising parts of the space by *Tree policy*. Genetic algorithms on the other hand rely on a population-based method - exploration is performed by combining individuals from the current population and is guided by selection based on the fitness of individuals.

Both these techniques have their advantages - MCTS enables to involve a hierarchical structure to the exploration process and therefore the exploration can be performed in a sequential manner. The genetic algorithms on the other hand make use of *near-optimal substructures* during the search which can prove very useful.

According to the *Building block hypothesis* [15], the genetic algorithm implicitly searches for so called *building blocks* that represent near-optimal solutions for some subproblem of a given problem (*near-optimal substructure*). Then it uses these blocks to create a solution for the whole problem (by combining the building blocks).

MCTS doesn't implicitly use the building blocks even though it would be useful. For example in planning the building blocks would represent near-optimal solutions (plans) for some subgoal of the problem, or in game playing it might represent an optimal response for a move (i. e. a sequence of length two) which would be used to find a *main line* (sequence of optimal moves for both sides). The optimal response for the move would not have to be searched every time it is needed but rather stored as a building block and then used when necessary. (This technique is closely related to *History heuristic* [16].)

The genetic algorithms however have several other drawbacks which make it difficult to use them directly for planning. The main problem is how to design the operators of cross-over and mutation so that feasible solutions (i.e. solving plans) would be transformed to feasible solutions again. It might be possible to use GA for planning indirectly - for example the combination of GA with local search by Kubalik et al. [17] created a powerfull optimization technique.

2. Use of MCTS in planning

Overview In this chapter we study the structure of a planning problem in order to identify possible difficulties that can arise when trying to use the MCTS algorithm for planning. We compare the planning task to several other problems successfully addressed by MCTS and try to find a pattern among them that would help us understand which problems are well suited for MCTS and which are not. It could also tell us how to modify the planning problem to make it more suited for MCTS.

To decide whether MCTS is suited for solution planning problems we first define the problem. We use definitions as in Ghalab, Nau, Traverso [18].

- a state is defined by a set of state-variables (predicates)
- transitions are defined by actions
- action is defined by its name, its preconditions, and its effects. Preconditions and effects can be both positive and negative
- action is applicable in a state if its preconditions hold in that state
- result of the application of an action a to a state s is a state determined by combination of the state s with action's effects (adding the positive effects and removing the negative effects). The result is denoted $apply(a, s)$
- plan is a sequence of actions
- plan $(a_1, a_2, .., a_n)$ is applicable at a state s if a_1 is applicable at s and a plan $(a_2, .., a_n)$ is empty or is applicable at the state $apply(a_1, s)$
- result of application of a plan $(a_1, a_2, .., a_n)$ to a state s is

$$apply(a_n,\ apply(a_{n-1},\ ...\ apply(a_2,\ apply(a_1, s))))$$

- planning problem is defined by the set of actions, the initial state and the goal states.
- a plan that is applicable at the initial state and result of its application at the initial state is some goal state is called a *solution plan* for the problem.

There are two basic types of planning problems - a satisficing planning and an optimal planning. A goal of satisfacing planning is to find any solution plan while the optimal planning is concerned with finding the best evaluated plan among all solution plans. (In the latter case the evaluation function for plans is given.) A satisfacing planning can be compared for example to Sudoku and the optimal planning to SameGame. As was discussed in chapter 1 the MCTS algorithm is more suited for optimization problems which it can

handle naturally. Special modifications can be used for satisfacing problems as well [19] [20] but the results are not so convincing. We will only deal with optimal planning in this thesis and by word *planning* we mean *optimal planning* from now on.

2.1 Classification of problems

The MCTS has been successfully used for solving several problems and games. If we found a connection between these problems it could help us with using MCTS for planning. We will use the problem classification developed by Hearn [21] which is based on computational complexity of the problem. The classification covers mostly games however other problems (such as planning) can be seen as one-player games and can be classified in this hierarchy as well.

2.1.1 Polynomial-length games

An important class of problems consists of games with polynomialy-bounded length, that is, the number of possible moves in any particular game is limited by a polynomial in the size of the problem. A typical member of this class is game Hex. In this game players alternately seize the tiles. Once seized the tile will never be freed again therefore a maximum number of moves in any game is limited by the size of the board. A similar limitation holds in the case of SameGame as well as many other games and problems.

Game Go is not a polynomialy-bounded game since the stones can be both added and removed from the board (in fact it is not so easy to classify its complexity at all). However most of current MCTS-based Go engines use strategy that during the simulation the stones are not removed, only new stones are successively added. This causes the simulations to be inaccurate but much easier to carry out. Using this strategy the game (at least in the simulation phase) is polynomially bounded.

Polynomial-length games (or problems) can in general be characterized as:

1. the moves (actions) are typically irreversible (once we apply an action we cannot go back to the previous state)
2. there exists a polynomial resource that is successively consumed by every move (action)

2.1.2 Unbounded length games

The class of Unbounded length games covers games that does not have a polynomial limitation on game-length. There can be either no limitation (for example Sliding Blocks Puzzle which is a generalization of the 15-puzzle by Sam Lloyd) or some limitation might exist but not a polynomial one (for example Chess - the rules prevent infinite games to

occur but the theoretical limitation on maximum number of moves is so huge that in practical view the game can as well be considered unbounded).

A typical feature of unbounded games (or problems) is that the moves (actions) are reversible.

2.1.3 Conclusion

Based on this classification we can try to distinguish the problems that are suitable for MCTS from those that are not. Go, Hex and SameGame has all been successfully approached by MCTS. Chess on the contrary are still resisting to any attempt of using MCTS efficiently [22] and to our best knowledge no one has ever tried to use MCTS for Sliding Blocks Puzzle.

The main reason why MCTS has dificulties with handling games with an unbounded length is that it uses simulations to evaluate the nodes. These simulations has to be guaranteed to end (otherwise the algorithm could't work at all) and should be as short as possible because the longer the simulation takes the less number of simulations can be carried out within the given time limit. And as MCTS is heavily dependent on the number of simulations, this would lead to poor performance.

There are several ways to modify the algorithm so that it could be used for unbounded problems. The possible modifications will be discussed later in this chapter.

2.2 Structure of a planning problem

We will now try to compare the problem of (optimal) planning to some of the other problems mentioned earlier. If we take into consideration that planning can be seen as a one-player game and also that the goal is to optimize some evaluation function, it seems to be closely related to SameGame. To some extend this correspondence holds and we will use MCTS algorithm for planning in a similar manner as it is used for SameGame [12]. However the state-spaces of SameGame and of the planning have quite different structures.

To describe the structure of a state-space of a general planning problem is very difficult since planning can cover many types of problems. The PDDL language which is usually used to describe the structure of the problem is expressive enough to cover a lot of different types of domains [23],[24]. (In this respect it can be compared to GGP - General Game Playing.) Therefore we must consider the worst case scenarios when investigating the possible structure of planning problems. We will focus on features that are relevant with respect to use of MCTS.

2.2.1 Infinite simulation length

Unlike the SameGame a planning problem could in general allow infinite paths in the state-space (even though the state-space is always finite in case of Classical Planning) and this is quite usual in practice since the planning actions are typically reversible. This could be a serious problem for the MCTS algorithm. We will discus several ways to deal with this.

1. Modifying the state-space so that it would not contain infinite paths.

   - Modifying the state-space in this manner is quite difficult, it actually requires to solve the underlying satisfacing problem which might be intractable for general planning problems. We will discuss this issue more thoroughly in the next chapter.

2. Using a simulation heuristic which would guarantee that the simulation will be finite.

   - Such a heuristic is always a contribution to the MCTS algorithm since it makes the simulations more precise (see also 1.4). However obtaining this heuristic typically requires a domain-dependent knowledge.

3. Setting an upper bound on the length of simulations.

   - This approach has two disadvantages: the upper bound has to be set high enough so that it would not cut off the proper simulations. However if most of the simulations ended on the cut-off limit, then every one of them would take a long time to carry out which would have a bad impact on the performance. Furthermore it is not clear how we should evaluate the simulations that ended by a cut-off. This issue is discussed further in section 3.5.2.

2.2.2 Dead-ends

We use the term *Dead-end* to denote a state which si final (i.e. there is no applicable action) but doesn't represent the goal state. (They represent the situations where a standard search algorithm would backtrack.) Note that dead-ends can't occur in any game domain since in games any state that doesn't have successors is considered a goal state and has a corresponding evaluation assigned to it (like Win, Loss, or Draw in case of Chess or Hex, or a numerical value in case of SameGame for example). In planning however the evaluation function is usually defined only for the goal states or in other words - only for solution plans.

The plan that cannot be extended doesn't necessarily have to be a solution plan. Consider the following example: in the Petrobras domain (A.4) with only one ship we periodically apply the action *Navigate* until the ship runs out of fuel and it will end up in a location which can't refuel it. Then in this state no action is available (so it is a final state) but the goal has no been achieved. It is not clear how we should evaluate the simulation that ends in this state. Possible ways to solve this issue:

1. Modifying the state-space so that it would not contain dead-ends.
    - Again making such modifications is quite difficult - it is equivalent to solving the satisfacing problem.
2. Using a simulation heuristic which would guarantee that the simulation never encounters a dead-end state.
    - As stated in the previous section: such heuristic would certainly increase the effectivity of MCTS but obtaining it typically requires a domain-dependent knowledge.
3. Ignoring the simulations that ended in dead-end state.
    - In this case we would not concern ourselves with dead-ends at all. If the simulation should end in a dead-end we just forget it and run another one (hoping that it would end in a regular state and gets properly evaluated). This approach might be effective if dead-ends are very sparse in the searched space otherwise we could be waiting very long until some successful simulation occurs.
4. Finding a way to evaluate the dead-end states.
    - We will study this approach thoroughly in a separate section (3.5.2).

2.2.3 Trap states

Several attempts have been made to discover why MCTS shows poor performance in some domains and to identify the crucial features of the search space that are responsible for that. Ramanujan et al. [22] have been studying the structure of search spaces of both Go and Chess to find the differences that would explain why MCTS performs well in one domain and poorly in the other. They argue that poor performance of MCTS in Chess is caused by so called *trap states*.

In a two-player game a trap state for the first player is a state such that the second player can win the game within a small number of moves from that state. Therefore playing towards a trap state would be a mistake by the first player. Ramanujan et al. showed that MCTS is much less suited for detecting the trap states than standard $\alpha - \beta$ algorithm and that in domains with many trap states the performance of MCTS deteriorates

Fortunately trap states only occur in adversarial search spaces (when there is more than one player involved). In one-player games (such as planning) it has no equivalent so we shouldn't have this kind of problems. If this is in fact the only reason for MCTS to perform poorly, we should be able design efficient MCTS-based planning algorithm. (Note that the problem of dead-ends described above is fundamentally different than the one of trap states.)

2.3 Evaluation strategy

Another issue that has to be resolved in order to use MCTS for planning is the evaluation strategy of simulations. In case of Go or Hex the evaluation is quite simple - it is either 0 or 1 according to the result of the playout. The evaluation strategy is closely related to the tree policy which uses results of the simulations to select promising nodes.

Main difference between planning and classical applications of MCTS is that while in the game applications the Expectation value is naturally in the interval $(0, 1)$, in case of planning this is not true and the result of a simulation can be arbitrary positive real number. This will cause the Expectation component in the UCB formula to be of much higher magnitude than the Bias component and therefore the selection will not work properly. (The expectation will totally suppress the Bias and the algorithm will only perform the exploitation part of the search.)

The problem of incomparability of the two components was mentioned in section 1.3.2 as well as the solution to this problem. In this case it is necessary to adjust the parameter c in the formula so that the components were of the similar magnitude again. The solution we use will is described in section 3.5.1.

2.4 Prunning techniques

There is another difference between the usage of the MCTS algorithm in two-player games and in planning:

In two-player games (like Go) we only search for one move each time. The position is set and the algorithm runs for a limited amount of time, after that the best move is chosen (usually the most visited one) then the search ends and the move is played. After receiving the opponents move the search starts again from a new position. The planning problem is different in these respects:

- we only run the algorithm once
- we search for the whole plan at once (not just for the first move)
- there is no time limit on the search for the first action

In other words - in game applications we use several short MCTS searches while in SP-MCTS we only run the algorithm once. Since this search takes much more time the tree gets much larger which might cause memory issues.

The process used in game applications can be seen as a pruning technique: we run the MCTS for limited amounth of time, then select the most promising successor of the root, remove all other successors of the root and continue the search in this branch only. Or in other words after specific amount of time we prune all the branches but one from the tree. In SP-MCTS however we keep all branches in the tree. Since this approach leads to rapid growth of the tree we need to introduce some pruning technique in order to avoid memory issues.

2.4.1 Hard pruning

Hard pruning is similar to the technique used in game applications - at some point it removes certain branch from the tree and the search continues in the remaining parts. In game applications this approach si necessary since the time for finding single move is usually limited so after some time we have to stop the search and return the best solution found so far. After doing that all the remaining branches are removed from the tree since there is no point in examining them any more. (They will not occur in the game.)

This represents an easy and straightforward way to reduce the size of the tree but it has several drawbacks and therefore it is not widely used in the case of SP-MCTS. In the Single-Player version there is no time limit for finding the first move but rather for finding the whole plan. Hence we don't have to yield the result sequentially - one action per time. A hard pruning would in this case be too restrictive.

Main drawbacks of this approach are:

1. loss of the convergence of the algorithm (see ssec:Convergence)
2. degradation of performance when wrong branch is pruned
3. designing the policy for selecting the branches to prune

2.4.2 Soft pruning

Soft pruning doesn't remove the branch from the tree permanently but rather makes it locked - i.e. doesn't allow the simulations to enter the branch. Later the branch may be unlocked again. This technique however doesn't reduce the tree size and therefore it is not very helpful for dealing with memory issues.

The benefit of this approach is that the simulations aren't spreaded over large space but rather focused on fewer number of branches. Therefore the tree can be expanded to greater depth and the solutions are more precise.

Together with soft pruning goes an *unpruning policy* which determines what branches should be unpruned and when this should be done. (This process is called *unpruning* or *widening.*) It can be shown that with proper *unpruning policy* the algorithm still converges. This allows us to combine the best properties of both approaches - the reduction of the tree and increased precision of the simulations (which is provided by pruning) while preserving the theoretical properties of the algorithm (which are guaranteed if no pruning is used). Furthermore it doesn't suffer from degradation of performance when wrong branch is pruned (for it can be unpruned again if necessary).

3. Design of the planner

Overview In this chapter summarize our theoretical findings about the use of MCTS in planning and we present the techniques we have developed for this task based on the analysis described in the previous section. It covers our modifications to the algorithm as well as domain preprocessing techniques. We describe our planner in detail and also discuss its advantages and drawbacks. We present our original work in these parts (with references to techniques developed by other outhors that we put in use which were described in previous chapters). At the end of this chapter we present general planning techniques that our works is based on (commonly used in the field) as well as a comparison to similar work by other authors. The more thorough list of similar work is given at he end of the thesis.

As was mentioned in the chapter chap:UseInPlanning there are several fundamental obstacles that prevent us from using MCTS directly for planning - potentially infinite simulation length (which is caused by cycles in the search-space) and the presence of Dead-ends and resulting problems with evaluation of the playouts. Possible ways to overcome these issues can be divided into two categories:

1. Modifying the algorithm so that it could handle these situations.
2. Modifying the domain so that these situations wouldn't occur.

In our planner we use both ways to achieve best possible performance. Detailed description of methods used follows. (Besides these fundamental problems there are several others minor issues with using MCTS in this field, one of them is the problem of scheduling described in the section 3.5.5)

Modifying the algorithm Our modifications to the algorithm cover the evaluation strategy, a heuristic function used as a default policy in the simulations and we also put in use several modifications suggested for the Single-player version of the algorithm. (see section 1.4 for details)

Modifying the domain We modify the planning domain by replacing the standard action model with so called meta-actions (will be explained). Meta-actions are learned during the search and their purpose is to rid the domain of cycles and dead-ends. The proposed model of meta-actions is theoretically applicable to all types of domains however in our planner we only use it for a specific type of planning domains where the technique is reasonably fast. The general approach to modify *any* domain will be described at the end of this chapter.

3.1 Restricting possible inputs

3.1.1 Reasons for restriction

As was mentioned earlier (2.2.1) modifying the domain so that its state-space wouldn't contain infinite paths nor dead-end states requires to actually *solve* the satisfacing problem. In this thesis we are concerned with optimal planning only (MCTS has been used to solve satisfacing problems like sudoku as well however it is much better suited for optimization problems). The problem of satisfacing planning is being extensively studied by many researchers and it seems to be far too complex to hope that MCTS can solve *any* satisfacing problem merely as a side-effect of optimal planning.

One of our goals is that the planner we design would be applicable for practical problems. Since solving arbitrary satisfacing problem is computationally too demanding and in general intractable, we have decided to restrict the class of domains which our planner would solve.

3.1.2 Selected class of problems

Based on good results with Petrobras [3],[4], we choose to work with the transportation domains. This kind of domains seems to be well suited for our method since:

1. it is naturally of an optimization type (typicaly the fuel and time consumption are to be minimized)
2. the underlying satisfacing problem is usually not difficult
3. transportation problems often occur in practice

3.1.3 Transportation component

We have restricted the class of input problems to problems of a transportation type therefore we would like to exploit the transportation structure of the domain by our planner. To do that we need to identify the typical parts of a transportation domain (i.e. what makes the domain to be of transportation type).

In other words we want to assign a *meaning* to the predicates, the operators, and the types in the domain. In the PDDL input the structure of the domain is described in a syntactic way - i.e. it tell us *how can we manipulate* with predicates and constants using operators, but it doesn't tell us what these symbols of predicates, constants, and operators actually *mean*. In this section we describe a way how to assign a meaning to these symbols so that we could later exploit this higher level knowledge during the planning.

We introduce the term *Transportation component* which would denote the part of the domain that has a typical *transportation structure*. We have created a template that describes such structure (i.e. it describes the *relations* between the *symbols* that are typical for a transportation domain). By *symbols* we mean names of the predicates, types of the constants, and names of the operators. By *relations* we mean the associations of operators, predicates, and types.

- When some predicate occurs among the preconditions or the effects of some operator then the name of the predicate is somehow associated with the name of the operator.
- The *kind* of such association is determined by other circumstances. See following examples (we use operators from the Zeno-Travel domain A.1).
- Both operators works with the predicate with the name *in* but the relation between the symbols *load* and *in* is different that between *unload* and *in* because *unload* has *in* as a precondition while *load* has it as an effect.

```
(:action load
         :parameters    (?a - airplane ?p - person ?l - location)
         :precondition  (and (at ?a ?l)
                             (at ?p ?l))
         :effect        (and (not (at ?p ?l))

                             (in ?p ?a)))

(:action unload
         :parameters    (?a - airplane ?p - person ?l - location)
         :precondition  (and (at ?a ?l)
                             (in ?p ?a))
         :effect        (and (not (in ?p ?a))

                             (at ?p ?l)))
)
```

- Another example shows the operators for creating an oriented graph

```
(:action addEdge
         :parameters    (?from ?to - vertex)
         :precondition  ()
         :effect        (isEdge(?from ?to))

(:action addLoop
         :parameters    (?v ?meaninglessParameter - vertex)
         :precondition  ()
         :effect        (isEdge(?v ?v))
```

```
)
```

- In this case the relation between the symbols *addEdge* and *isEdge* is different than between *addLoop* and *isEdge* because the predicate is used differently by the operators.
- Last example showes two different operators possibly from two different domains

```
(:action load
        :parameters     (?a - airplane ?p - person ?l - location)
        :precondition   (and (at ?a ?l)
                             (at ?p ?l))
        :effect         (and (not (at ?p ?l))
                             (in ?p ?a)))

(...)

(:action get
        :parameters     (?v - vehicle ?c - cargo ?d - destination)
        :precondition   (and (isCargoAt ?c ?d)
                             (isVehicleAt ?v ?d))

        :effect         (and (not (isCargoAt ?c ?d))
                             (isInside ?c ?v)))
)
```

- Even though these operators use different predicates, the relation between the symbols *(get, vehicle, cargo, destination, isCargoAt, isVehicleAt, isInside)* is exactly the same as the relation between *(load, airplane, person, location, at, at, in).* (We used here a relation between types as well.)

The last example shows that the *structure* of the domain (which is what we want to capture) does not depend on the symbols used but only on the relations between the symbols. This gives us means to define a *generic transportation structure* and then we can check whether some given domain *matches* this predefined structure. (E.g. the action *get* in the last example can be seen as a *template of loading* and we can say that the action *load* matches this template.

