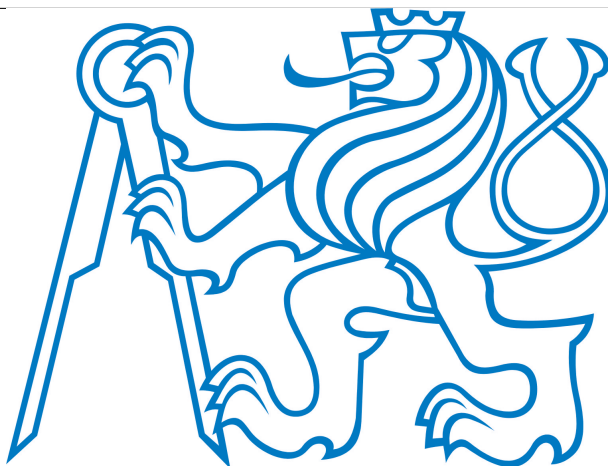


CZECH TECHNICAL UNIVERSITY
IN PRAGUE

FACULTY OF ELECTRICAL ENGINEERING



DIPLOMA THESIS

COMPARISON OF DETERMINISTIC DISTRIBUTED AND
MULTI-AGENT PLANNING TECHNIQUES

Author: Bc. Karel Durkota

Advisor: Ing. Antonín Komenda

May 2013

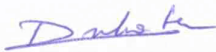
Copyright © 2013 Bc. Karel Durkota

All Rights Reserved

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 7. 5. 2013



_____ podpis

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Karel Durkota
Study programme: Open Informatics
Specialisation: Artificial Intelligence
Title of Diploma Thesis: Comparison of Deterministic Distributed and Multi-Agent Planning Techniques

Guidelines:

1. Study techniques for deterministic distributed and multi-agent planning: Distributed Graphplan, Multi-agent A*, and DCSP-based Multi-agent Planner.
2. Implement a multi-agent planner based on the principles of Distributed Graphplan and create common experimental platform for all three multi-agent planners.
3. Get familiar with the standard benchmarks from International Planning Competition (IPC) and extend them for multi-agent planning.
4. Experimentally validate properties of the planners (esp. time and communication complexity).
5. Discuss results of the experiments.

Bibliography/Sources:

- [1] Ghallab M., Nau D., and Traverso P. – Automated Planning: Theory & Practice – Morgan Kaufmann, 2004.
- [2] Iwen M., and Mali A. D. – Distributed Graphplan – In ICTAI '02 Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence. IEEE Computer Society Washington, DC, USA, 2002.
- [3] Pellier D. – Distributed Planning through Graph Merging – In ICAART 2010 Proceedings of the International Conference on Agents and Artificial Intelligence - Volume 2. INSTICC Press, Valencia, Spain, 2010.
- [4] Nissim R., Brafman R. I., and Domshlak C. – A General, Fully Distributed Multi-Agent Planning Algorithm – In AAMAS '10 Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems - Volume 1. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, USA, 2010.
- [5] Nissim R., Brafman R. I. – Multi-Agent A* for Parallel and Distributed Systems – In AAMAS '12 Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems – Volume 3. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, USA, 2012.

Diploma Thesis Supervisor: Ing. Antonín Komenda

Valid until: the end of the summer semester of academic year 2013/2014


prof. Ing. Vladimír Mařík, DrSc.
Head of Department




prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2013

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Karel Durkota
Studijní program: Otevřená informatika (magisterský)
Obor: Umělá inteligence
Název tématu: Porovnání deterministických distribuovaných a multi-agentních plánovacích technik

Pokyny pro vypracování:


1. Prostudujte techniky pro deterministické distribuované a multi-agentní plánování: Distributed Graphplan, Multi-agent A* a DCSP-based Multi-agent Planner.
2. Implementujte multi-agentní plánovač založený na principech Distributed Graphplan a vytvořte společnou experimentální platformu pro všechny tři multi-agentní plánovače.
3. Seznamte se se standardními benchmarky z Mezinárodní plánovací soutěže (IPC) a rozšiřte je pro multi-agentní plánování.
4. Experimentálně ověřte vlastnosti daných plánovačů (zejm. časovou a komunikační náročnost)
5. Diskutujte výsledky experimentů.