A *transportation component* describes all the relations between symbols that has to hold if the domain is of a transportation type. These relations are expressed as a set of constraints over a set of variables. We say that some part of a planning domain *matches* the transportation template if we can substitute the *symbols* from given domain into the template such that all the constraints would hold. Or in other words if there exists an

endomorphism from the structure of the planning domain to the structure of the template. (Some examples will be given after the formal definition.)

Formaly: A transportation component is defined by:

- Three names of types that represent cargo, vehicles and locations. (Designated *Cargo*, *Vehicle*, *Location*)
- Three names of predicates which represent facts that
  1. cargo is at location (designated *CargoAtName*)
  2. vehicle is at location (designated *VehAtName*)
  3. cargo is in vehicle (designated *CargoInName*)

  All these predicates are of arity at least two. Two integers are associated with each predicate name. These integers represent positions of important variables among the predicate variables. These numbers are denoted as follows:

  1. *CA-CargoPos* - position of a variable representing a cargo in a predicate with the name *CargoAtName*
  2. *CA-LocPos* - position of a variable representing a location in a predicate with the name *CargoAtName*
  3. *VA-VehPos* - position of a variable representing a vehicle in a predicate with the name *VehAtName*
  4. *VA-LocPos* - position of a variable representing a location in a predicate with the name *VehAtName*
  5. *CI-CargoPos* - position of a variable representing a cargo in a predicate with the name *CargoInName*
  6. *CI-VehPos* - position of a variable representing a vehicle in a predicate with the name *CargoInName*
- Three names of operators that perform loading and unloading the cargo and moving the vehicle (designated *OpLoadName*, *OpUnloadName*, and *OpMoveName* respectively) All these operators have at least three variables and there are three integers associated with every operator that represents positions of important variables. These numbers are designated as follows.
  1. *OpL-CargoPos* - position of a variable that represents cargo among variables of operator with name *OpLoadName*
  2. *OpL-VehPos*, *OpL-LocPos* - positions of variables that represent vehicle and location respectively among variables of operator with name *OpLoadName*

3. *OpU-CargoPos*, *OpU-VehPos*, *OpU-LocPos* - positions of variables that represent cargo, vehicle and location respectively among variables of operator with name *OpUnloadName*
4. *OpM-VehPos*, *OpM-StartPos*, *OpM-TargetPos* - positions of variables that represent vehicle, start location and target location respectively among variables of operator with name *OpMoveName*

(Since the operator is uniquely identified by its name, we will sometimes refer to operators *OpLoad OpUnload OpMove* themselves instead of just their names.) Transportation component has to meet several conditions:

1. There has to be a predicate X_1 with a name *CargoAtName* among positive preconditions of operator *OpLoad* such that variable at position *OpL-CargoPos* in this operator is the same as variable at position *CA-CargoPos* in predicate X_1. (We say that position *OpL-CargoPos* in operator *OpLoad* is bound to position *CA-CargoPos* in X_1) And position *OpL-LocPos* in operator *OpLoad* is bound to position *CA-locPos* in X_1.
2. There has to be a predicate Y_1 with a name *VehAtName* among positive preconditions of operator *OpLoad* such that positions *VA-vehPos*, *VA-locPos* in Y_1 are bound to positions *OpL-VehPos* and *OpL-LocPos* respectively in the operator.
3. There has to be a predicate Z_1 with a name *CargoInName* among the positive effects of the operator *OpLoad* such that positions *CI-cargoPos* and *CI-vehPos* in the predicate Z_1 are bound to positions *OpL-CargoPos* and *OpL-VehPos* respectively in the operator.
4. (Similar conditions must hold for other two operators.) There exist predicates X_2, Y_2, Z_2 with names *CargoAtName*, *VehAtName*, *CargoInName* respectively such that X_2 is in positive effects of operator *OpUnload*, Y_2 and Z_2 are in positive preconditions of *OpUnload*. Position *OpU-CargoPos* in the operator is bound to position *CA-CargoPos* in the predicate X_2 and to position *CI-CargoPos* in the predicate Z_2, position *OpU-VehPos* is bound to position *VA-VehPos* in Y_2 and to position *CI-VehPos* in Z_2 and position *OpU-LocPos* is bound to position *CA-LocPos* in X_2 and to position *VA-LocPos* in Y_2.
5. There exist predicates X_3, Y_3 with name *VehAtName* such that X_3 is in positive precontitions and negative effects of operator *OpMove* and Y_3 is in positive effects of this operator. Position *OpM-VehPos* in the operator is bound to position *OpM-VehPos* in X_3 and in Y_3, position *OpM-StartPos* in the operator is bound to position *VA-LocPos* in X_3 and position *OpM-StartPos* in the operator is bound to *VA-LocPos* in Y_3.

The symbols *OpL-CargoPos*, *OpM-VehPos* and so on can be seen as a variables. When we search for the transportation components in the domain, we try to find the values for

these variables such that the constraints would hold. Every solution of this problem represents one transportation component. (Note that the single domain may contain several transportation components.)

Examples:

The Zeno-Travel domain (A.1) contains exactly one transportation component. The component is in this case defined by this substitution:

Cargo = "person"
Vehicle = "airplane"
Location = "location"
CargoAtName = "at"
VehAtName = "at"
CargoInName = "in"
CA-CargoPos = 1
CA-LocPos = 2
VA-VehPos = 1
VA-LocPos = 2
CI-CargoPos = 1
CI-VehPos = 2
OpLoadName = "load"
OpUnloadName = "unload"
OpMoveName = "fly"
OpL-CargoPos = 2
OpL-VehPos = 1
OpL-LocPos = 3
OpU-CargoPos = 2
OpU-VehPos = 1
OpU-LocPos = 3
OpM-VehPos = 1
OpM-StartPos = 2
OpM-TargetPos = 3
X_1 = "at(?p, ?l)"
Y_1 = "at(?a, ?l)"
Z_1 = "in(?p, ?a)"
X_2 = "at(?p, ?l)
Y_2 = "at(?a, ?l)"
Z_2 = "in(?p, ?a)"
X_3 = "at(?a, ?from)"
Y_3 = "at(?a, ?to)"

The original Zeno-Travel domain (A.2) contains two transportation components which differs only in the value of the variable *OpMoveName*. These components are defined as:

Cargo = "person"
Vehicle = "airplane"
Location = "location"
CargoAtName = "at"
VehAtName = "at"
CargoInName = "in"
CA-CargoPos = 1
CA-LocPos = 2
VA-VehPos = 1
VA-LocPos = 2
CI-CargoPos = 1
CI-VehPos = 2
OpLoadName = "load"
OpUnloadName = "unload"
OpMoveName = "fly"
OpL-CargoPos = 2
OpL-VehPos = 1
OpL-LocPos = 3
OpU-CargoPos = 2
OpU-VehPos = 1
OpU-LocPos = 3
OpM-VehPos = 1
OpM-StartPos = 2
OpM-TargetPos = 3
X_1 = "at(?p, ?l)"
Y_1 = "at(?a, ?l)"
Z_1 = "in(?p, ?a)"
X_2 = "at(?p, ?l)

Y_2 = "at(?a, ?l)"
Z_2 = "in(?p, ?a)"
X_3 = "at(?a, ?from)"
Y_3 = "at(?a, ?to)"

and

Cargo = "person"
Vehicle = "airplane"
Location = "location"
CargoAtName = "at"
VehAtName = "at"
VehAtName = "at"
CargoInName = "in"
CA-CargoPos = 1
CA-LocPos = 2
VA-VehPos = 1
VA-LocPos = 2
CI-CargoPos = 1
CI-VehPos = 2
OpLoadName = "load"
OpUnloadName = "unload"
OpMoveName = "flyFast"
OpL-CargoPos = 2
OpL-VehPos = 1
OpL-LocPos = 3
OpU-CargoPos = 2
OpU-VehPos = 1
OpU-LocPos = 3
OpM-VehPos = 1
OpM-StartPos = 2
OpM-TargetPos = 3
X_1 = "at(?p, ?l)"
Y_1 = "at(?a, ?l)"
Z_1 = "in(?p, ?a)"
X_2 = "at(?p, ?l)
Y_2 = "at(?a, ?l)"
Z_2 = "in(?p, ?a)"
X_3 = "at(?a, ?from)"
Y_3 = "at(?a, ?to)"

A few notes about the notion of transportation component:

- Part of the definition of the transportation components uses types therefore this analysis technique can only be used on domains that use typing.
- During the search for the transportation components we don't require the domain to be *isomorphic* to the template but rather only to *match* the template - i.e. we search for an *endomorphism* from the domain to the template.
- This allows us to handle the situations where the transportation component is embedded within a more complex structure in the domain. For example if in the Zeno-Travel domain the predicates had arity more than two (if they described some more complex relations), the operators had more parameters than three, and they had more preconditions and effects (i.e. if they performed some more complex work), we would still be able to identify the transportation component in the domain.
- The algorithm for searching for the transportation components in the domain is described in the section 3.4.1.

3.2 Meta-actions

Meta-actions (or composite actions) consist of sequences of original actions. The purpose of using Meta-actions instead of original actions is to prevent some paths in state-space from being visited. The original action model allows *every* possible path in search space to be explored by the planner but most of the paths are formed of meaningless combinations of actions which don't lead to the goal. We use the meta-actions to eliminate as much meaning-less paths as possible which would make the Monte-Carlo sampling process efficient enough to be worth using.

The idea of meta-acions is not new, it has been asdressed by the researchers for quite some time [29] [30]. (The meta-actions are called differently in some papers - *meta-actions*, *composite actions* or *compound actions*. We will use the term *meta-actions*. The original actions are sometimes reffered to as *primitive actions*, we will use the term *original actions*.) Although to our best knowledge the idea was never used with the connection to a HTN-learning nor to the MCTS algorithm.

3.2.1 Definitions

The meta-action can be seen as a sequence of original actions. If we want to apply the meta-action, we apply all the actions in the sequence successively. Formally:

- A meta-action A is defined by a finite nonempty sequence of original actions $(a_1, a_2, .., a_n)$
- The length of a meta-action is the length of its defining sequence
- A meta-action $A = (a_1, a_2, .., a_n)$ is applicable in a state s if
  1. its length is 1 and the primitive action a_1 is applicable in s. Or
  2. its length is greater then 1 and a_1 is applicable in s and the meta-action $B = (a_2, .., a_n)$ is applicable in $apply(a_1, s)$
- If a meta-action $A = (a_1, a_2, .., a_n)$ is applicable in s then the result of applying A to s is
  1. if length of A is 1 then $apply(A, s) = apply(a_1, s)$ else
  2. $apply(A, s) = apply(B,\ apply(a_1, s))$ where B is a meta-action defined by the sequence $(a_2, .., a_n)$

Note that these are only formal definitions. In our implementation we use different way to compute an application of a meta-action. Our algorithm has the same result as in the definitions but is faster than applying the original actions one-by-one. The algorithm is described in section 3.4.5.

3.2.2 Example of meta-actions model

We show an example of Zeno-Travel Domain (see A.1 for definition) with the original action model and with the meta-actions model to see the difference. We use meta-actions that were learned by the planner (the learning methods will be described later). We show a structure of the state-space of problem with three locations, two passengers and one airplane.

Prior to presenting the figures we shall describe the visualization techniques and conventions that we use when displaying the planning problems. We represent the state-space as an oriented graph with the vertices representing the states and edges representing transitions. Vertices are labeled by predicates that hold in the state, static predicates are omited (static predicates are true in all states). The edges are labeled by the instance of operator that was used as the transition. Sometimes we omit the edge labels to save space. The vertex representing the initial state is colored orange, vertices that represent final states are colored blue. The graphs were created by our own software, for the visualization we use a graph visualization library *GLEE* (currently named *Microsoft Automatic Graph Layout (MSAGL)* developed by Lev Nachmanson et al. under Microsoft Research.

Picture 3.1 shows the state-space of the problem using original action model (as defined in A.1), edge labels are omited. The graph has total 64 vertices and 192 edges. In the figures 3.2 and 3.3 there are fragments of this graph showing cycles and dead-end respectively.

We replace the original actions model by another that uses meta-actions. We use meta-actions that the program learned in the Zeno-Travel Domain. The actions are as follow.

```
(:action fly+load
         :parameters    (?a - airplane ?p - person ?from ?to - location)
         :precondition  (and (at ?a ?from)
                             (at ?p ?to))
         :effect        (and (not (at ?a ?from))
                             (not (at ?p ?to))
                             (at ?a ?to)
                             (in ?p ?a)))

(:action fly+unload
         :parameters    (?a - airplane ?p - person ?from ?to - location)
         :precondition  (and (at ?a ?from)
                             (in ?p ?a))
         :effect        (and (not (at ?a ?from))
                             (at ?a ?to)
                             (not (in ?p ?a))
```

```
                              (at ?p ?to)))

(:action load
         :parameters    (?a - airplane ?p - person ?l - location)
         :precondition  (and (at ?a ?l)
                             (at ?p ?l))
         :effect        (and (not (at ?p ?l))
                             (in ?p ?a)))

(:action unload
         :parameters    (?a - airplane ?p - person ?l - location)
         :precondition  (and (at ?a ?l)
                             (in ?p ?a))
         :effect        (and (not (in ?p ?a))
                             (at ?p ?l)))
)
```

The problem space with action model replaced by meta-actions is shown in figure 3.4, the edge labels are omited. Number of vertices is in this case 33 and there are 92 edges. Following figures 3.5 and 3.6 show detailed view on the space.