Seznam odborné literatury:


- [1] Ghallab M., Nau D., and Traverso P. – Automated Planning: Theory & Practice – Morgan Kaufmann, 2004.
- [2] Iwen M., and Mali A. D. – Distributed Graphplan – In ICTAI '02 Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence. IEEE Computer Society Washington, DC, USA, 2002.
- [3] Pellier D. – Distributed Planning through Graph Merging – In ICAART 2010 Proceedings of the International Conference on Agents and Artificial Intelligence - Volume 2. INSTICC Press, Valencia, Spain, 2010.
- [4] Nissim R., Brafman R. I., and Domshlak C. – A General, Fully Distributed Multi-Agent Planning Algorithm – In AAMAS '10 Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems - Volume 1. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, USA, 2010.
- [5] Nissim R., Brafman R. I. – Multi-Agent A* for Parallel and Distributed Systems – In AAMAS '12 Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 3. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, USA, 2012.

Vedoucí diplomové práce: Ing. Antonín Komenda

Platnost zadání: do konce letního semestru 2013/2014


prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry




prof. Ing. Pavel Ripka, CSc.
děkan

ABSTRACT

Deterministic domain-independent planning techniques for multiagent systems stem from the principles of classical planning. Three most recently studied approaches comprise (i) DisCSP+Planning utilizing distributed Constraint Satisfaction Problem solving for coordination of the agents and individual planning using local search, (ii) multiagent adaptation of A* with local heuristics and (iii) distribution of GraphPlan approach based on merging of planning graphs.

In this work, I summarize the principles of these three approaches and describe a novel implementation and optimizations of the Distributed Planning through Plan Merging (DPGM) multiagent GraphPlan approach. Domain and problem description were adapted for their utilization in multiagent planners. I experimentally validate influence of the parametrization of inner extraction phase of individual plans and compare the best results with the former two multiagent planning techniques.

Keywords: multi-agent systems, Automated planning, Distributed algorithms

ABSTRAKT

Distribuované doménově nezávislé plánovací techniky pro multiagentní systémy jsou založené na principech klasického plánování. V současné době byly představeny tři různé přístupy: (i) DisCSP+Planning využívající distribuovaného CSP pro řešení koordinace agentů a individuálního plánování pomocí místního prohledávání, (ii) upravený A* pro multiagentní problémy využívající lokálních heuristik a (iii) distribuovaného GraphPlanu založeného na slučování plánovacích grafů.

V této práci popisuji a shrnuji tři výše zmíněné přístupy, nové implementace a dále několik optimalizací algoritmu multiagentního distribuovaného plánování pomocí slučování plánu (DPGM). Popisy domén a plánovacích problémů byly uzpůsobené pro využití v multiagentních plánovačích. Experimentálně byly ověřeny vlivy parametrizace vnitřní extrakční fáze individuálních plánů. Nejlepší výsledky byly srovnány se dvěma výše zmíněnými multiagentními plánovacími technikami.

Klíčová Slova: multi-agentní systémy, automatizované plánování, distribuované algoritmy

ACKNOWLEDGMENTS

I would like to acknowledge Ing. Antonín Komenda for many advice, discussions and help about this thesis and Samuel Sydney for reviewing and correcting the grammar of this thesis. I would also like to thank my brother Eduard Durkota for the support in my studies.