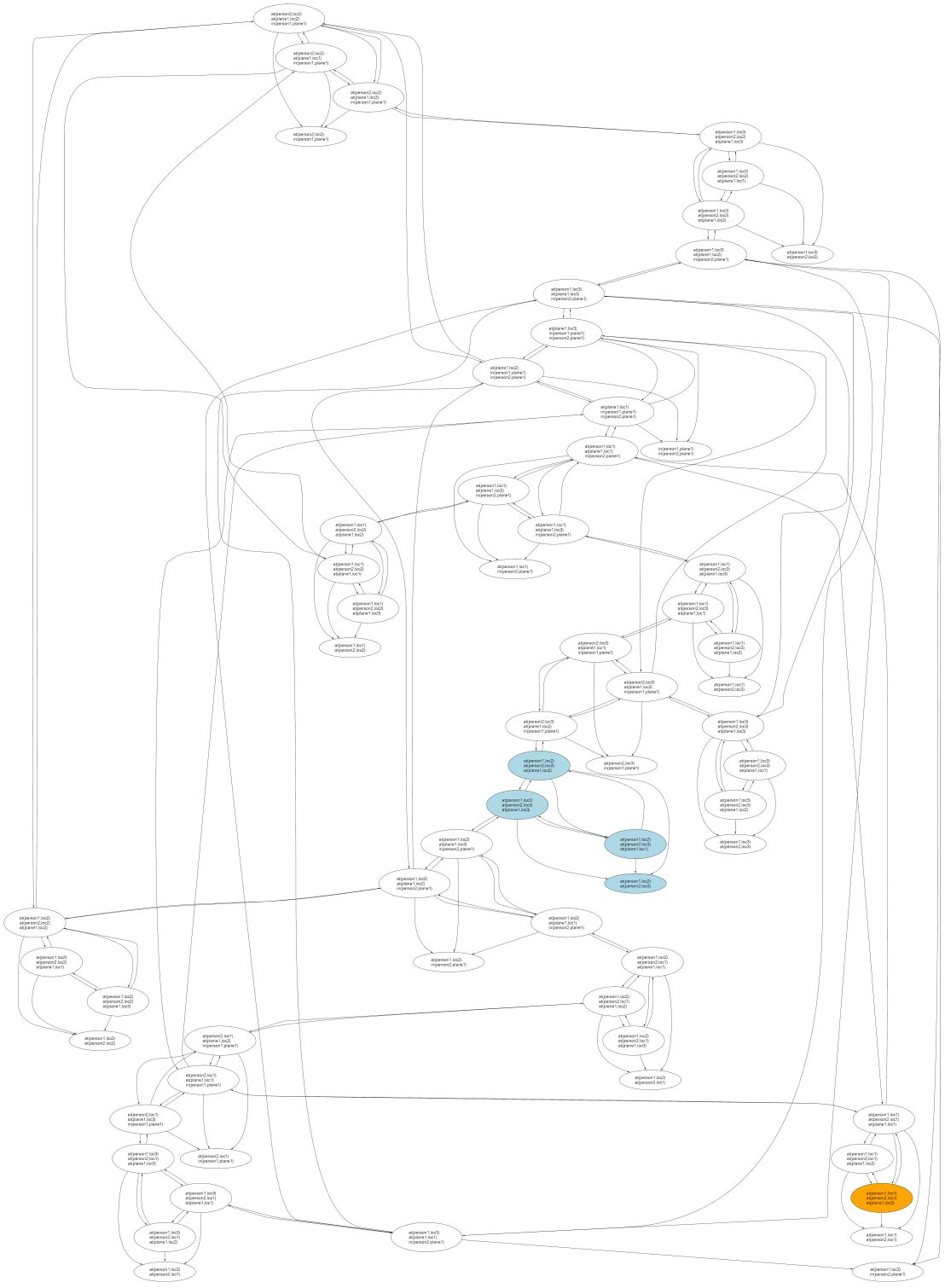


Figure 3.1: Original action model. 64 states, 192 edges

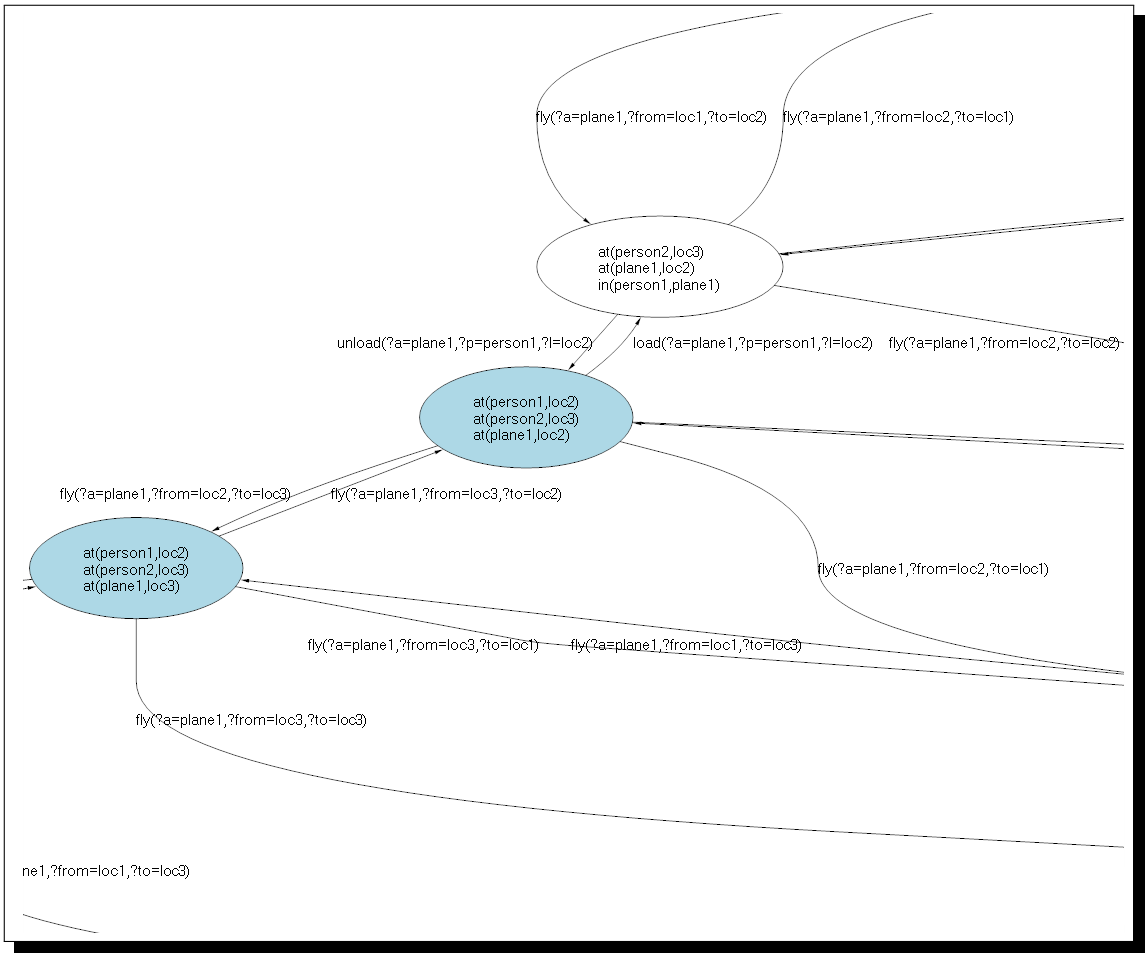


Figure 3.2: Original action model - cut-out showing cycles

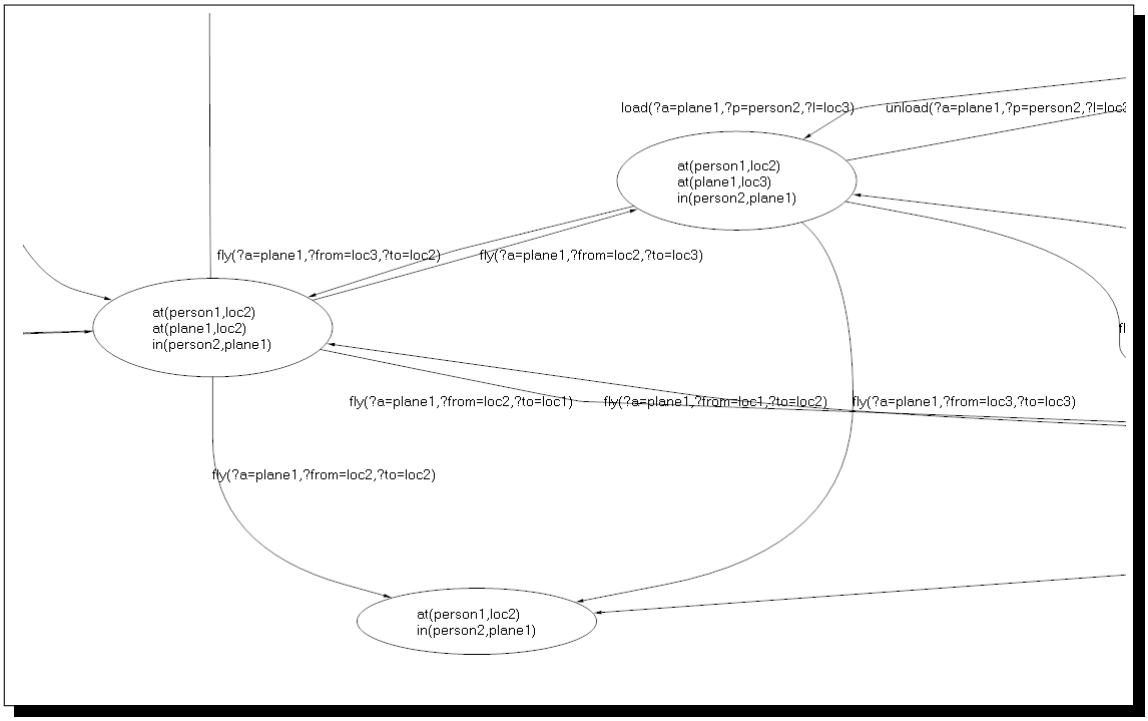


Figure 3.3: Original action model - cut-out showing a dead-end

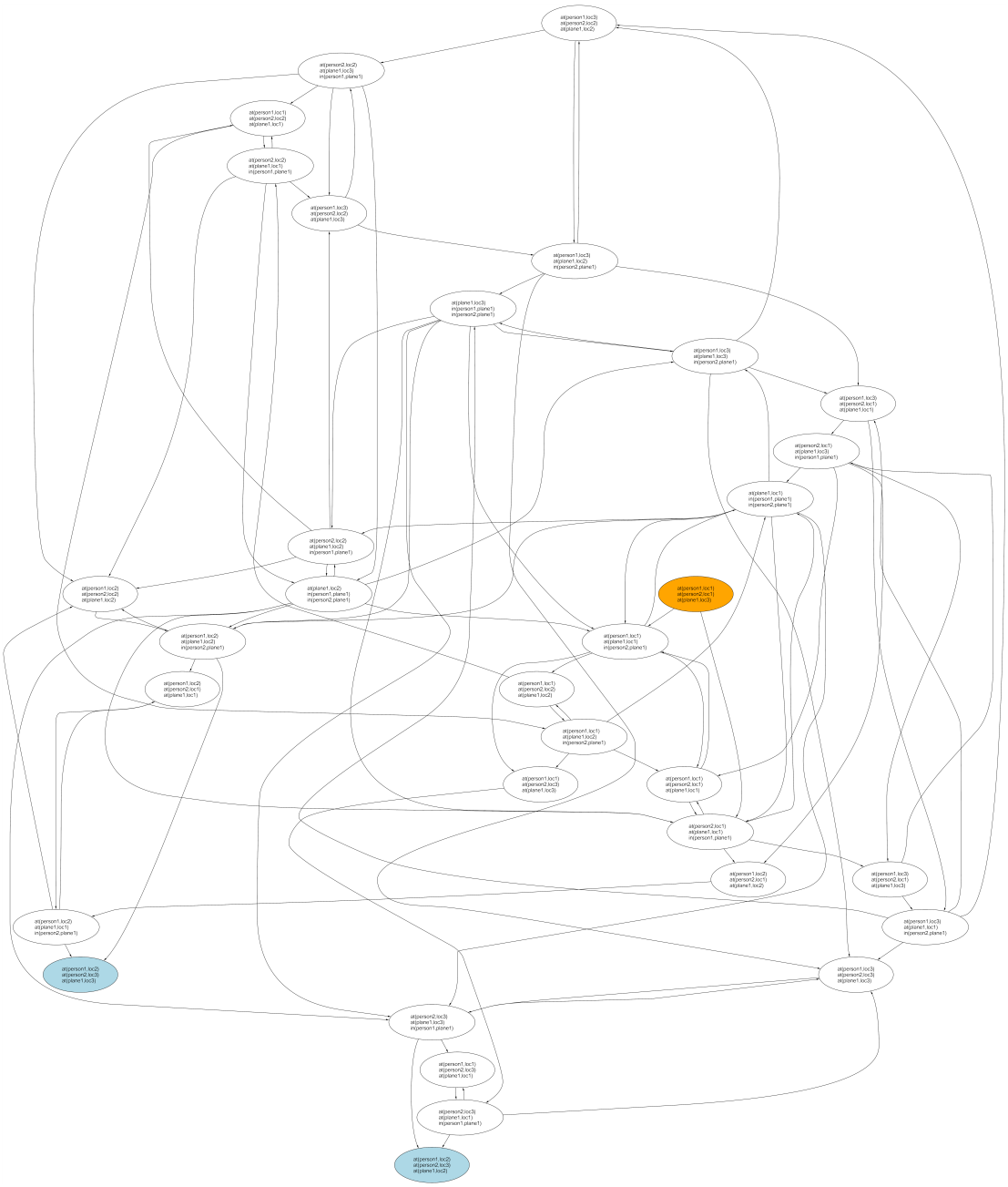


Figure 3.4: State-space with meta-actions. 33 states and 92 edges

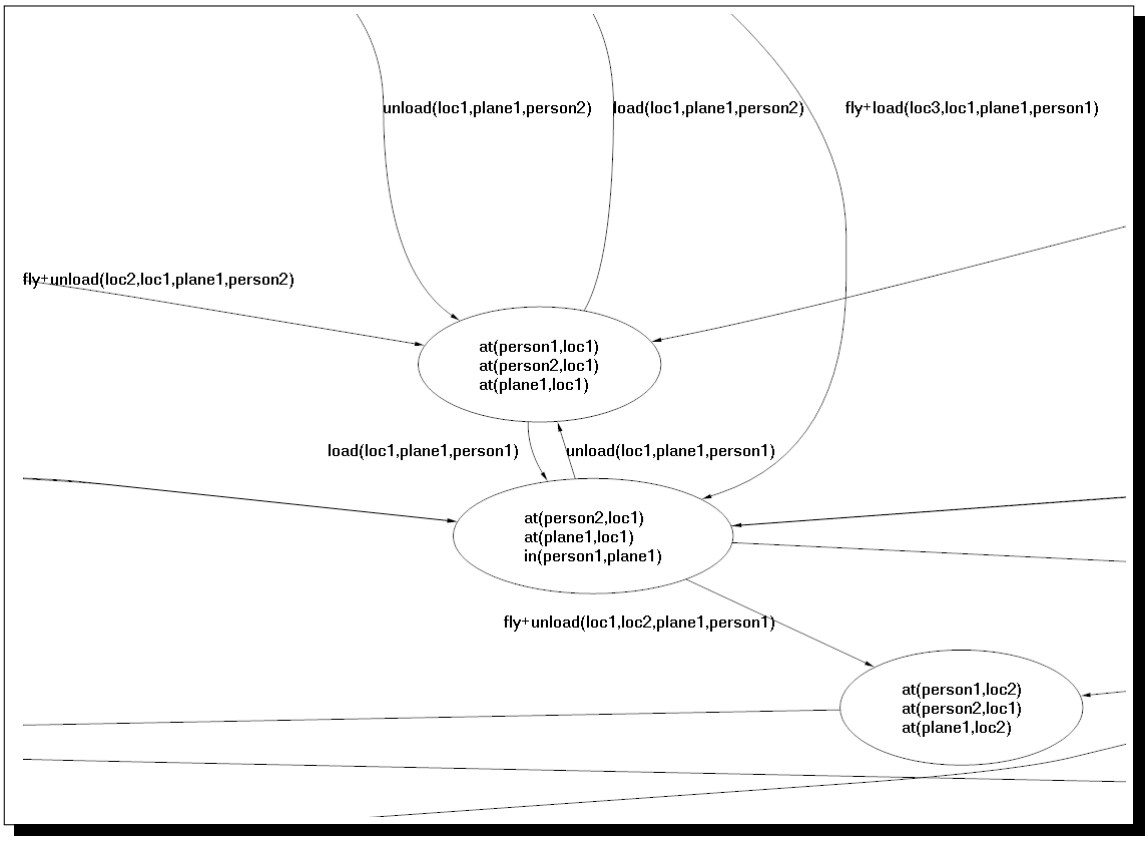


Figure 3.5: State-space with meta actions - cut-out showing a cycle

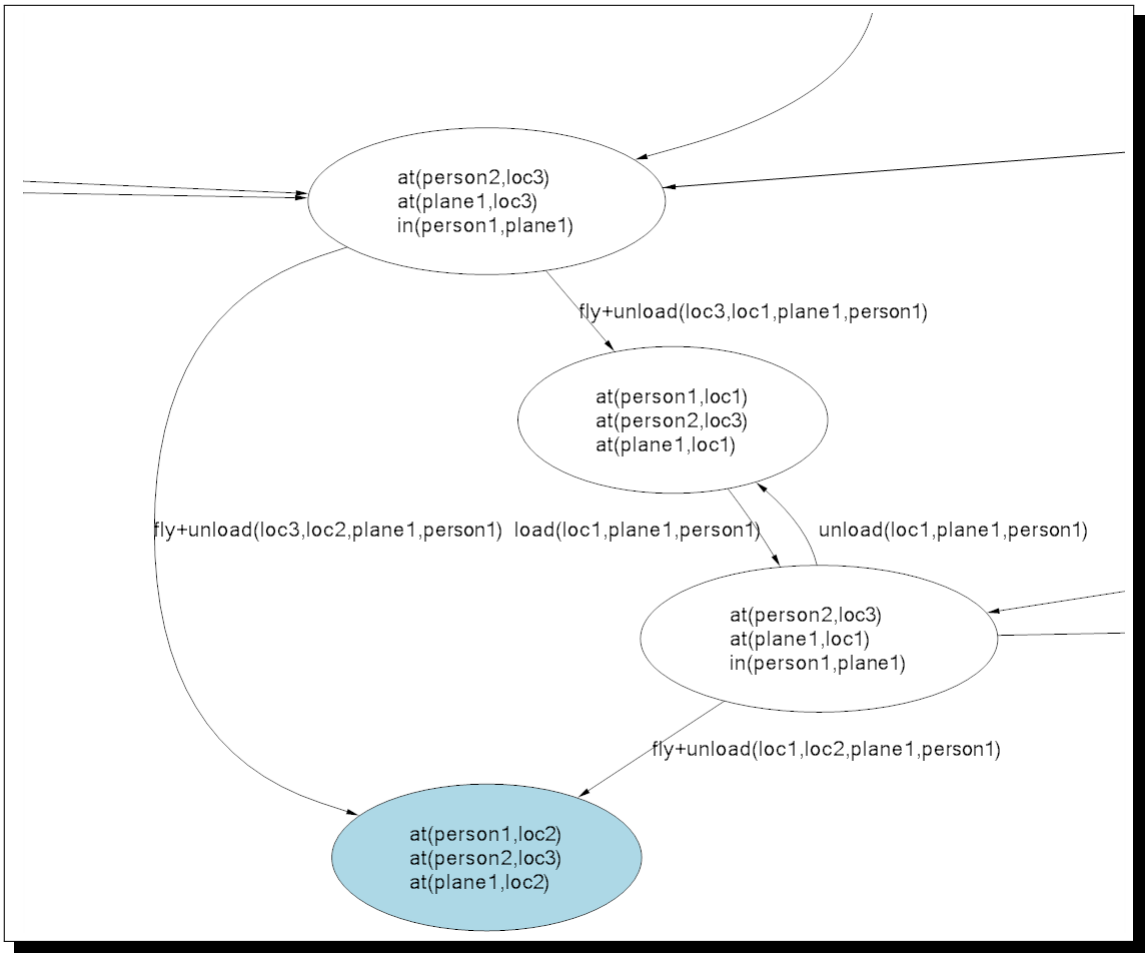


Figure 3.6: State-space with meta actions - cut-out showing a goal state

3.2.3 Merit of the meta-actions

In the figure 3.5 we can see a cycle in the state-space induced by the model of meta-actions. This rises the question of how did the meta-actions actually help us (since we introduced them to get rid of cycles and dead-ends). Truth is that the meta-actions don't eliminate all the cycles and dead-ends from the state-space (at least not when used in the way we use them). Yet they still dramatically increase the performance.

Eliminating *all* the cycles is equivalent to numbering the states in a way that every path towards a goal state (from arbitrary state) goes over the states in a decreasing manner. (This would guarantee that we never go to the same state again since we get closer to the goal state with every action performed.) However obtaining such knowledge is again equivalent to solving the satisficing problem. We will use a weaker approach which doesn't guarantee eliminating all the cycles but which is tractable.

The main asset of the model of the meta-actions is that it eliminates the cycles caused by moving the vehicles without purpose however it doesn't remove the cycles caused by the perpetual loading and unloading. We will deal with this type of cycles in a different way - by the use of a (domain-independent) heuristic function (see 3.5.3).

The real goal that we need to achieve is to make the simulations short enough for the algorithm to be efficient. (The performance is based on the number of simulations performed which is dependent on the length of the simulations.) Since the simulations are randomized their length can be seen as a random variable. We need the mean of this random variable to be as low as possible. (In other words we don't mind if there are some cycles remaining in the state-space as long as they are very sparse or have low probability of being reached by the simulation - that is if the expected length of the simulation is low.)

We will use this approach (with random variables) to show how much the meta-actions contribute to the performance of the algorithm. Suppose we want to estimate the expected length of a simulation in the case of domains with the original actions (like in the figure 3.1) and with the meta-actions (figure 3.4). Suppose that we use the simplest possible simulation policy - a random walk.

To compute an expected number of steps of a random walk on such a graph would be difficult (and even more difficult in the general case). Therefore we introduce a different model. We abstract from the concrete vehicles, locations, and cargo and we only take into account the number of loaded cargo and the number of delivered cargo (which denotes how far from the goal the simulation is).

An example of the model we use is in the picture 3.7. It shows an abstract representation of the domain with 3 cargo (with no regard to the number of vehicles or locations). The

circles representing the states are divided into layers. The horizontal layers (colored yellow) show how many cargo is loaded in the appropriate states (e.g. in the layer 2 there are states that represents situations where 2 cargo is loaded in some vehicle or vehicles). The vertical layers (colored orange) show how many cargo has already been delivered to its target location in the appropriate state. The states are denoted by the numbers of corresponding layers - e.g. the state denoted 2/1 represents a situation where two cargo has already been delivered and one cargo is loaded in some vehicle. Obviously the sum of these numbers cannot exceed the total number of cargo (in this case 3) which cause the triangular shape. The initial state is denoted 0/0 and represents the situation where no cargo is loaded nor delivered. The final state is denoted 3/0 which represents the situation where we have delivered all the cargo.

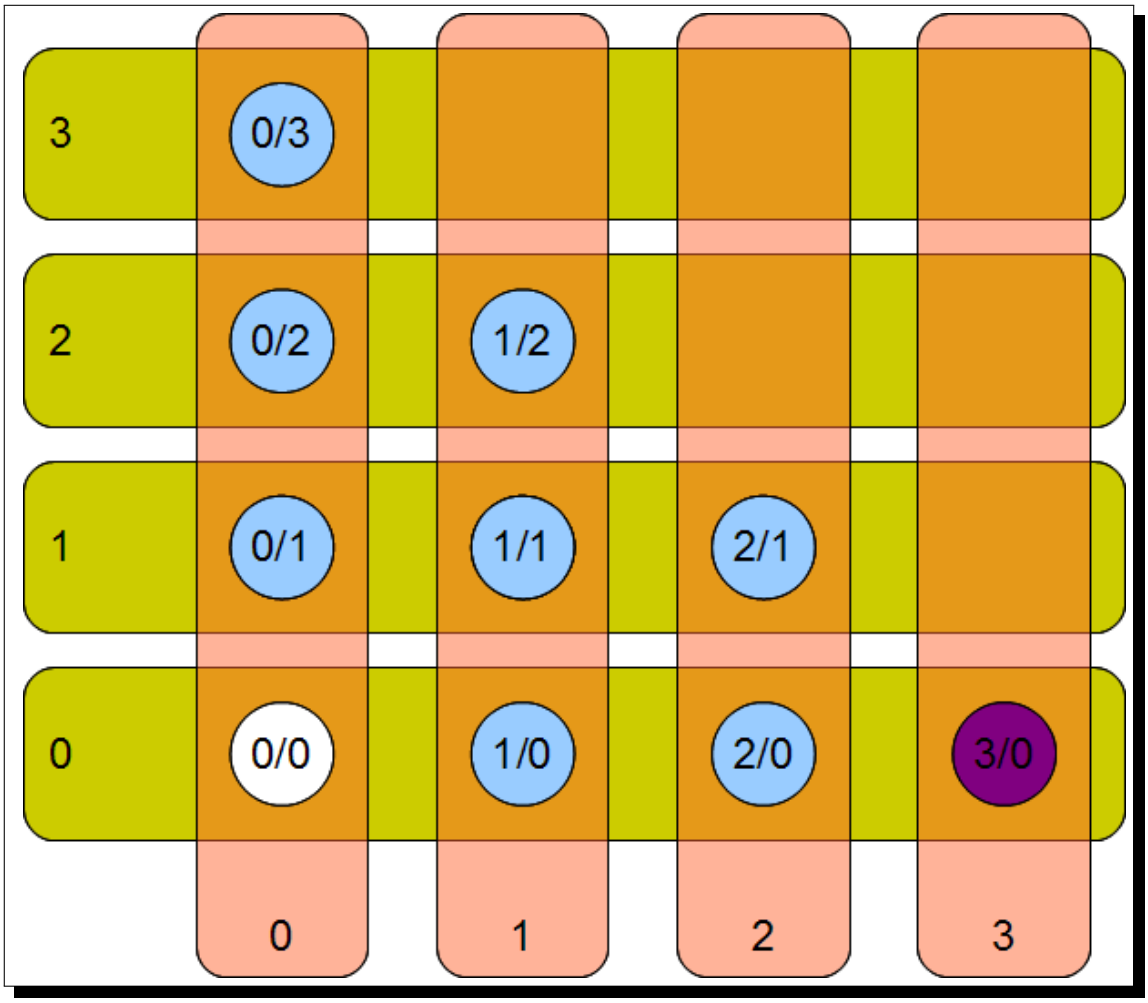


Figure 3.7: Model for estimating the simulation length in a domain with 3 cargo

The figure 3.8 shows the same model with transitions. Each transition corresponds to exactly one action in the original domain. The meaning of the transitions is as follow:

- the greed arrow represents loading some cargo into some vehicle (i.e. corresponds to some action of the type *Load* or *Move+Load*)
- the blue arrow represents unloading some cargo at a location where this cargo is destined (i.e. increases the number of delivered cargo)
- the red arrow represent unloading some cargo at a location where this cargo is not destined.