Contents

Table of Contents	xiii
1 Introduction	1
1.1 Planning Formally	3
1.1.1 Block-World Problem Example	4
1.2 Multiagent Planning	6
1.3 DisCSP+Planning-based planner	7
1.4 Multi-Agent A*	8
2 Distributed Graphplan	11
2.1 Graphplan	11
2.1.1 Plan extraction using dynamic CSP	17
2.2 Distributed Graphplan	18
3 Distributed Planning through Graph Merging (DPGM)	19
3.1 Definitions	19
3.2 Algorithm	20
3.2.1 Global Goal Decomposition	20
3.2.2 Expansion and Planning Graph Merging	21
3.2.3 Individual Plan Extraction	22
3.2.4 Coordination	23
3.2.5 Example	23
3.3 Optimizations	25
3.3.1 Constraint cache	25
3.3.2 Ordering of agents	25
3.3.3 Removing unnecessary actions	26
3.4 Implementation	27
4 PDDL Adaptation for Multiagent Planning	31
4.1 PDDL	31
4.1.1 PDDL Example	32
4.1.2 PDDL Adaptation for Multiagent Planning	34

5 Experiments	39
5.1 Domains	39
5.1.1 Rovers	39
5.1.2 Satellites	40
5.1.3 Logistics	40
5.1.4 Linear Logistics	41
5.1.5 Deconfliction	41
5.2 Experiment settings	42
5.2.1 Comparison of Used CSP Solvers in DPGM	43
5.2.2 Comparison of the Multiagent Planners	44
5.3 Discussion	45
6 Conclusion	47
A List of used symbols	49
B Content of the CD	51
Bibliography	52
Bibliography	52

Chapter 1

Introduction

Over the last three decades planning became an important field of the research in the artificial intelligence. It is used in shipping and logistics, scheduling trains and buses, games, etc. Its importance has even more intensified with the advent of autonomous robots, intelligent agents and unmanned vehicles. Currently, the most popular Mars rover, Opportunity [15] and Curiosity [4] created by NASA, could serve as a great example of the autonomous robots, that could not function as well as they do, without their planning abilities.

Planning is generally a search for a sequence of actions that leads from an initial state to a goal state. Planners are divided into the three main categories: *Domain-specific*, *Domain-independent* and, *Configurable*. *Domain-specific* planners are the most successful and powerful because they are tailored for a specific problem, for which they give excellent results; however, for any other domain, they will not work well, if at all. *Domain-independent* planners in theory should work in any planning domain; however, it is not feasible to develop a planner that works in every possible domain. Therefore certain assumptions are made to restrict the set of domains. In the classical planning problem, which is primarily dealt with in this work, the assumptions about the environment are [5, 16]:

finite - it contains finite number of states, actions and events

fully observable - planner knows all the relevant information to make correct choice of the action

deterministic - each action has exactly one predictable outcome

static - environment cannot change during the deliberation

implicit time - actions have no time durations - they are instantaneous

sequential plans - next state depends on the previous actions.

Finally, the *configurable* planner is somewhere between the domain-specific and domain-independent planners. It consists of a domain-independent search engine that uses domain-specific search-control knowledge, given to the planner as an input together with the problem. It can be used for different domains by altering the domain descriptions, however the planning engine is intact [10].

Planning can also be either single-agent, where one agent performs all actions; or multiagent, where multiple agents act cooperatively. The multiagent planning (MAP) resembles more the reality of the complex problems. For instance, to build a house, each performer can represent an agent: concrete mixer, brick layer, digger, etc. Some problems even require multiple agents, since one agent cannot perform a task alone. Moreover, team of the agents might achieve goal faster and more efficiently and their solutions are usually more robust. Disadvantages of this approach is that agents' coordination is very hard to solve, since there is no global view. Criteria for the MAP are usually: (i) computation complexity, (ii) communication complexity among the agents, (iii) flexibility, how much freedom agents give each other, (iv) robustness, as a resistance of the plan to changes in the environment, (v) plan quality, and (vi) scalability for changes in size of the problem or number of agents.

Multiagent planning can be either centralized multiagent planning (CMAP) and distributive multiagent planning (DMAP), both of which bring certain advantages and disadvantages. A centralized approach is often faster than DMAP, gives better results and requires lower communication (only to gather the problem information and send the results back to the agents). After the centralized computation, the resulting plan has to be decomposed for agents, so that they can execute

the plan distributively. DMAP, which in many aspects overlaps with the parallel algorithms, on the other hand, uses agents' cooperation, instead of centralized computation. This approach gives agents opportunity to behave in self-interest and computation runs in parallel.

1.1 Planning Formally

In this work I use definition for the planning domain, problem and solution as it is defined for STRIPS in [5, 7, 14]. *Planning domain* is a restricted state-transition system $\Sigma = (\mathbb{S}, A, \text{app})$ over the finite set of *proposition symbols* $P = \{p_1, \dots, p_n\}$, where:

- *proposition* $p \in P$ is a statement, which can be either true or false, i.e., (block is in location A), (car has fuel), etc.;
- *state* $s \in \mathbb{S}$ is a subset of P and notifies which propositions from P currently hold. If proposition $p \in s$, then proposition p holds in state s , otherwise p does not hold¹;
- *action* $a \in A$ is a triple $a = \langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$. Set $\text{pre}(a) \subseteq P$ is called *preconditions* of action a and sets $\text{add}(a), \text{del}(a) \subseteq P$ are called *positive* and *negative effects* of action a , respectively; it is required that $\text{add}(a) \cap \text{del}(a) = \emptyset$;
- app is an *application* function, defined as following:

$$\text{app}(a, s) = \begin{cases} (s \cup \text{add}(a)) \setminus \text{del}(a) & \text{if } \text{pre}(a) \subseteq s \\ \text{not defined} & \text{otherwise} \end{cases}$$

where $a \in A$ and $s \in \mathbb{S}$. I also define an application function app for a set of actions $A_i \subseteq A$ over state s , denoted as $s_2 = \text{app}(s, A_i)$, if actions in A_i are *independent* from each other; otherwise set of actions A_i are not applicable on state s . Actions a and b are *independent* if

¹ $p \notin s$ can be also interpreted as $\neg p \in s$

$(\text{pre}(a) \cup \text{add}(a)) \cap \text{del}(b) = \emptyset$ and $(\text{pre}(b) \cup \text{add}(b)) \cap \text{del}(a) = \emptyset$, and they can be applied simultaneously or in arbitrary order on state s . Resulting state s_2 is achieved by application all the actions in A_i on s , i.e., $s_2 = (s \cup \bigcup_{a \in A_i} \text{add}(a)) \setminus (\bigcup_{a \in A_i} \text{del}(a))$.

A *planning problem* is defined as a triple $\mathcal{P} = (\Sigma, s_0, G)$, where:

- $s_0 \subseteq P$ is set of *initial propositions* (or $s_0 \in \mathbb{S}$ is *initial state*)²
- $G \subseteq P$ is called *goal propositions*. Every $s \in \mathbb{S}$ that satisfies G , in other words $G \subseteq s$, is a *goal state*. Set of goal states is $\mathbb{S}_G \subseteq \mathbb{S}$.

Finally, a *plan* to the planning problem $\mathcal{P} = (\Sigma, s_0, G)$ can be either:

- *totally ordered* sequence of actions $\pi = \langle a_0, \dots, a_N \rangle$, where $a_i \in A$, $i = 1, \dots, N$; and it is applicable to a state s_i defining a sequence $\langle s_0, \dots, s_N \rangle$ such that $s_{i+1} = \text{app}(s_i, a_i)$ and $s_N \in \mathbb{S}_G$; N is the *number of the actions* of the plan and the *length* of the plan; or,
- *partially ordered* sequence of actions $\pi = \langle A_0, \dots, A_K \rangle$, where $A_i \subseteq A$, $i = 1, \dots, N$; and it is applicable to a state s_i defining a sequence $\langle s_0, \dots, s_N \rangle$ such that $s_{i+1} = \text{app}(s_i, A_i)$ and $s_N \in \mathbb{S}_G$ ³; K is the *length* of the plan and $\sum_{i=0}^K |A_i|$ is *number of actions*.