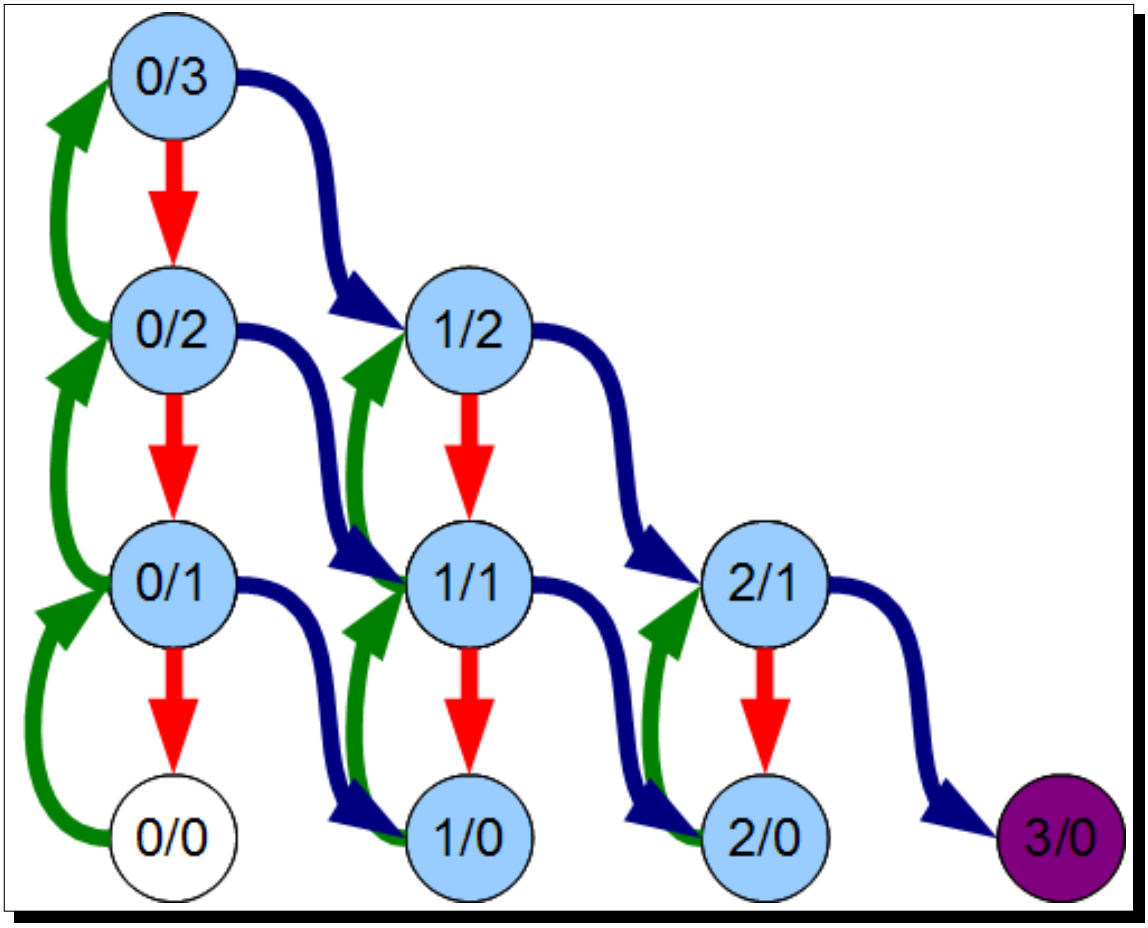


Figure 3.8: Model for estimating the simulation length in a domain with 3 cargo, with transitions

A random walk in the original graph can be replaced by a random walk in this model. Since every edge in the new graph corresponds to the performation of one meta-action, we can estimate the expected length of an original simulation by the expected length of a

random walk in this new graph. (An expected length until the random walk reaches the goal state.)

We can see this process as an Absorbing Markov chain with one absorbing state. To calculate the expected number of steps until being absorbed we need to know the probabilities of the transitions. The simulation strategy we use works as follow: (more thoroughly described in the section 3.4.2)

1. select random undelivered cargo
2. if it is loaded then unload it at a random destination
3. if it is not loaded then load it into a random vehicle

The probability that randomly selected undelivered cargo will be loaded is $\frac{loaded\ cargo}{all\ undelivered\ cargo}$, and the probability of selecting the correct target location for the cargo is $\frac{1}{locations}$

We use the following notation:

- let l be the number of locations in the *transportation component* (will be described in the following section)
- c be the number of cargo in the transportation component and
- v be the number of vehicles in the transportation component.

Then for an arbitrary node of our model (see the picture 3.9) which represents a state with i delivered cargo and j loaded cargo the transition probabilities are:

- the probability of loading the cargo (green arrow) is $\frac{c-i-j}{c-i}$
- the probability of unloading the cargo in the correct location (blue arrow) is $\frac{1}{l}\frac{j}{c-i}$ and
- the probability of unloading the cargo in an incorrect location (red arrow) is $\frac{l-1}{l}\frac{j}{c-i}$ and

The expected number of steps in this case turns to be $2 * c * l$

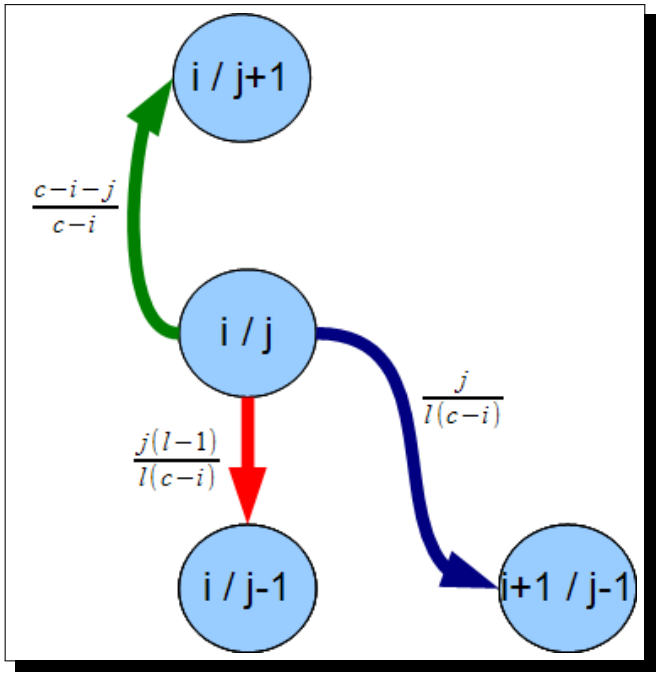


Figure 3.9: A generic node of the model with its transitions.

If we used the original model of actions then there would be one more possible transition which would represent an action of the type *move* (colored yellow in the picture 3.10). Furthermore this transition would have a very large probability since there are a lot of possible actions *move* - every vehicle can move to every location - there are $v * (l - 1)$ possible actions move at every step. This causes the expected number of steps to be much larger. The exact formula is in this case difficult to find (for it depends on the number of vehicles and on the method for performing the simulation). The experiments we conducted show that the number of steps is about two orders of magnitude greater than in the previous case. (This applies to rather small instances that we used in the experiment. For larger instances the difference would be even greater.)

This shows that the use of the meta-actions for eliminating at least some cycles is definitely worth the effort. Further improvement of the performance that we achieved by the use of a heuristic function is described in section 3.5.3.

Note that in this analysis we exploit the fact that in a transportation-based domain the goals are independent and therefore once we deliver the cargo we never need to load it again. (I.e. the number of undelivered cargo can never increase during the simulation.)

Another asset of the meta-action: back in the section 1.5.3 we have stated that Genetic algorithms have a substantial advantage against MCTS because they implicitly use *near-optimal substructures* as their building blocks. The meta-actions that we use here can be

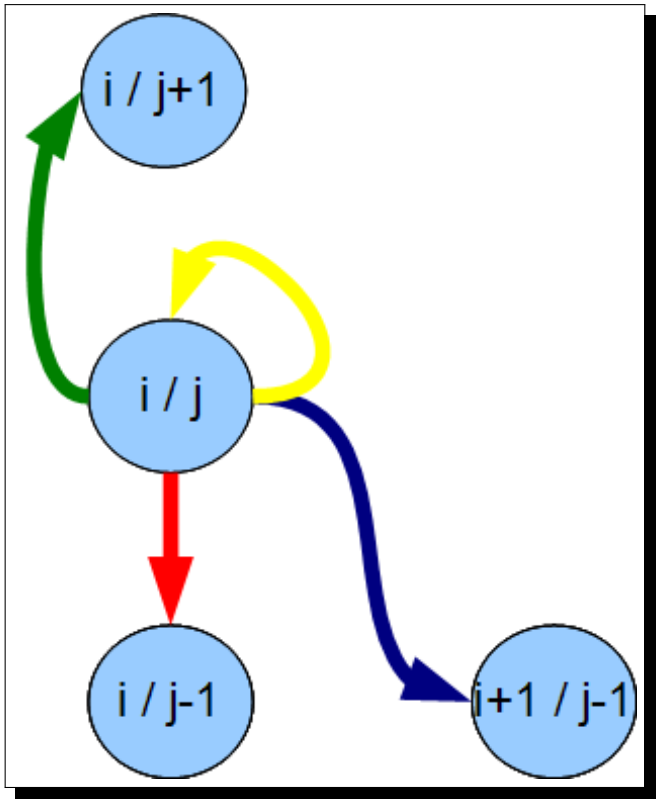


Figure 3.10: A generic node of the model allowing the action move with its transitions.

seen as such *building blocks* therefore by introducing them to the planning process we can counter one of the disadvantages of MCTS.

3.2.4 Soundness and completeness

After replacing the original model of actions by a model of meta-actions we actually solve a different problem than the original. Therefore we need to show the connection between these problems and especially between solutions of the problems.

Let P be a transformation process that transforms a problem A to a problem B. We say that P is *sound* if every solution of the problem B can be transformed to a solution of the problem A. We say that P is *complete* if every solution of the problem A can be achieved by the transformation of some solution of the problem B. In other words a transformation process is sound if correct solutions of the secondary problem somehow represent correct solutions of the primary one. And it is complete if any solution of the primary problem is represented by some solution of the secondary one.

The task of planning can be seen as a path-finding problem in some graph. In our transformation we replace the set of edges of this graph (i.e. the original actions) by

another set which contains some *paths* of the original graph (i.e. the meta-actions). This process is definitely sound since every path in the new graph represents some (possibly longer) path in the original graph. (Or in terms of metric graph theory: the image of a connected graph under a non-expansive mapping is still connected. [25])

The question of completeness is more difficult. Our technique is definitely not complete in a sense that every path from the initial state to some goal state in the original graph can be represented in the new graph. We wouldn't even want this property since most of the paths in the original graph represent meaningless sequences of actions even if they lead to the goal. To get rid of such paths is the very reason why we have introduced the transformation. (Consider for example the longest possible path from the initial state to some goal state. It represents a proper solution to the problem and doesn't even contain cycles, yet we still wouldn't consider it a meaningful solution to transportation problem.)

We would like to preserve only the *important* paths in the graph, especially those that represent optimal solutions. However it is not possible to decide what paths are important without the knowledge of the evaluation function. For what we can tell there might be a problem where the *longest* paths are desired and highly evaluated -i.e where the longer plans have higher evaluation. In practical problems however this is not the case and the evaluation function is usually decreasing in the length of the plan.

The knowledge of the transportation components allows us to identify the important nodes in the state space and transform the edges such that the paths between these important nodes are preserved and other paths are eliminated. For example the important point is when the cargo is loaded and when it is unloaded. We can use this as our subgoals and plan the paths between these points, and then use those paths as the meta-actions. This is very similar to the technique we use in our planner. At the end of this chapter we describe this technique in a more general way using known terms (as *landmarks*, *Hierarchical task networks (HTNs)* and so on).

The technique proved to perform well in practice (see 4) however we cannot guarantee that our transformation always preserves the optimal solution of the problem (i.e. that the optimal plan in the original problem can always be achieved using the meta-actions). Therefore our transformation procedure is not complete. Note that preserving the completeness is difficult since without the knowledge of the evaluation function we would have to preserve all the paths which would most likely have very little effect on increasing the performance (most of the meaningless sequences would remain).

3.3 Obtaining the model of meta-actions

There are basically two ways how to obtain meta-actions from the original action model.

1. Manually by rewriting the domain specifications
2. Automatically by domain analysis techniques

We will first describe a procedure how to modify the domain manually. This technique is independent on the type of planner and can be used to improve performance of any planner on a transportation domain. Description of an automated technique that we use in our planner will follow.

3.3.1 Manual domain modification

Main cause of inefficiency when solving a transportation problem lies in the fact that there are meaningless sequences of actions (like moving the vehicles between locations without delivering any cargo). Without a proper heuristic the planner has no way to distinguish meaningful sequences from the others.

The key idea to solve this problem is to combine action *move* with action *load* or *unload* so that the vehicle wouldn't move around without purpose. In terms of meta-actions this means replacing the action move (or its equivalent) by actions *move+load* and *move+unload*.

The idea is fairly simple however the implementation might be complicated. Complications can emerge in these situations:

1. There are more types of actions *move* and *load* or *unload*.
   - In this case we would have to add one operator *move+load* for every combination of actions *move* and *load*.
   - Example: Suppose we extend the Zeno-Travel Domain - Original (A.2) by adding actions *LoadFast* and *UnloadFast* Then during the domain rewriting we would have to add four more actions of type *Move+Load* (*Fly+Load*, *Fly-Fast+Load*, *Fly+LoadFast* and *FlyFast+LoadFast*) and similarly four actions of type *Move+Unload*.
   - In general if there are n types of action *move* and m types of action *load*, we need to add nm new actions. This might be inconvenient to do by hand for larger numbers m and n.
2. There are more types of vehicles and locations
   - The domain may involve more kinds of transportation - for example by aircrafts between airports and by buses between bus stations or between a bus station and an airport.

- In this case it is necessary to add the meta-actions for every type of vehicle separately and to be careful to use appropriate type of location for every vehicle. (For example not to let an airplane arrive at a bus station.)
- This might again by quite complicated and time-consuming task for a human operator.

3. The goal is not only to deliver the cargo.

   - It might be requested to move the vehicles to some predefined location after making the deliveries. This requirement is quite common in practical problems for we typically don't want the vehicles to stay at the place of delivery but rather to return to the headquarters or to a depot after delivering all the cargo.
   - To accomplish this goal we would need to perform action *move* which is not followed by any *load* nor *unload* action (i.e. perform action *move* alone).
   - This is not possible in our model of meta-actions since we no longer allow action *move* to be performed alone.
   - Note that unlike previously described situations which were inconvenient for the human operator but still possible to handle, this problem is fundamental and the domain preprocessing technique described here can't be used in this situation at all.

4. Nontrivial preconditions of the actions move, load or unload

   - So far we assumed that the operators have only trivial preconditions. (For example in case of action load the trivial precondition is that the vehicle is at the same location as the cargo. These preconditions are mandatory, they define the structure of transportation component.) Therefore we could assume that action *move* can be directly followed by action *load* or *unload*.
   - For general planning problem this doesn't have to be true. Operator *load* might have many other preconditions that needs to be accomplished before *load* can be performed. (Same holds for operators *unload* and *move*.)
   - Example: Suppose that operator *load* has a precondition p that is not achieved by operator *move*. Suppose that there are n ways how to accomplish this precondition. Then we have to create n meta-actions, one for each of them. In this case the number n might be quite larger depending on how many actions will be needed to accomplish p (i.e. how long is the path in state space that the meta-action represents). Furthermore finding all the ways to accomplish some precondition is equivalent to solving some satisfacing planning problem therefore a search has to be involved in this case.
   - Note that the previous situations only required syntactical manipulation with the domain however this situation is different since it requires to solve a planning task. Hence if the operators have non-trivial preconditions then the domain is not suited for manual modifications.

Another idea how to modify the transportation domain would be to create a composite action for the whole delivery process i.e. create a meta-action of a type *move+load+move+unload*. This might seem to be better approach than creating the meta-actions *move+load* and *move+unload* separately since it reduces the searched space even more and still preserves the important paths. This approach however doesn't work since the vehicles would only carry at most one cargo all the time. (After loading it can't load anything else since the meta-action performs moving and unloading right away). This would most likely not lead to optimal solutions.

This is an example of bad design of the meta-actions which still preserves *some* important paths but not those that represents the optimal solutions. As was discussed in section 3.2.4 we need to find an equilibrium between week reduction (which would preserve the optimal solutions but wouldn't prune the searched-space much) and strong reduction (which would greatly reduce the searched space but wouldn't preserve the optimal solutions). This is an example of too strong reduction. (Although we still cannot guarantee that our technique preserves the optimal solution in general case.)

3.3.2 Automated domain modification

The automated process for domain modification that we use in our planner is similar to the manual procedure described above with several differences. The main problem with automated processing is that we don't know the meaning of the actions nor of the predicates and types. We first need to identify objects that represents cargo, vehicles and locations, the predicates that represent relations between these objects (like whether the cargo is at the location or inside a vehicle) and the operators that changes these relations (loading, unloading, and moving).

We will extract this information by identifying the *transportation components* in the domain. (This is the reason why we introduced the notion of transportation component at the first place.) Algorithm for doing that is described in the section 3.4.1.

Other steps of the domain modification algorithm are the same as those described in the manual procedure. We will asses how the algorithm can handle the problematic situations mentioned in the previous section:

1. More types of actions *move* and *load* or *unload*
2. More types of vehicles and locations
    - In these situations the program has to create all possible combinations (as was described above). That would be laborious to do by hand but it is not an issue with automated processing.
3. Other goals than delivering the cargo

- We handle this problem by allowing the planner to use original model of actions in situations where no meta-action is applicable but the goal is not yet achieved.
- In other words the planner uses meta-actions model during the planning (and therefore takes advantage of reduction of the searched space) but in situations where meta-action model doesn't suffice (like moving the vehicles after delivering the cargo) it uses the standard actions.
- Note that this behavior can't be achieved merely by modifying the actions model. It requires a cooperation of the planner.

4. Nontrivial preconditions of the actions move, load or unload

   - In this case finding meta-actions requires to solve a satisfacing problem. Since this can't in general be avoided, we perform here an exhaustive search to find all possible ways to accomplish the necessary precondition (i.e. we find all the paths from the state that represents the situations after performing action *Move* to the state that represents the preconditions of action *Load* or *Unload*. Then we create one meta-action for every path we found.
   - This phase might be computationally demanding (see the following example) but usually this subproblem is much easier than the original problem since the paths are usually short.
   - Detailed description of the algorithm we use in this phase is given in the section 3.4.3

3.3.3 Drawbacks of the method

Here we provide an example of a situation where our learning algorithm doesn't work well. Suppose that the domain describes a transportation of boxes. These boxes are stored in towers (one on top of the other) and when loading we first have to rearrange the towers by some crane so that the boxes we want to load were on the top of the towers (i.e. the loading is connected with solving the box-world problem).

Our algorithm would in this case successively learn new meta-actions for loading. However each of them would only be applicable in one special case.

Suppose it first encounters a situation where the box it intends to load is already on top. Then it would easily learn a new meta-action for doing that. Whenever the planner encounters this situation in the future (i.e. the situation that the cargo to load is on top of some tower) it would use an already learned meta-action to perform the task. However when it encounters different situation (e.g. when the cargo is second from the top) it has to run the learning procedure again to discover the way how to deal with the new situation.

In other words it would learn one meta-action for every possible position of the cargo in the tower (like: on the top, second from the top and so on). This would mean literally memorizing all the plans in box-world to clear some box. The number of the meta-actions would be very large in this case which would dramatically reduce the speed of the algorithm. Suppose we want to generate some applicable meta-action in the simulation phase - the algorithm would try *all* the known met-actions until it finds the applicable one. Deciding the applicability involves finding the applicable instance - i.e. the correct substitution.

If for example the box is the tenth from the top, it would try the meta-actions for loading the box that is on the top, then the one for loading the second from the top, then the third and so on. None of these meta-actions would be applicable which the planner only finds out after trying all possible substitutions for all these actions.

Our procedure would still be able to handle this situation (at least theoretically) but the practical performance would be poor.

3.4 Used algorithms

3.4.1 Searching for the transportation components

The transportation component is defined by values of the variables components.LoadOp component.CargoAtName and so on. (As described in the section 3.1.3). The values has to satisfy several constraints (described in the section 3.1.3 as well). Finding the components is equivalent to finding all the solutions of a CSP (Constraint Satisfaction Problem) with given variables and constraints.

To solve this problem we use standard CSP solving techniques - a backtracking search algorithm enhanced with domain independent heuristics like *Succeed First* and *Fail First* along with *BackJumping* technique and *domain filtering*.

This procedure could be computationally demanding if the input domain were complex. However the search for the components is performed during the preprocessing phase and therefore it doesn't slow the planning algorithm.

3.4.2 Generating the possible actions

Algorithm for generating all applicable meta-actions for given state. At this point we have already identified the transportation components in the domain and use them to generate the actions. Algorithm is designated as Generate possible actions

Algorithm Generate possible actions:

Input: a state s, a list of transportation components C
Output: a list of ground meta-actions applicable in s

```
result = empty list
foreach component in C in random order do
    foreach not delivered cargo in component.allCargo in random order do
        if cargo is loaded in some vehicle then
            foreach destination in component.allLocations in random order do
                if vehicle is located at destination then
                    templates = component.Unload_operators
                else
                    templates = component.Move+Unload_operators
                foreach operator in templates in random order do
                    foreach instance in all instances of operator do
                        if instance is applicable in s then
                            add instance to result
                if no instances were added then
                    Learn new meta-actions and add them to templates and their
                    instances to result
        else
            foreach vehicle in component.allVehicles in random order do
                if vehicle is in the same location as cargo then
                    templates = component.Load_operators
                else
                    templates = component.Move+Load_operators
                foreach operator in templates in random order do
                    foreach instance in all instances of operator do
                        if instance is applicable in s then
                            add instance to result
                if no instances were added then
                    Learn new meta-actions and add them to templates and their
                    instances to result
```

Result: result

Informal description of the algorithm would be

1. Find some not delivered cargo
2. If it is loaded try all possible locations to unload it and generate appropriate meta-actions of type *Unload* or *Move+Unload*
3. If it is not loaded try all possible vehicles to load it and generate appropriate meta-actions of type *Load* or *Move+Load*

When generating the applicable actions we make use of previously found meta-actions. These meta-actions are divided to four groups - *Load*, *Unload*, *Move+Load* and *Move+Unload* and are stored for each transportation component in the lists designated as *component.Load-Operators*, *component.Unload-Operators* and so on.

These lists are initially empty and are consequently filled by newly learned meta-actions during the run of MCTS. The learning procedure is called when no action in the lists is applicable in given state. This approach can be called *Lazy Learning* since we only learn the meta-actions that we need.

Note that by manual iteration through the cargo (line 3), destinations (line 5) and vehicles (line 17) we can significantly speed up the instantiation process since we only try the relevant constants.

3.4.3 Learning the meta-actions

The learning procedure we use is parameterized by the type of the meta-action we want to learn. For simplicity we only present algorithm to learn meta-actions of the type *Move+Load*, other types are similar. The algorithm is designated as Learn Move+Load.

Informal description of the algorithm:

1. We find all possible ways to accomplish preconditions of the action *Move* (Lets denote them M)
2. For all $m \in M$ we performs the sequence m and then perform action *Move*
3. From resulting state we find all possible ways to accomplish preconditions of the action *Load* (Lets denote them L_m)
4. For all $l \in L_m$ we create a meta-action from sequnce m+*Move*+l+*Load* and add it to the list of applicable actions
5. We lift the meta-action and add it to the list of learned meta-action.

Algorithm Learn Move+Load:

Input: a state s, a transportation component component

Output: a list of ground meta-actions of type *Move+Load* applicable in s, a list of newly learned meta-operators of type *Move+Load*

```
1  applicableActions = empty list
2  learnedOperators = empty list
3  firstParts = find all the shortest paths from s that ends with action Move in
   state-space of the domain restricted to component
4  foreach initialPath in firstParts do
5      intermediateState = a state where initialPath ends
6      secondParts = find all the shortest paths from intermediateState that ends with
       action Load in state-space of the domain restricted to component
7      foreach terminalPath in secondParts do
8          wholePath = concatenate initialPath and terminalPath
9          meta-action = create meta-action from sequence of actions wholePath
10         add meta-action to applicableActions
11         liftedMeta-action = generalize meta-action to a meta-operator by lifting its
           constants to variables
12         if learnedOperators doesn't contain operator isomorphic to liftedMeta-action
           then
13             add liftedMeta-action to learnedOperators
```

Result: applicableActions, learnedOperators

The pseudo-code requires further clarification:

- When searching for the shortest paths (lines 3 and 6) this paths always consist of original actions (even if some meta-actions were already learned).
- The shortest paths (lines 3 and 6) are requested to end with a specific action (*Move* or *Load* respectively). These reffer to actions specified in the description of a transportation component (see the section 3.1.3). Note that the domain may contain several actions of type *Move* and *Load*, we create a transportation component for every one of them so each component is bound to one specific action of type *Move*, *Load* and *Unload*. (All of these are of course primitive actions.) So we could write component.MoveAction instead of *Move* on line 3 and component.LoadAction instead of *Load* on line 6.
- Algorithm for searching for the shortest paths is described in the section 3.4.4.
- Intermediate state (line 5) can be alternatively described as a result of application of the sequence of actions initialPath to the state s
- Procedure of creating a meta-action from sequence of actions (line 9) is described in the section 3.4.5
- The test for isomorphism of the operators (line 12) is necessary since the operators might have different names of the variables (and therefore are not identical) but represents the same opeartions (i.e. have the same sets of instances).

3.4.4 Finding the shortest paths during meta-actions learning

The algorithms operates in state-space of the domain represented as an oriented graph where it searches for the shortest paths from given state to any state that satisfies given condition. The state-space of a planning problem is usually very large so the algorithm restricts the search to only one transportation component at a time. For finding the shortest paths we use standard BFS (Breadth-first search) algorithm.

Inside the component the paths should be reasonably short however we cannot guarantee that (see section 3.3.3). Therefore we set a limit on maximum possible length of the paths. If no path is found within the limit then we end the algorithm and report to the user that given domain is not suited for our algorithm.

3.4.5 Creating a meta-action from a sequence of actions

The procedure correctly handles the preconditions and effects of the sequence of actions to create the preconditions and effects of the resulting meta-action. An informal description follows. The pseudo-code is designated Create meta-action from sequence.

1. We process the primitive actions sequentially and we keep current sets of preconditions and effects of the resulting meta-action (initially empty)

2. If newly added action has a precondition that is not achieved by previous actions we add this precondition to the preconditions of resulting meta-action

3. If newly added action has a precondition that is incompatible with previously added actions effects then the input sequence is incorrect (this should never happen in our case)

4. Positive effects of newly added action are added to effects of resulting meta-action

5. Negative effects on newly added action are removed from effects of resulting meta-action

The algorithm will work even if the actions in the sequence were independent (i.e. didn't share any constants) but such meta-actions wouldn't be very useful. (For example the action for moving *vehicle1* from *location1* to *location2* and the action that moves *vehicle2* from *location3* to *location4* can be combined into single meta-action which would perform both operation at once). The real merit of the meta-actions is in combinations of actions that share some constants (or variables - after lifting).

Comments to the code:

1. The output is described as a meta-action equivalent to the sequence. This means that applying the meta-action will have the exact same result as applying all the actions in the sequence one by one. (And also the meta-action is applicable in the same states as the sequence.)

2. Note that applying the meta-action is computationally less demanding than applying the sequence. In the sequence there might be situations where some actions have an effect that is later removed by another action - we have to add them and remove them later. In case of a meta-action we will detect this during the creation and when applying the meta-action we will only add the real effects.

3. The foreach cycle at line 2 has to iterate the actions in standard order (from the first to the last).

Algorithm Create meta-action from sequence:

Input: a sequence s of primitive actions
Output: a meta-action result equivalent to the sequence s

```
1  set all result.PositivePreconditions, result.NegativePreconditions,
   result.PositiveEffects and result.NegativeEffects to empty list
2  foreach action in s do
3      foreach predicate in action.PositivePreconditions do
4          if predicate ∈ result.NegativeEffects then
5              cannot create meta action
6          if predicate ∉ result.PositiveEffects ∧ predicate ∈
           result.NegativePreconditions then
7              cannot create meta action
8          if predicate ∉ result.PositiveEffects then
9              result.PositivePreconditions =
               result.PositivePreconditions ∪ predicate
10     foreach predicate in action.NegativePreconditions do
11         if predicate ∈ result.PositiveEffects then
12             cannot create meta action
13         if predicate ∉ result.NegativeEffects ∧ predicate ∈
           result.PositivePreconditions then
14             cannot create meta action
15         if predicate ∉ result.NegativeEffects then
16             result.NegativePreconditions =
               result.NegativePreconditions ∪ predicate
17     foreach predicate in action.PositiveEffects do
18         result.NegativeEffects = result.NegativeEffects \ predicate
19         result.PositiveEffects = result.PositiveEffects ∪ predicate
20     foreach predicate in action.NegativeEffects do
21         result.PositiveEffects = result.PositiveEffects \ predicate
22         result.NegativeEffects = result.NegativeEffects ∪ predicate
```

Result: result

3.5 Other techniques used

Here we describe other modifications and improvements we use. Some of them are necessary in order to use MCTS (like 3.5.1) others are expendable but might increase the performance (like 3.5.4).

3.5.1 Modifications in the selection phase

We have made a few modifications to the standard UCT tree policy:

- Since the standard UCT formula is designed for maximization problems we have to adjust it to a minimization version (in planning the goal is to *minimize* the objective function).
- We use the variance-based modification described in section 1.4.
- We use an adaptive technique for adjusting the parameter c in order to keep the two components in the formula of a same magnitude (see the end of section 1.3.2 for details).

The final formula we use is $Expectation - Standard\ deviation - Urgency$, or formally let

1. $Visits_n$ be the number of simulations that passed through the node n,
2. $ValuesSum_n$ be the sum of evaluations of all the simulation that passed through the node n,
3. $SquaredValuesSum_n$ be the sum of squared evaluations of all the simulation that passed through the node n,

Then the formula is

$$\frac{ValuesSum_n}{Visits_n} - \sqrt{\frac{SquaredValuesSum_n}{Visits_n} - \left(\frac{ValuesSum_n}{Visits_n}\right)^2} - c\sqrt{\frac{2\ln\left(Visits_{parrent}\right)}{Visits_n}}$$

We select the node that *minimizes* this formula.

3.5.2 Evaluation of incomplete simulations

The model of meta-actions doesn't necessarily rid the domain of all dead-ends. (see section 3.2.3) Therefore we need to devise the means for evaluating the simulations that ended in a dead-end (or in general the simulations that don't represent a solution plan) Some possible approaches were mentioned in section 2.2.2.

This problem is caused by the fact that the evaluation function is only defined for solution plans. The most straightforward way to deal with this issue would be to evaluate such simulations with some large number which would tell the planner that this solution is considered very bad.

This approach however has a few drawbacks:

- it would bias the Expectation values of all the nodes that the simulation passed through (some of these nodes might be promising but this one simulation can change their Expectation by a high magnitude)
- it would increase the variance of the nodes that the simulation passed through (in this case if the node truly is unpromising, the growth of the variance would cause it to be selected more often than it should be.)

This solution seem to slow the MCTS algorithm (for it blurs the information gained from the simulations) nevertheless we use it in our planner since there doesn't seem to be any better alternative. If the dead-ends were sparse in the state-space it would be better to use the ignoring strategy (2.2.2) however we cannot guarantee that in general. On the other hand from the experiments performed it seems that our preprocessing techniques are able to prevent the occurrence of the dead-ends to a great extend.

We evaluate the incomplete simulations by the value of the first (successfull) simulation performed during the search (which is usually the worst one in the whole search). This value should be large enough to give the planner the information that this solution is bad but not too large to bias the expectation values of the nodes on the path very much. (Since simulations with similar evaluation has actually occurred in the early phases of the search and are already involved in the Expectation values of some node.)

We also use another technique to increase performance in these cases: if the dead-end node occurs in the tree (as a successor of some node during the Expansion phase) then we remove it from the tree as soon as we detect it (i.e. as soon as some simulation visits it). We use the same technique for the nodes that represents a solution - if the tree is expanded enough to contain some node that already represents the solution to the problem (in this case there is no Simulation phase when reaching this node) then we remove such node from the tree. This increases the performance a lot since the solution node usually has a high Expectation and therefore is selected very often. However the simulations that reach it don't provide any information (since there is nothing more to explore about this node) they can even worsen the process since they repeatedly propagate the same good result back to the tree which would cause the selection policy to preffer this node even more.

Some other ideas of dealing with dead-ends are mentioned at the end of this thesis as a possible future work.

3.5.3 Heuristic function

We use a heuristic function during the simulation phase to increase performance. The use of a simulation heuristic in SP-MCTS is strongly suggested (for the reasons explained in the section 1.4). There are two kinds of such heuristic:

1. Domain dependent heuristic
2. Domain independent heuristic

Since our planner should be able to handle more than just one domain, we use a domain independent heuristic. However we exploit the fact that the domain is of a transportation type. The basic idea we use is that in simulation phase we preffer actions that *should* lead to good solutions.

We propose a technique for assigning prefferences to the meta-actions of generic transportation domain. We assume that we have already identified the transportation components in the domain. The procedure for finding one action (in the simulation phase) is similar to the standard algorithm (described in 3.4.2) for finding all applicable actions with several differences:

- The first applicable action found is returned immediately and the procedure is ended. (To save time - we only need *one* action in the simulation phase, not all of them.)
- The *ForEach* cycles (lines 5 and 17) preffer some values over others to increase efficiency. (i.e. some values have high probability to be the first during the enumeration)

Preffered actions are as follows:

- On the line 5 the preffered destination is the target destination of the cargo (i.e. when unloading the cargo we preffer to unload it at its target destination)
- On the line 17 the preffered vehicle is the one that is at the same location as the cargo (i.e. when loading the cargo we preffer to load it to a vehicle that is already at the cargo's current location - if there is any) with exception described on the following line
- For each cargo we keep track of the last vehicle that unloaded the cargo and this particular vehicle is not preffered to load the cargo again. (To prevent meaningless sequences of loading and unloading.)

These suggestions should improve the performance in most cases but they are not *universally* correct i.e. don't help in *all* domains. The counterexamples for each of them is as follows:

1. Suppose we have a transportation domain that contains lots of locations divided into several clusters such that the distance between two location in the same cluster is small but the distance across different clusters is long. In this situation it might be useful to assign one vehicle to transport the cargo between the clusters and another that would deliver it to the final location within the cluster. Therefore it might be prefferable to unload the cargo at a different location than its target destination.

2. The simplest example would be the situation with two vehicles and four locations showed in the figure 3.11. The arrows shows where each cargo should be delivered. The best plan with respect to the makespan is to load *cargo1* into *vehicle1* and *cargo2* into *vehicle2* and deliver it to its destinations. Loading *cargo1* into *vehicle2* as the heuristic would suggest leads to suboptimal solutions.

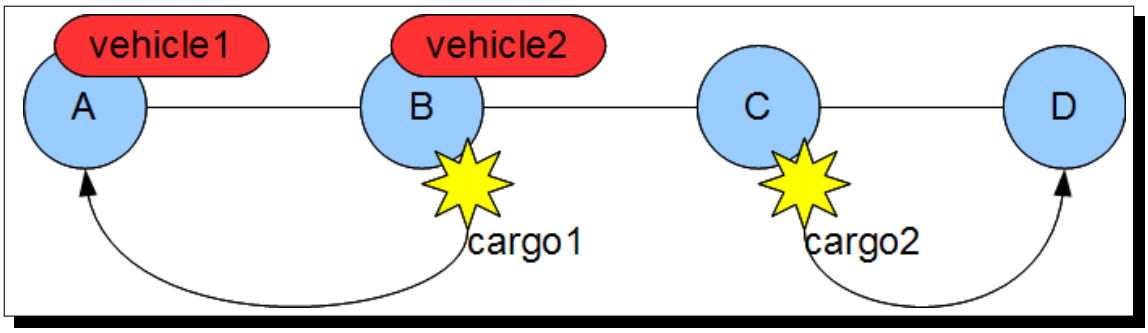


Figure 3.11: Counterexample to a heuristic of loading all the cargo to the nearest vehicle

3. Suppose we have similar situation as in the first case and in addition the fuel consumtion of the vehicles is dependent on the quantity of loaded cargo. Suppose we only have a few vehicles. In this situation the optimal plan would be to first find an optimal route between the clusters and then in each cluster the vehicle should deliver all the cargo that belongs to some location within the cluster. Before doing that it should unload all other cargo (that doesn't belong there) to save fuel. After making the deliveries within the cluster it should load this cargo again and continue to another cluster. If we didn't allow unloading the cargo and then loading it into the same vehicle the optimal plan could never be found in this situation.

Since these heuristics are not always correct and sometimes they might even prevent reaching optimal solution we won't use them as a hard rules but rather in the sense of probabilities. The randomized *foreach* cycles on the lines 5 and 17 will preffer some values (they will have high probability to be the first during the enumeration) but all the other values will still have nonzero probability to be the first and therefore every applicable acion has a chance to be selected.

The use of this heuristic function can dramatically decrease the expected length of the simulations. Especially the first suggestion - to prefer unloading in the target area of the cargo can be useful. Back in the section 3.2.3 we presented a model of measuring the expected number of steps of the simulations. In this model we assumed that we select the location for unloading randomly. The probability of delivering the cargo at its destined location were in this case quite low. The heuristic function we use preffers the destined location among the others. With a certain probability it selects the destined location otherwise it picks the location randomly. This probability is a parameter of the planner and can be set by the user. We use the default value of 0.7.

The experimental results show that with this technique the average length of the simulation is about 50 meta-actions on a problem with 15 cargo and 10 locations. (Which is a significant improvement - without the heuristic the expected number of steps would be 300 as showed in 3.2.3.)

The higher values of this parameter would of course lower the expected length of the simulation even more, but the use of *greedy steps* should be limited since the heuristic can be misleading in some cases and more importantly we need the simulations to be random. Replacing them with a deterministic process would not work with MCTS. We should set the parameter low enough to preserve the randomness but high enough to use the full potential of the heuristic. That is we should allow at least one completely greedy simulation to occur. In other words the probability the at least one simulation composed only of greedy steps will occur among all the simulation performed should be high.

In our case the value of 0.7 fits to problems with about 20-30 cargo. The completely greedy simulation would in this case contain 20-30 unloading actions. The probability that at every step the heuristic will be applied is $0.7^{20} \approx \frac{1}{1253}$ for 20 cargo and $0.7^{30} \approx \frac{1}{44367}$ for 30 cargo. That means that after performing 20 000 - 50 000 simulations we should have reasonable chance to encounter a greedy simulation.

3.5.4 Symmetry-breaking

In our Petrobras planner [3] (described in the introduction part) we used strong symmetry breaking rules to dramatically increase the performance of the planner. Symmetries are quite often in this kind of problems and may cause that we search much larger space than necessary.

We have developed a simple and efficient technique that exploits the fact that we're dealing with a transportation domain. We have identified a large group of actions that are independent and can be reordered without changing the result of the plan. The planner implicitly searches all their possible orderings even though the resulting subtrees are identical.

The actions that we consider independent are these performed by different vehicles. To break the symmetries we define a total order on the vehicles (as a preprocessing - the vehicles are known as soon as we find the transportation components), then we assign an action to every node in the MCTS tree except the root (action that "leads" from the parrent node to this node). Every (meta-)action is connected with some vehicle (since every meta-action loads or unloads something) therefore the total order on the vehicles induces a partial order on all possible meta-actions as well as on the nodes. (Node n_1 is less or equal than n_2 if and only if the vehicle in the action of n_1 is less or equal to the vehicle in the action of n_2).

Now we break the symmetries by forbidding a node to be greater than its parrent. That means that during the expansion phase we generate all possible actions (i.e. all possible successors) and then we remove those actions that involves vehicles with greater number that has the vehicle of the action assigned to the expanded node.

3.5.5 From planning to scheduling

As was mentioned at the beginning of the chapter 2 the *optimal planning* problem is to find a solving plan with the best possible evaluation. In order to do that some evaluation function has to be given. In practice however the situation is slightly different - usually it is the *schedule* of the plan that is evaluated, not the plan itself. (For example the *makespan* - which is usually considered during the evaluation - can only be defined for a schedule.)

Common practice is to separate the planning from the scheduling - to first find some plan and then find a schedule for this plan. An extensive search is usually involved in both phases and sometimes these phases are even performed by different software.

In our approach we can't afford to perform any time-demanding operations during the evaluation (like complete scheduling) since the performance of the MCTS algorithm is strongly dependent on the number of simulations performed and every simulation has to be evaluated. To achieve a reasonable performance the algorithm should perform at least tens of simulations per second therefore the evaluation process has to be as fast as possible.

The scheduling process can be seen as a function from the set of all plans to the set of schedules. The whole optimal planning can then be presented as:

$$problem \xrightarrow{planning} plan \xrightarrow{scheduling} schedule \xrightarrow{evaluating} value$$

In our planner we use a simple greedy algorithm for the scheduling. We iterate through the actions from the first and we schedule each action at the first available time according to the sources it uses. This procedure doesn't guarantee finding the optimal solution but is fast and easy to compute.

In our case the procedure can be represented as:

$$problem \xrightarrow{treepolicy} partial\ plan \xrightarrow{simulation} plan \xrightarrow{greedy\ scheduling} schedule \xrightarrow{evaluating} value$$

The *greedy scheduling* is a deterministic process that can be expresses as a function. Formally let

- P be a set of all possible plans
- S be a set of all possible scheduled plans (schedules)
- f be a function from P to S that represents the scheduling process.

We have found out that we don't actually need to use an optimal scheduling process in order to find the overall optimal solution. In fact the scheduling procedure is not that important at all in this case. All we need of the function f is that it would be *surjective*. If it is, we are able to find the optimal solution even if the scheduling procedure doesn't find the optimal schedule. If the function is surjective than it exists a preimage of the optimal schedule. If this preimage (a plan) is reachable by the planning process then the planner *can* find the optimal solution regardless to the scheduling process used.

For example: suppose we have found a plan which can be scheduled such that the resulting schedule is globaly optimal. Our scheduler however doesn't find the optimal schedule. That doesn't matter since there exists a different plan which will yield the optimal schedule even by our scheduler. This will only work if this *second plan* is reachable (i.e. if the planner can find such plan from the imput problem). In our case the simulation is randomized and every meta-action has a nonzero probability to be selected therefore every possible plan composed of the meta-actions is reachable. The matter of expressive power of the meta-actions is discussed in the section 3.2.4.

It remains to show that our scheduling function really is surjective - i.e. that some optimal schedule can always be achieved by greedy scheduling from some plan. We will not prove that here but it seems reasonable - idea of the prove would be: suppose we have the optimal schedule, lets sort the actions by their start times and create a plan where these actions would be in such order. Then the greedy scheduler would schedule them just the way the are in the optimal schedule. (This holds if in the optimal schedule every action has started in the earliest possible time - which doesn't have to be true - there might be gaps in the optimal schedule that don't affect the resulting evaluation. But filling such gaps should not mae the evaluation worse.)

We have stated that our planner *can* find the optimal solution if the scheduling function is surjective. However the real behavior is another issue. In fact we need the function f to have one other property if we want the planning to be effecient. Consider the following

example: before applying the function f we apply a random permutation. Then the composition would still be surjective and therefore should still allow the planner to find the optimal solution.

However we need to preserve the connection between the plan and associated schedule so that the algorithm would know where to search for a promising plans. In our case we would need even stronger property - for the function f to be *sequential* - in the sense that if two plans contained an identical initial sequences of actions than these sequences would be transformed by f to the identical parts of the schedules in both cases (i.e. the transformation of an initial part should be independent on the rest of the plan). This would allow us to compute the schedule yet during the simulation phase. Furthermore the Expectation values of the nodes would be more stable since all the simulations that would pass the node would have that same initial part of the schedule - i.e. each node of the tree would represent a part of the searched space that would be in some sense *coherent* or *local*. This is important since the MCTS algorithm is based on the assumption of *local similarity* which says that parts of the searched space that are close to each other should have similar evaluation. Therefore it makes sense to use averaging as a way of interpolation.

The greedy scheduling procedure that we use does have this property.

3.6 Summary

Here we summarize the whole planning process we have developed to provide an overall picture.

1. Inputs to the algorithm are
   - description of a planning domain in PDDL
   - an evaluator that can assign a nonnegative real number to given scheduled plan
   - description of a planning problem
2. the planner reads the domain and the problem, creates the operators and identifies the static predicates (which will be handled differently to increase speed)
3. program searches for the transportation components in the domain and strores the results found. (This might be a time demanding step. The algorithm is described in 3.4.1.)
4. the MCTS algorithm is started. In each iteration it performs the following steps:
   (a) adjusts the parameter c according to the current expectation of the root for the reasons described in 1.3.2. We use a transformation which ensures that $\frac{desired\ Expectation}{real\ Expectation} = \frac{desired\ constant}{real\ constant}$ where *desired Expectation* is a parameter of the

algorithm - we use the value 0.75, *real Expectation* is the average result of all the simulations performed, *desired constant* is the value of c that we want to achieve (as well the parameter of the algorithm - we use the value 0.3) and the *real constant* is the value we set by the transformation (and use in the selection formula).

(b) selects the most urgent leaf using the formula described in section 3.5.1

(c) during the selection phase it checks whether the widening criterion has been met. We use a criterion of the number of visits. Before the first widening a certain number of simulations has to visit the node. After the widening this limit is increased so the next widening is delayed. This causes the nodes to be widened early in the search but only promising nodes (i.e. with a lot of visits) will be widened to a great extend.

(d) if the widening is in order, one new successor of the node is unlocked (i.e. added) in the tree.

(e) after selecting the leaf it checks whether the expansion criterion has been met. We expand the node when the number of visits reaches a certain limit. The limit is one of the parameter of the algorithm, default value is 5.

(f) if the expansion is in order we add some successors of the selected node (We create all the successors but some of them are initially locked. The algorithm used is described in section 3.4.2). The number of successors is a parameter of the algorithm, the default value is 3.

(g) runs a simulation from that node. The simulation process works as follows:

   i. A transportation domain is selected randomly (among those that contain some undelivered cargo)

   ii. An undelivered cargo is selected in that component randomly

   iii. All appropriate meta-actions that are already known are iterated and if any of them is applicable in given state, then it is applied and this process is repeated from step 4(g)i (see 3.4.2). A heuristic function is used during the operator instantiation process to preffer certain actions. (see 3.5.3)

   iv. If no known meta-action is applicable then the learning procedure is called and new meta-actions are found and stored for future use (see 3.4.3). (This step is computationally demanding but it is only called once for every meta-action. After that the template is stored and used in future planning)

   v. If no meta-action can be learned then we have reached a dead-end and deal with the situation as described in 3.5.2.

(h) after reaching the end it translates the resulting plan to original actions (the plan we found by the simulation is composed of meta-actions), then it evaluates the translated plan. This number is used to evaluate the simulation.

(i) it propagates the result back in the tree, updating the statistics of the nodes.

5. after each iteration the pruning criterion is checked. The criterion is the total number of nodes in the tree. If it reaches a certain limit a hard pruning is performed (see 2.4) so that the planner wouldn't run out of memory.

6. these steps are repeated until some terminating criterion is met.

3.7 Generalization

We have shown the way how to modify a planning domain if it is of a transportation type. In this section we describe possible ways to generalize our approach to an arbitrary domain. We also describe our work in more general terms and we compare it to other techniques widely used in the field of both satisficing and optimal planning (developed by other authors).

Let us summarize the main problems with using the MCTS algorithm for planning:

- cycles in the state-space - they cause the expected length of the simulations to be extremely high (and therefore slow the algorithm)
- dead-ends in the state-space - they cause that the simulations cannot find a proper solution and therefore the sampling is ineffective (or even impossible)
- combination of both - dead-cycles (i.e. strongly connected components in the state-space that don't contain any goal state) - this is similar to the dead-ends problem except that we can easily detect a dead-end (since there are no applicable actions there) but it's much more difficult to detect a dead-component since we would have to store all the visited states (and search among them every time) which would make the simulation process much slower and the algorithm less effective.

Basicly what we want is to make the simulations short (i.e. low expected number of steps) and to ensure that most of the simulations will reach the goal (i.e. low probability of reaching a dead-end or a dead-component). We can achieve that either by preprocessing the domain or by guiding the simulation by some heuristic function towards the goal. In both ways we need to gain some information about the domain. There are to major ways to do that.

- Use the techniques based on satisficing planning
- Use the techniques based on formal verification

3.7.1 Techniques based on satisficing planning

This approach is a direct generalization of the method we used. It can provide either the hard-rules to prune the state-space or a soft-rules (i.e. suggestions) in a form of a domain-independent heuristic. The method is as follows:

1. identify the *important states* in the state space
2. at each *important state* find the way to the nearest next *important state* until reaching the end
3. paths between the important points should be safe in a sense that the simulation can't go astray to a dead-end.

In our case the *important states* were the points of loading and unloading the cargo and the paths between these points were represented by the meta-actions. We specified the important points by hand (based on the knowledge that the domain is of a transportation type). The paths between these points we learned during the planning process, stored them in a form of the meta actions, and use them in the future.

This principle is identical to the notion of Hierarchical task network planning (HTN planning). In HTN planning we achieve the goal task by dividing it into several simpler tasks and dealing with these tasks separately. In this case the planner has to be provided with a set of *HTN-tasks* (which tells it *what* should be achieved) and a set of *HTN-methods* which give the information *how* the task can be achieved. This information is domain-dependent and the HTN-based planners request it as an imput from the user.

In this thesis we have chosen a different way - we do not require this information as an imput but rather try to learn it during the planning process. This approach is less effective but more general since it's domain-independent and also cheaper since it doesn't require the assistance of a human expert.

This approach is called the HTN-learning and is currently intensively studied by many researchers. It can be divided to two categories:

- learning the HTN-tasks
- learning the HTN-methods

We can see that the *important points* mentioned above are a special case of HTN-tasks - they describe where we need to go in the searched-space and the *meta-actions* are a special case of HTN-methods - they describe the way from current state to some given state (to a state where the task is accomplished). The main difference between meta-actions how we defined them (3.2.1) and HTN-methods is that the meta-action can only be composed of *original* actions while the HTN-method can decompose the task to other *compound* tasks. We could achieve this if we allowed the meta-action to be composed of other meta-actions however that would make the learning process much more difficult therefore we only use this special case.

In our case we specified the *tasks* manually with the knowledge of the type of a domain (the tasks corresponds to the *important points* mentioned above). Therefore the task-learning phase was reduced to recognizing some predefined patterns in the domain structure. The method-learning phase was in our case implemented using an exhaustive search during the planning process where we assumed that the domain is simple enough for this process to be tractable - i.e. that the paths between the important points are short and simple. (the HTN-methods corresponds to the meta-actions).

In this respect our work is quite unique since most of the papers presented on this topic are only concerned with one of these phases (either task learning or method learning) and the vast majority use methods of *supervised learning* which requires training data to be provided by the user. Our planner combines both phases and uses *unsupervised learning* i.e. it doesn't require any additional information about the domain besides the standard description.

Another important notion related to this topic is the notion of *landmarks*. A landmark represent some area of the searched-space which every possible solution plan has to visit - i.e. a *cut* between the initial state and the set of all goal states. The landmarks are intensively studied in the field of satisficing planning since they can be very helpful when solving the problem. There are several kind of landmarks, we give the refferences in the section Related work at the end of this thesis. The work on landmark can be divided to two categories - methods for *finding* the landmarks and methods for *using* them during the planning. In our case the landmarks can be used as the *important points* in the state-space. When all the solution plan has to pass the landmark then it makes sense to decompose the planning task to two phases - finding the paths from the initial state to the landmark and then from the landmark to some goal state. (They can help us to distinguish the *important paths* that we want to preserve from those we want to eliminate - see 3.2.4.)

To sum up the previous information: our technique can be generalized to arbitrary domain by the use of HTNs and landmarks. The meta-actions can be obtained by HTN-method-learning techniques and the *important points* can be found by HTN-task-learning, possibly by using landmarks. The landmarks can also provide strong domain-independent

heuristics to guide the simulations to avoid dead-ends. [31] [32] [33] Implementing this approach for arbitrary domain however doesn't seem possible (with the current state of the research), at least not in the form we did for the transportation component. Reasons for this are as follow:

- current HTN-learning techniques rely on *supervised learning* and requires the training data
- HTN-planning still uses backtracking - in fact there are even more backtracking points in HTN-planning since it searches for the correct instantiation of the operators as well as for the correct decomposition of the tasks i.e. there are two independent searches involved, but the simulation in MCTS doesn't backtrack so it may cause a rapid increase of dead-end states.
- strong landmark-based heuristics takes long time to compute. They are applicable when we search for *one* plan and we want to guarantee the optimality (and have a lot of time) however they are unfit to be used in the simulation phase of MCTS where we need to perform dozens of successful simulation every second.

A universal MCTS-based planner would require a strong and fast technique for solving the underlying satisficing problem which doesn't seem possible for arbitrary planning domain. It can however be used in the domains where the underlying problem is easy and optimality is the real task. A further research in this field is required, we give some ideas at the end of this thesis.

3.7.2 Techniques based on formal verification

Formal verification studies methods that can be used to prove that some system (typically software system) has some property. The typical topics are proving that the system never reaches some state or that it always meets some specified condition. Practical examples would be proving that the program is dead-lock-free or that some given method always returns a nonnegative number (for all possible inputs).

The methods used in this field are based mostly on logic and automated theorem proving. They can be often interpreted as a manipulation with the state-space of the system. State-spaces of programs are typically infinite therefore abstraction techniques are used to make the search possible.

The purpose of an abstraction is to identify only those features of the states that we actually need to resolve the given question. The other features are "abstracted away" - i.e. we consider states that only differ in non-important features to be equivalent. This technique can greatly reduce the search-space if we can identify the essential features correctly.

Many tool for verification have been developed so far - predicate abstraction and its generalizations, Kripke Structures, Computation tree logic, transition invariants, well-founded relations-based methods and other. Some of them can be used in the field of planning as a domain-analysis techniques.

One of the most important tasks in the field of verification is the program termination problem (i.e. - to decide whether given program will terminate). Even if this question is know to be undecidable in general, many techniques have been devised that work in specific cases. These techniques are based on the detection of cycles in an abstract state-spaces of the problem. [26] [27].

An extensive reasearch si also dedicated to the problem of proving the correctness of a program. In this case the problem is to decide whether the program will visit some undesirable state and identify the imput that leads to that state. (For example when we put an *assert* statement checking whether some property holds and we want to identify the imput that triggers the assert or prove that none such imput exists.)

Since a planning domain can also be seen as a state-space with transitions, the verification techniques could be used here as well. In our case we would use them to detect cycles and dead-ends in the state-space. (Termination analysis techniques can be used to detect the cycles and correctness proving can be used to prove that the planning process can never reach a dead-end.)

This process would most likely be time-consuming since deciding such properties in general is intractable. (However several verification tools have already been developed and they work in reasonable time - at least for some inputs.) Another question is how to use the results of the analysis - if we found that there is a cycle or a dead-end, how would we use this information to eliminate it. (Note that the verification techniques don't deal with this question - if they find a bug in the program they don't repair it. That is still left to the user.)

We have tried to adapt some verification techniques (namely the abstraction refinement [28]) for the task of cycles and dead-ends elimination during writing this thesis (as an alternative way to preprocess the domain in order to use MCTS). The technique showed some promise however we haven't been able to create a working and efficient algorithm (so we chose a different approach using the meta-actions as described in this thesis). Since the field of automated verification is currently in the spotlight of many researchers, we can expect new methods to be devised that could potentially be used for planning.

4. Experiments

In this chapter we present results of the experiments we conduted. We compared our planner with the *LPG-td* planner [34],[35], a domain-independent planner for PDDL2.2 domains, the top performer at IPC4 in plan quality. This is a short description of the planner provided by its authors:

LPG (Local search for Planning Graphs) is a planner based on local search and planning graphs that handles PDDL2.1 domains involving numerical quantities and durations. The system can solve both plan generation and plan adaptation problems. The basic search scheme of LPG was inspired by Walksat, an efficient procedure to solve SAT-problems. The search space of LPG consists of "action graphs", particular subgraphs of the planning graph representing partial plans. The search steps are certain graph modifications transforming an action graph into another one. LPG exploits a compact representation of the planning graph to define the search neighborhood and to evaluate its elements using a parametrized function, where the parameters weight different types of inconsistencies in the current partial plan, and are dynamically evaluated during search using discrete Lagrange multipliers. The evaluation function uses some heuristics to estimate the "search cost" and the "execution cost" of achieving a (possibly numeric) precondition. Action durations and numerical quantities (e.g., fuel consumption) are represented in the actions graphs, and are modeled in the evaluation function. In temporal domains, actions are ordered using a "precedence graph" that is maintained during search, and that takes into account the mutex relations of the planning graph. The system can produce good quality plans in terms of one or more criteria. This is achieved by an anytime process producing a sequence of plans, each of which is an improvement of the previous ones in terms of its quality. LPG is integrated with a best-first algorithm similar to the one used by FF. The system can automatically switch to best-first search after a certain number of search steps and "restarts" have been performed. Finally, LPG can be used as a preprocessor to produce a quasi-solution that is then repaired by ADJ, a plan-analysis technique for fast plan-adaptation.

4.1 Problems used for testing

We conduct the experiments in the Zeno-Travel domain (A.1) since it is a typical example of a transportation-based domains. This domain may seem fairly simple however it is expressive enought to describe the Travelling salesman problem (TSP) and therefore solving it is NP-Complete.

We have generated 40 problems of different size. The smallest problem contained 5 persons, 1 airplane, and 3 locations, the largest one contained 45 persons, 9 planes and 16 locations. The size of the problems is shown in the figure 4.1. We run each planner for a

given amount of time. We gave the planners shorter amount of time to solve the smaller problems and longer amount for the bigger problems. The time limit for every problem is $locations * 2$ in minutes (that is about 5 minutes for the smallest problems and about 30 minutes for the largest).

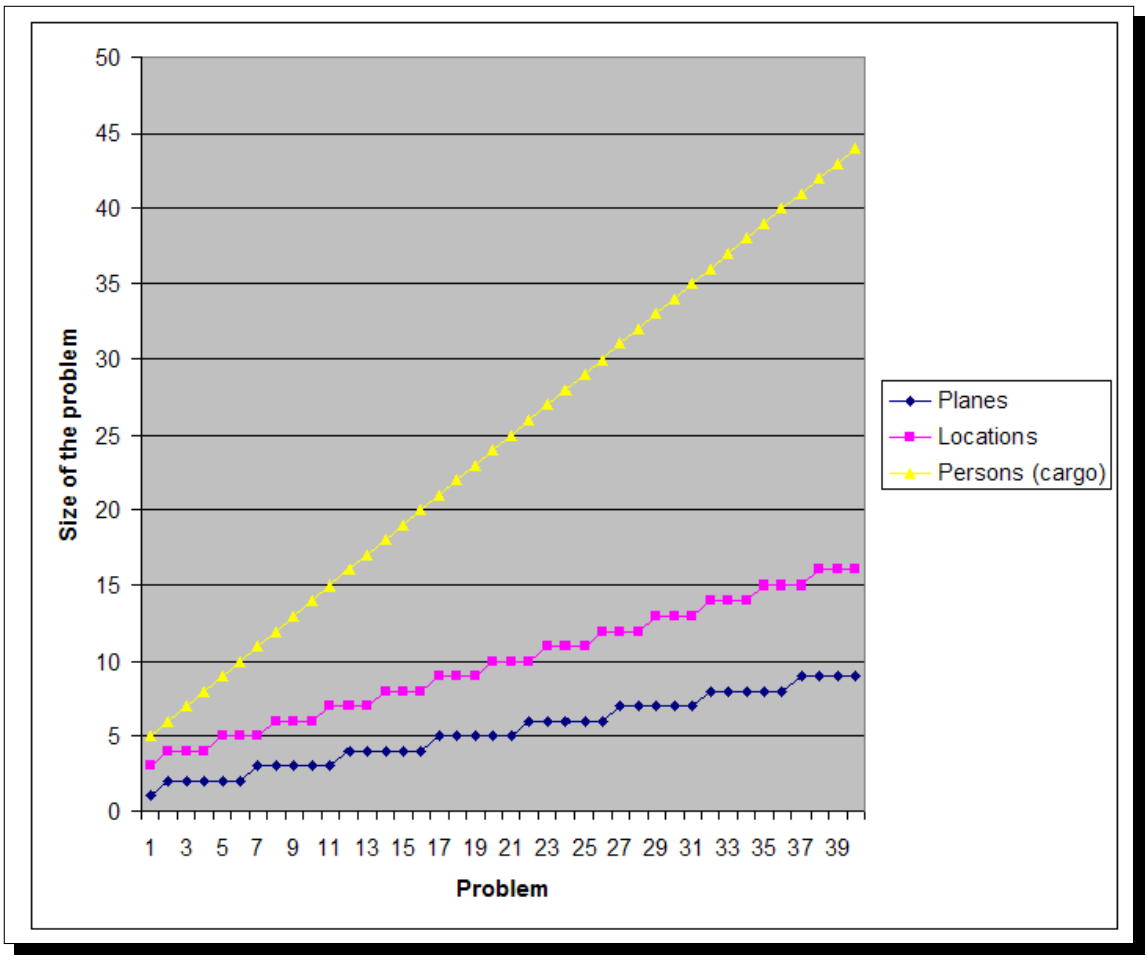


Figure 4.1: Size of the problems used

4.2 Results

We used three versions of our planner with different values of parameters. Our planners minimized the sum of fuel usage and makespan while the LPG planner minimized the fuel only (since it cannot schedule and therefore it cannot compute the makespan)

Results are given in the graphs 4.2, 4.3, 4.4, 4.4, 4.6,

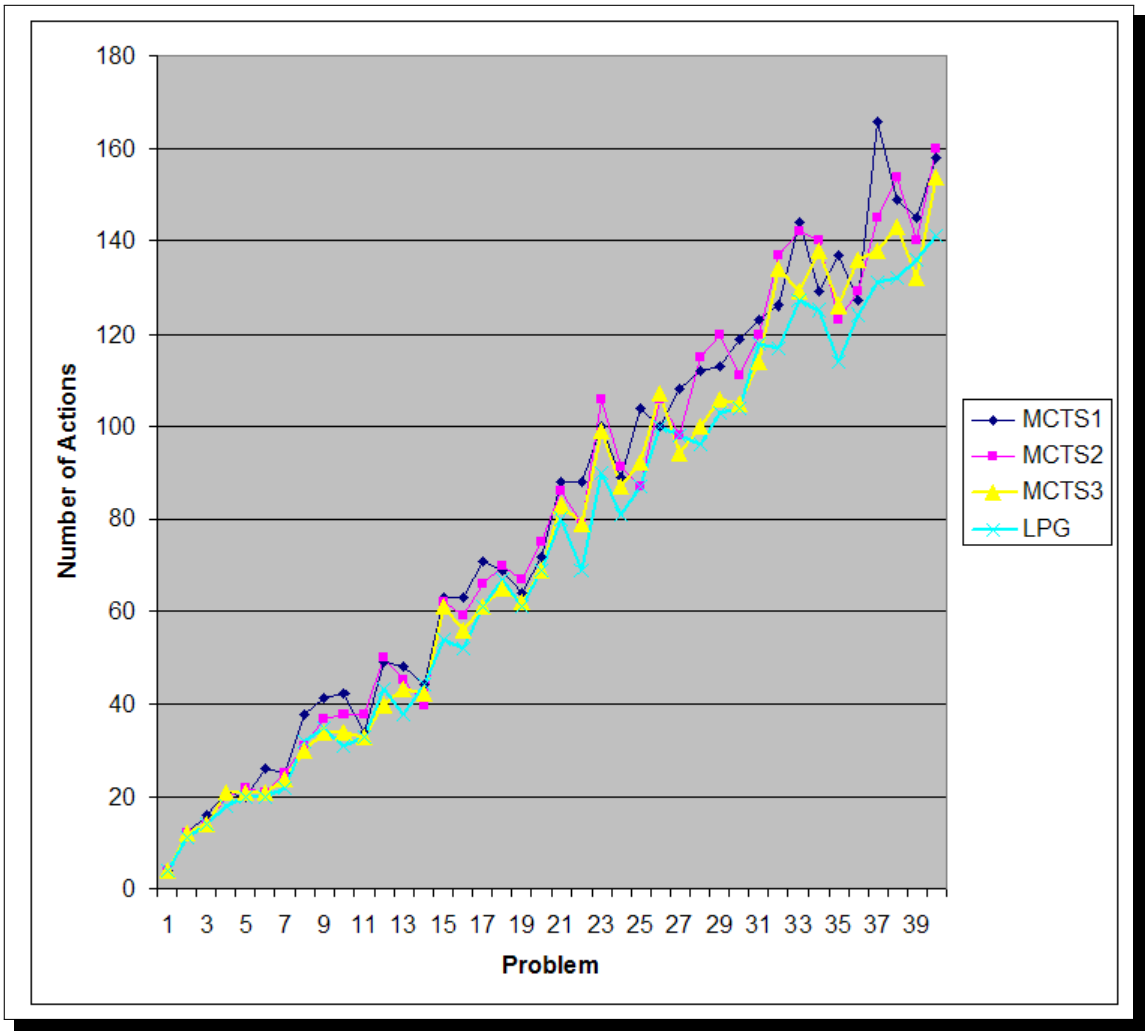


Figure 4.2: Number of actions of found plans

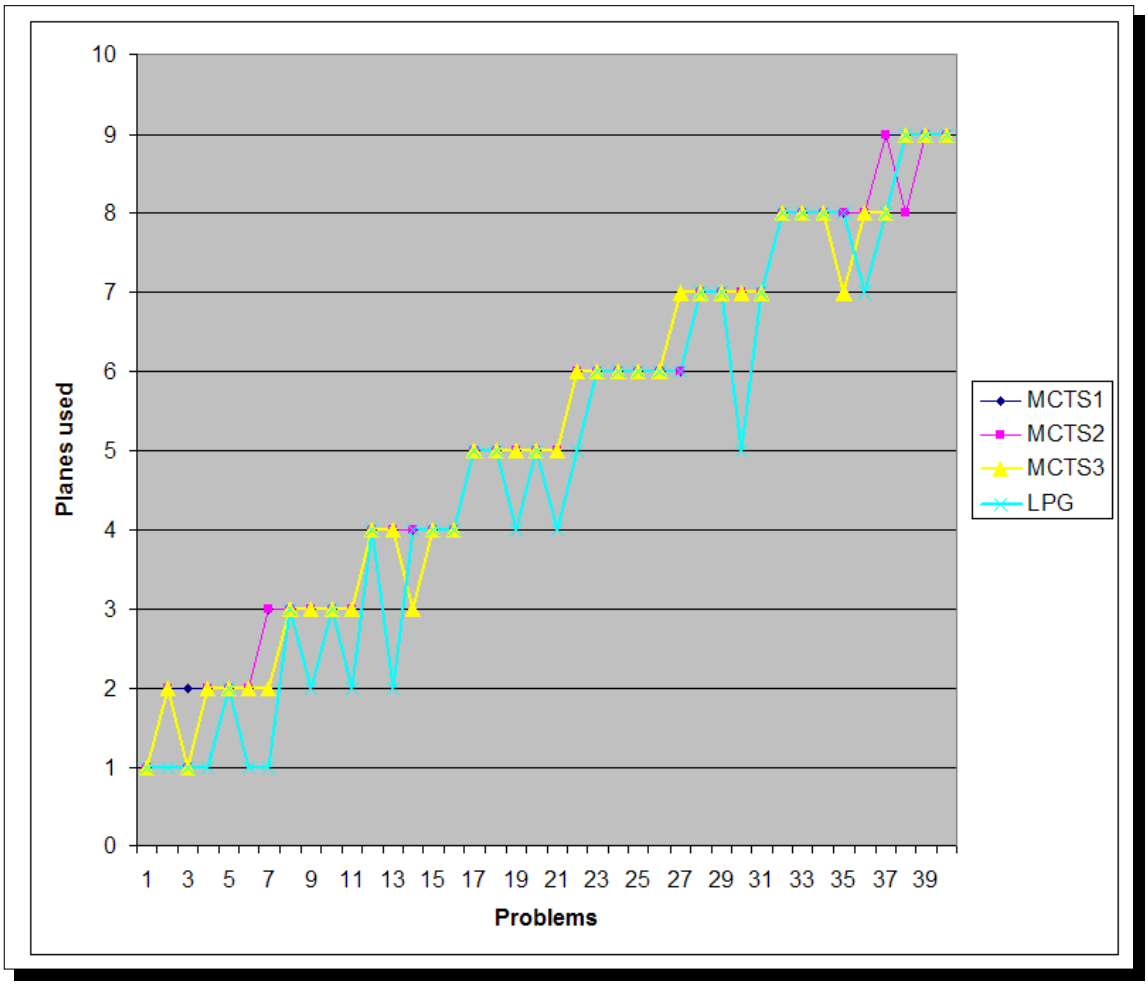


Figure 4.3: Number of planes used

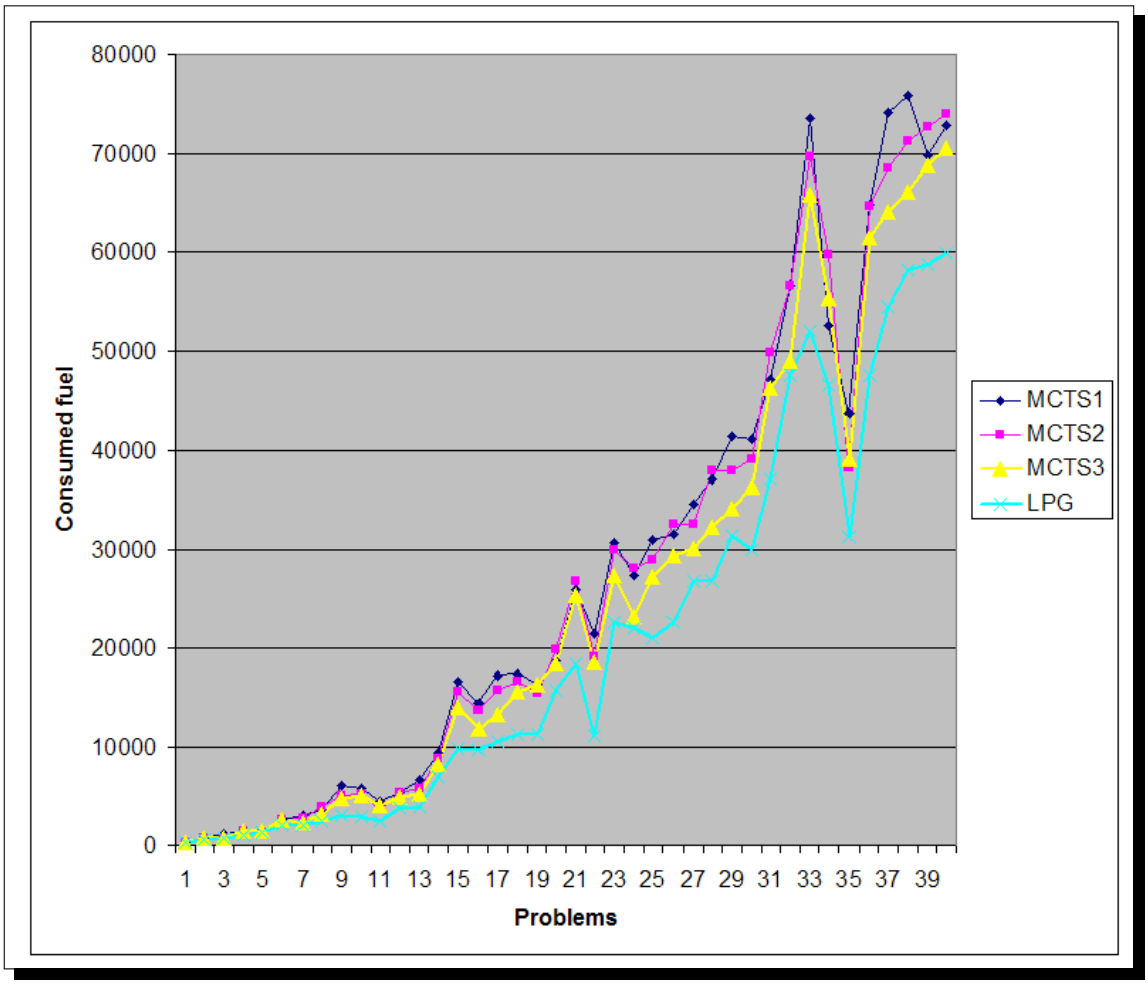


Figure 4.4: Fuel consumption

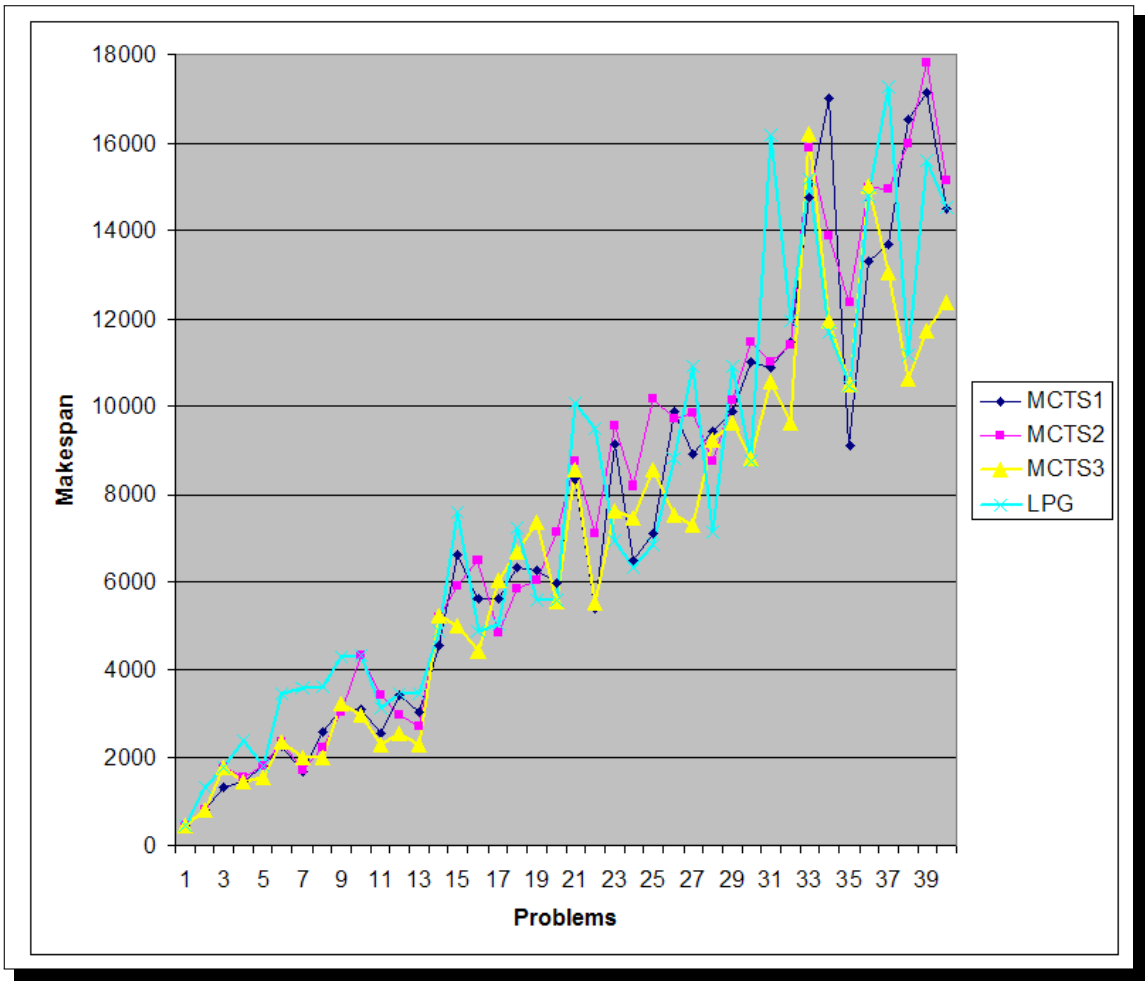


Figure 4.5: Makespan

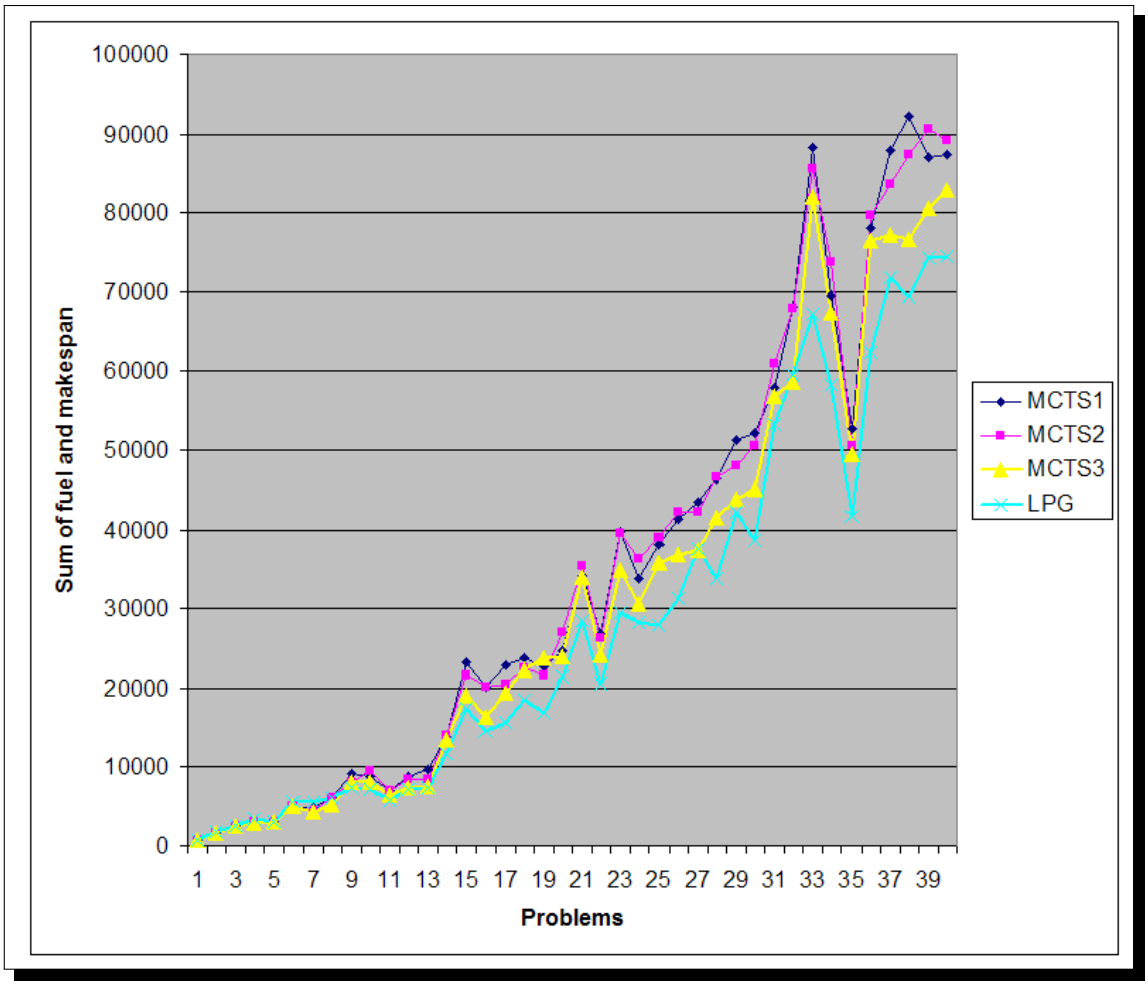


Figure 4.6: Sum of fuel and makespan

4.3 Discussion

The results show that LPG planner outperforms our planner in most cases, especially in fuel comsumption (which it minimized) but also in the sum of both component. Our planners showed better results in makespan (since the LPG didn't take it into acount).

Also we have found out that the values of parameters that the planner uses are not as much important as we thought. There are some differences in the performance between MCTS1, 2, and 3. In the MCTS3 we used all the techniques described in the thesis, in MCTS2 we disabled the symmetry-breaking and in MCTS1 we further used a different strategy for the widening. The results show that the symmetry breaking as well as the correct widening approach can improve the performance.

The LPG shows better overall results however the difference between its performance and the performance of our planner is not big. We believe that we could further increase the performance (some ideas are given in the section Future work) and that the MCTS method has a potential to compete with state-of-the-art planners.

Conclusion

In this thesis we studied the possible ways of using the MCTS algorithm in the field of planning and scheduling. We analyzed the problem and identified the important features of MCTS approach that would cause difficulties when applied in the field of planning and we developed methods to overcome these difficulties and to make the algorithm and the domain compatible. We achieved that by modifying both the algorithm and the planning problem.

To modify the algorithm we used several techniques previously published by other authors as well as our own ideas and we developed the way to combine these modifications together to work efficiently and to make the algorithm suited for the task of planning.

We also developed a technique for preprocessing the planning domain to make the algorithm more efficient. Our method is based on an unsupervised learning of promissing combinations of actions that can be seen as a simplified version of HTN-methods-learning. We described the technique and provided both theoretical and experimental results about its contribution to the overall performance. We also used several domain-independent techniques adopted from both optimal and satisficing planning.

During the analysis we found out that MCTS algorithm is more suited for some kinds of planning problems and less suited for the others. Our method can theoretically be applied to arbitrary planning domain but only on certain types of domains will it be efficient. One of these types are domains based on transportation. We decided to implement the technique for this kind of domains only to ensure that the planner is usable in practice.

We have implemented the designed solution and performed experiments to compare the planner with other available planning software. The experiments showed that the planner is competitive with state-of-the-art planning software (on the specific kind of domains). It can achieve comparable quality of the plans but requires more time and memory then the other planner we tried (*LPG-td* planner - a top performer at IPC4 in plan quality).

In this thesis we discovered that using MCTS for the task of optimal planning requires an efficient technique for solving the underlying satisficing problem therefore the algorithm is best suited for problems where the satisficing part is easy and the optimality is the real issue. Transportation domains are a good example. We believe that there are many more planning domains that have this property (especially those that are practically motivated) and therefore MCTS-based techniques may become a new and efficient way to address such problems.

Related work

In this part we mention techniques developed by other authors that are related to our work.

- near-optimal substructures - in section 1.5.3 we have stated that Genetic algorithms implicitly use *near-optimal substructures* as their building blocks which MCTS cannot do even though it would increase the performance. Later in section 3.2.3 we noted that the meta-action we use can partially play the role of the building blocks. Several other methods have been developed to deal with this drawback of MCTS:
  - Last Good Replay heuristic [36] - idea: we store the last good response to every move and preffer it in the simulation phase. The responses are gained (learned) from the successful simulations. (The pair move-response forms a near-optimal substructure that is used later.) This idea is very similar to history heuristic [16] used in chess.
  - Patterns [37] - idea: we store some predefined patterns - i.e. positions that are known to be either favorable or unfavorable for the player and use them during the selection or simulation phase (to achieve them or avoid them).
- symmetry breaking mentioned in section 3.5.4 - other similar problems:
  - MCTS based symmetries: transpositions [38] MCTS uses a tree but most problems can be better modeled as a DAG (directed acyclic graph). That creates identical subtrees in the MCTS tree. Several techniques have been devised to deal with this problem (mentioned in the cited paper).
  - planning based symmetries: several objects in the planning problem that are absolutely identical (example: two identical vehicles at the same location. when we search for the plan we there will be two identical trees - one for each vehicle). In this case the vehicles should be used more like a resource (i.e. we don't care what vehicle we use, we only keep track of how many of them are use and how many are left)
- preprocessing the domain to increase performance. The techniques we use to do this is generally called *domain analysis*. More about the topic can be found in [18] in the chapter 24.6
- HTN-learning - the method we used to find the meta-actions is a simple form of a HTN-learning. This topic is currently in a spotlight of many researchers and many paper have been published ([39],[40],[41] to mention at least some). There are several approaches in this field - some papers describe the ways how to learn the methods from vaguely described plans (i.e. when we don't even know what are the possible actions) others are closer to our technique however unlike us the vast majority use some kind of supervised learning - they require the user to provide a set of problems

in the domain together with solution plans for this problems. Also many published techniques tries to learn the HTNs in some special format so that the learning would be simpler. (We do this in our work as well.)

- landmarks - the technique we used to find the goals we want to achieve (i.e. to load the cargo somehow and the to unload it somehow) is a very simple example of a landmark generation [42]. A *state landmark* is a set of states that every solution plan has to visit. (For example every cargo has to be loaded and unloaded in order to be delivered). In our work we used a predefined template to describe how the landmark should look like and then automated technique to find it in the domain. This is a very simple example, there exists much more complicated procedures that can find the landmarks automatically in any domain [42].

- verification-based techniques - in 3.7.2 we briefly described some techniques of automated verification that could be used in planning. Some techniques have already been used in this way - mostly in the field of *planning with partial information* (see [18] chapter 17 - Planning Based on Model Checking).

- another possibility how to guide the simulations in MCTS is by *Control rules*. Control rules are an alternative to HTNs, in some domain the Control rules work better, in other domains the HTNs are better. (see [18] chapter 10 - Control Rules in Planning.

- other approaches to use MCTS in planning. Some work has already been done in this field [43],[44],[45]. However most of available work use quite different approach then we do in this thesis. They often make so drastic changes to the MCTS algorithm that the approach can no longer be considered Monte-Carlo-based. (For example replacing the random sampling with a deterministic evaluation function.)

Future work

We mention several ideas how to further improve the planner:

- using CSP during the operator instantiation
- using statistical testing during the pruning
- use the Last Good Replay heuristic and Weighting simulation results [14]
- improve the learning phase by using better algorithm for path-finding (i.e. satisficing planning)
- use Automated Parameter Tuning techniques to adjust the parameters.
- use restarts during the planning
- use some hybrid algorithm that would allow backtracking in the simulation phase

Bibliography

[1] Schadd, M. P. D., Winands, M. H. M., van den Herik, H. J., Aldewereld, H. *Addressing NP-Complete Puzzles with Monte-Carlo Methods* in *AISB 2008 Convention: Communication, Interaction and Social Intelligence*, 2008.

[2] Gelly, S., Schoenauer, M., Sebag, M., Teytaud, O., *The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions* in *Communications of the ACM* vol. 55, no. 3, pp. 106–113, 2012.

[3] Toropila, D., Dvořák, F., Trunda, O., Hanes, M., Barták, R., *Three Approaches to Solve the Petrobras Challenge: Exploiting Planning Techniques for Solving Real-Life Logistics Problems* in *Proceedings of 24th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2012)*, pp. 191–198, IEEE Conference Publishing Services, 2012.

[4] Hanes, M, *Petrobras Planning Domain: PDDL Modeling and Solving.* Bachelor thesis, Department of Theoretical Computer Science and Mathematical Logic at the Faculty of Mathematics and Physics, Charles University in Prague, 2012.

[5] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S. *A Survey of Monte Carlo Tree Search Methods* in *IEEE Transactions on Computational Intelligence and AI in Games,*, vol.4, no.1, 2012.

[6] Kroese, D. P., Taimre, T., Botev, Z. I. *Handbook for Monte Carlo methods.* Hoboken, N.J.: Wiley, 2011. ISBN 978-0-470-17793-8.

[7] Chaslot, G., Bakkes, S., Szita, I., Spronck, P. *Monte-Carlo Tree Search: A New Framework for Game AI* in *Proceedings of the 4th Artificial Intelligence for Interactive Digital Entertainment conference (AIIDE)*, 2008

[8] Auer, P., Cesa-Bianchi, N., Fischer, P. *Finite-time Analysis of the Multiarmed Bandit Problem* in *Machine Learning* vol. 47, no. 2-3, pp. 235–256, 2002.

[9] Kocsis, L., Szepesvári, C. *Bandit Based Monte-Carlo Planning* in *15th European Conference on Machine Learning (ECML)* pp. 235–256, 2006.

[10] Gelly, S., Silver, D. *Combining online and offline knowledge in UCT* in *Proceedings of the International Conference of Machine Learning (ICML)* pp. 273-280, 2007.

[11] Baudiš, Petr, *Balancing MCTS by Dynamically Adjusting Komi Value* in *ICGA Journal* vol. 34, pp. 131–139, 2011.

[12] Schadd, M. P. D., Winands, M. H. M., van den Herik, H. J., Chaslot, G. M. J.-B., Uiterwijk, J. W. H. M. *Single-player Monte-Carlo tree search* in *Computers and Games* vol. 5131, of *Lecture Notes in Computer Science*, pp 1–12, 2008.

[13] Björnsson, Y., Finnsson, H. *CadiaPlayer: A simulation-based general game player* in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, pp. 4–15, 2009.

[14] Xie, F., Liu, Z. *Backpropagation modification in Monte-Carlo game tree search* in *The Third International Symposium on Intelligent Information Technology Application (IITA)*, vol.2, pp. 125–128, 2009.

[15] Goldberg, D. E. *Genetic algorithms in search, optimization, and machine learning.* Reading: Addison-Wesley, 1989. ISBN 0-201-15767-5.

[16] Schaeffer, J. *The history heuristic and alpha-beta search enhancements in practice* in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.11, no.11, pp. 1203–1212, 1989.

[17] Kubalik, J., Faigl, J. *Iterative Prototype Optimisation with Evolved Improvement Steps* in *Proceedings of the 9th European Conference on Genetic Programming (EuroGP)*, pp. 154–165, 2006.

[18] Ghallab, M., Nau, D. S., Traverso, P. *Automated planning: theory and practice.* Amsterdam: Elsevier/Morgan Kaufmann, 2004. ISBN 1-55860-856-7.

[19] Cazenave, T. *Nested Monte-Carlo Search* in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 456–461, 2009.

[20] Malý, D. *Solving problems using MCTS.* Master thesis, Department of Theoretical Computer Science and Mathematical Logic at the Faculty of Mathematics and Physics, Charles University in Prague, 2010.

[21] Hearn, R. A. *Games, Puzzles, and Computation.* Ph.D. thesis, Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, 2006.

[22] Ramanujan, R., Sabharwal, A., Selman, B. *On adversarial search spaces and sampling-based planning* in *Proceedings of 20th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 242–245, 2010.

[23] Erol, K., Nau, D., Subrahmanian, V. S. *Complexity, decidability and undecidability results for domain-independent planning* in *Artificial Intelligence*, vol.76, no.1,2, pp. 75–88, 1995.

[24] Erol, K., Hendler, J., Nau, D. *Complexity results for hierarchical task-network planning* in *Annals of Mathematics and Artificial Intelligence*, pp. 69–93, 1996.

[25] Bandelt, H.-J., Chepoi, V. *Metric graph theory and geometry: a survey* in *Surveys on Discrete and Computational Geometry, Contemporary Mathematics, AMS, Providence, RI*, vol. 453, pp. 49-86, 2008.

[26] Tsitovich, A., Sharygina, N., Wintersteiger, C. M., Kroening, D. *Loop summarization and termination analysis* in *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems*, 2011.

[27] Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C. M. *Loop summarization using abstract transformers* in *Automated Technology for Verification and Analysis, Lecture Notes in Computer Science* vol.5311, pp. 111-125. Springer, 2008.

[28] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., Veith, H. *Counterexample-guided abstraction refinement for symbolic model-checking* in *Journal of Association for Computing Machinery* vol.50, no.5, pp. 752-794, 2003.

[29] Lenz, A., Lallee, S., Skachek, S., Pipe, A. G., Melhuish, C., Dominey, P. F. *When shared plans go wrong: From atomic- to composite actions and back* in *International Conference on Intelligent Robots and Systems (IROS)* IEEE/RSJ, pp. 4321-4326, 2012.

[30] Milicic, M. *Planning in Action Formalisms based on DLS: First Results* in *Proceedings of the Intl Workshop on Description Logics*, 2007.

[31] Bonet, B., Helmert, M. *Strengthening landmark heuristics via hitting sets* in *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*, pp. 329-334, 2010.

[32] Haslum, P., Geffner, H. *Admissible heuristics for optimal planning* in *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pp. 140-149, 2000.

[33] Helmert, M., Domshlak, C. *Landmarks, criticalpaths and abstractions: What's the difference anyway?* in *Proceedings of the Nineteenth International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pp. 162-169, 2009.

[34] Gerevini, A., Serina, I. *LPG: a Planner based on Local Search for Planning Graphs* in *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

[35] Gerevini, A., Saetti, A., Serina, I. *Planning in PDDL2.2 Domains with LPG-TD* in *International Planning Competition, 14th International Conference on Automated Planning and Scheduling (IPC/ICAPS)*, 2004.

[36] Drake, P. D. *The Last-Good-Reply Policy for Monte-Carlo Go* in *International Computer Games Association Journal*, vol.32, no.4, pp. 221-227, 2009.

[37] Drake, P. D., Uurtamo, S. *Heuristics in Monte Carlo Go* in *Proceedings of the International Conference on Artificial Intelligence*, pp. 171-175, 2007.

[38] Childs, B. E., Brodeur, J. H., Kocsis, L. *Transpositions and Move Groups in Monte Carlo Tree Search* in *Proceedings of IEEE Symposium on Computational Intelligence and Games* , pp. 389-395, 2008

[39] Hogg, C. M., Munoz-Avila, H., Kuter, U. *HTN-maker: Learning HTNs with minimal additional knowledge engineering required* in *Proceedings of Association for the Advancement of Artificial Intelligence*, pp. 950-956, 2008

[40] Li, N., Kambhampati, S., Yoon, S. *Learning probabilistic hierarchical task networks to capture user preferences* in *Proceedings of the 21th International Joint Conference on Artificial Intelligence*, 2009

[41] Hogg, C. M., Kuter, U., Munoz-Avila, H. *Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning* in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010

[42] Bonet, B, Castillo, J. *A complete algorithm for generating landmarks* in *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*, 2011

[43] Xie, F, Nakhost, H., Muller, M. *A Local Monte Carlo Tree Search Approach in Deterministic Planning* in *Proceedings of Association for the Advancement of Artificial Intelligence*, pp. 1832-1833, 2011

[44] Pellier, D, Bouzy, B., Metivier, M. *An UCT Approach for Anytime Agent-Based Planning* in *Proceedings of International Conference on Principles and Practice of Multi-Agent Systems*, pp. 211-220, 2010

[45] Walsh, T.J., Goschin, S., Littman, M. L. *Integrating Sample-based Planning and Model-based Reinforcement Learning* in *Proceedings of Association for the Advancement of Artificial Intelligence*, pp. 612-617, 2010

List of Abbreviations

AMAF - All Moves As First
BFS - Breadth-first search
CSP - Constraint satisfaction problem
DAG - Directed acyclic graph
GA - Genetic algorithms
GGP - General game playing
HTN - Hierarchical task network
MCTS - Monte Carlo Tree Search
RAVE - Rapid Action Value Estimation
SP-MCTS - Single-Player Monte Carlo Tree Search
TSP - Travelling salesman problem
UCT - Upper Confidence Bounds for Trees

A. Planning domains description

Here we provide description of the planning domains that we reffer to in this thesis.

A.1 Zeno-Travel Domain - simplified

Characterization

We use a simplified version of the Zeno-Travel Domain, firstly introduced at The Third International Planning Competition in 2002 as a basic example of transportation-based domain. The domain can describe problems of transporting passengers between airports. Passangers can be embarked and disembarked onto airplane which can travel between any two locations.

Loading a passanger requires constant amount of time (that can be specified in the desription file), traveling between locations requires a time proportional to the distance between locations and to the speed of the plane Speed can be specified in the description file and is the same for all the planes. Traveling consumes the amount of fuel proportional to the distance between locations and to "'Fuel consumption rate"' which can be specified in the description file.

No sources limitations are present in this domain - the plane can load arbitrary number of passengers, and an arbitrary number of planes can be present at any location in the same time. Moreover the fuel is not taken into account when deciding applicability of the actions i.e. the plane can always travel between any two locations no matter of their distance. (This can be seen as that the plane has a fuel tank large enought to travel between any two locations and at every location the fuel tank is fully replenished.)

Goal is to deliver all passengers to their target locations and minimize some linear combination of fuel and time (makespan) required.

Formal description (PDDL)

```
(define (domain zeno)

(:requirements :typing)

(:types person airplane - thing
                location)

(:predicates (at ?obj - thing ?l - location)
```

```
            (in ?p - person ?veh - airplane))

(:action fly
         :parameters    (?a - airplane ?from ?to - location)
         :precondition  (at ?a ?from)
         :effect        (and (not (at ?a ?from))
                             (at ?a ?to)))

(:action load
         :parameters    (?a - airplane ?p - person ?l - location)
         :precondition  (and (at ?a ?l)
                             (at ?p ?l))
         :effect        (and (not (at ?p ?l))
                             (in ?p ?a)))

(:action unload
         :parameters    (?a - airplane ?p - person ?l - location)
         :precondition  (and (at ?a ?l)
                             (in ?p ?a))
         :effect        (and (not (in ?p ?a))
                             (at ?p ?l)))
)
```

Problem example

Following PDDL code describes a Zeno-Travel problem with two passengers, three locations and one airplane.

```
(define (problem zeno-1_3_2))

        (:domain zeno)

        (:objects plane1 - airplane
                  person1 person2 - person
                  loc1 loc2 loc3 - location)

        (:init (at plane1 loc3)
               (at person1 loc1)
               (at person2 loc1)
        )

        (:goal  (and (at person1 loc2)
                     (at person2 loc3))
        )
```

)

A.2 Zeno-Travel domain - original

Characterization

Previously descripted domain was a simplification of the original in many respects. Originally introduced domain allowed the airplanes to travel at two speeds - slow and fast, where with the higher speed the airplane consumes more fuel. Domain descripted here differs from the previous one by allowing one more action for traveling at higher speed.

Formal description (PDDL)

```
(define (domain zeno)

(:requirements :typing)

(:types  person airplane - thing
                location)

(:predicates  (at ?obj - thing ?l - location)
              (in ?p - person ?veh - airplane))

(:action fly
         :parameters     (?a - airplane ?from ?to - location)
         :precondition   (at ?a ?from)
         :effect         (and (not (at ?a ?from))
                              (at ?a ?to)))

(:action flyFast
         :parameters     (?a - airplane ?from ?to - location)
         :precondition   (at ?a ?from)
         :effect         (and (not (at ?a ?from))
                              (at ?a ?to)))

(:action load
         :parameters     (?a - airplane ?p - person ?l - location)
         :precondition   (and (at ?a ?l)
                              (at ?p ?l))
         :effect         (and (not (at ?p ?l))
                              (in ?p ?a)))
```

```
(:action unload
         :parameters     (?a - airplane ?p - person ?l - location)
         :precondition   (and (at ?a ?l)
                              (in ?p ?a))
         :effect         (and (not (in ?p ?a))
                              (at ?p ?l)))
)
```

Problem example

All the problems for simplified domain can be used as an input for this domain as well. And all results found in the simplified domain are valid results in this domain. Allowing one additional action may lead to finding a better solution.

A.3 Logistics domain

Characterization

Logistics is a classical transportation domain used at The Fourth International Conference on Artificial Intelligence Planning Systems in 1998. The domain can describe problems where the task is to deliver cargo between locations using two kinds of vehicles - truks of planes.

Formal description (PDDL)

```
; Logistics domain used in (Nguyen & Kambhampati 2001).

(define (domain logistics-strips)
        (:requirements :equality)
        (:predicates (obj ?o)
                (truck ?t)
                (airplane ?p)
                (location ?l)
                (airport ?a)
                (city ?c)
                (at ?o ?l)
                (in ?o ?v)
                (in-city ?o ?c))

        (:action load-truck
                :parameters (?o ?truck ?loc)
                :precondition (and (obj ?o)
                        (truck ?truck)
```

```
                (at ?o ?loc)
                (at ?truck ?loc))
        :effect (and (not (at ?o ?loc))
                (in ?o ?truck))
)

(:action load-plane
        :parameters (?o ?p ?loc)
        :precondition (and (obj ?o)
                (airplane ?p)
                (at ?o ?loc)
                (at ?p ?loc))
        :effect (and (not (at ?o ?loc))
                (in ?o ?p))
)

(:action unload
        :parameters (?o ?v ?loc)
        :precondition (and (in ?o ?v)
                (at ?v ?loc))
        :effect (and (at ?o ?loc)
                (not (in ?o ?v)))
)

(:action fly
        :parameters (?p ?s ?d)
        :precondition (and (airplane ?p)
                (airport ?s)
                (airport ?d)
                (at ?p ?s)
                (not (= ?s ?d)))
        :effect (and (at ?p ?d)
                        (not (at ?p ?s)))
)

(:action drive
        :parameters (?truck ?s ?d ?city)
        :precondition (and (truck ?truck)
                (at ?truck ?s)
                (in-city ?s ?city)
                (in-city ?d ?city)
                (not (= ?s ?d)))
        :effect (and (at ?truck ?d)
```

```
                (not (at ?truck ?s)))
        )
)
```

A.4 Petrobras domain

Characterization

This domain was proposed at The International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS) conducted under *The 22nd International Conference on Automated Planning and Scheduling (ICAPS)* in 2012 only in the form of verbal description. The task was to create a matematical model and use it to solve problems in this domain.

The domain can describe problems of transporting cargo between port and platforms over the sea. The actions have in this case more complicated preconditions that in the previous domains. Before the ship can load anything it has to *dock* in the station. Only a limited number of ships can be docked simultaneously. There is also limit on maximum cargo that the ship can load. Another difference between Petrobras and other domain described here is that the load and unloading operations take much longer that moving operation i.e. the ship can navigate to some destination within several hours while loading in the ports may take several day. The goal is to deliver all the cargo and minimize some combination of consumed fuel, makespan, docking time and several other vaguely described criteria (like that the ships should travel with as little fuel in the tanks as possible and so on).

Petrobras domain was the first where we tried the MCTS approach, we have developed a domain-dependent planner to solve it without even creating a PDDL representation (our planner took the inputs in a different format.) The good results our planner has achieved motivated us to continue the research of MCTS techniques in planning which we did in this thesis.

Formal description (PDDL)

The formal description wasn't provided at the conference however Martin Hanes created the PDDL representation. It can be found in his thesis. [4]

Problem example

Problems can be found in the our previous work on this topic. [3]

B. Content of the attached DVD

- directory Planner - all the source codes of our planner, as well as the problems generator
- directory Executable - the executable file of our planner along with short user manual
- directory Experiments - the experiments we conducted together with the results
- thesis.pdf - the text of this thesis