1.1.1 Block-World Problem Example

In the Block-world planning problem the goal is to find a plan of how to rearrange blocks from the initial configuration to the goal configuration. Blocks can be put on the table or on top of each other. A block can be moved only if it is the top block; additionally, only one block can be moved at a time. Figure 1.1 illustrates all possible states of this problem. Action MFT stands for Move-From-Table, MTT stands for Move-To-Table, and M stands for Move. Set of propositions P , set of

²in this work, the state $s \in \mathbb{S}$ is represented by a set of *propositions* from P . So terms *state* and *set of propositions* are interchangeable and have the same meaning.

³*partially ordered* plan can be easily flattened to a *totally ordered* plan by setting arbitrary orders to sets A_i in respective way.

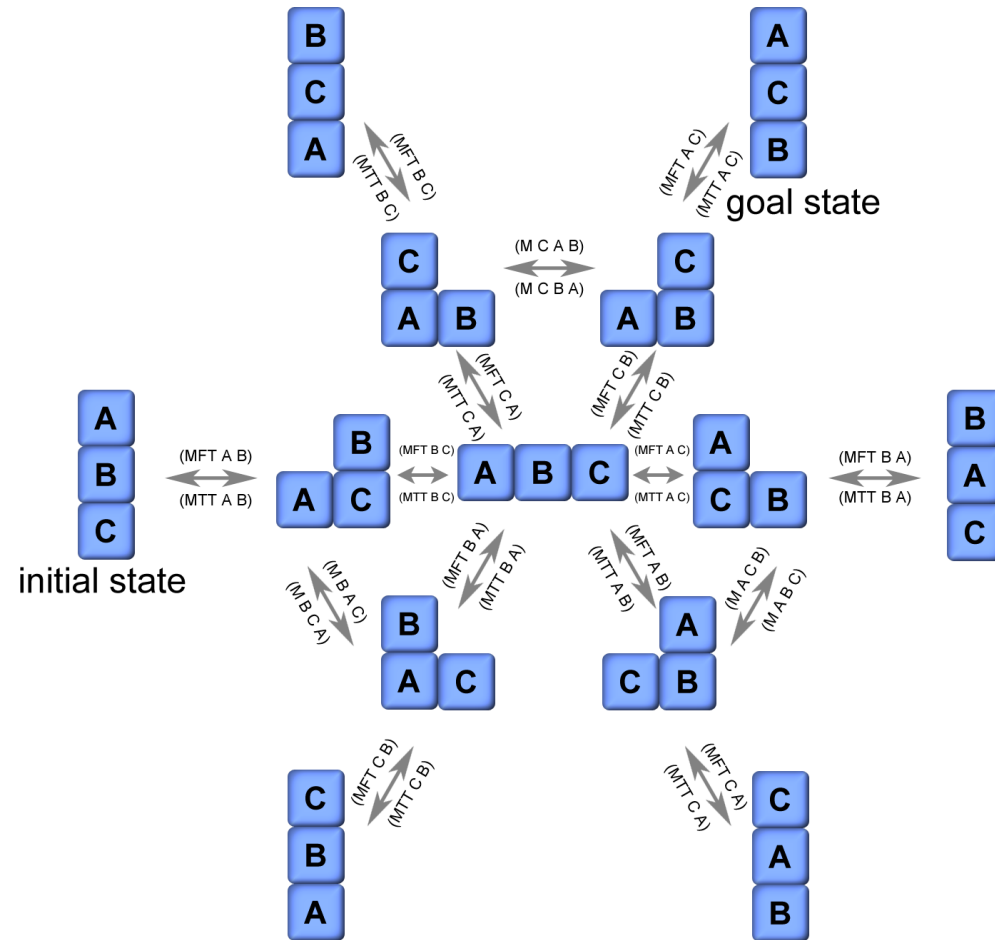


Figure 1.1 State-space of a simple block-world planning problem. Action M means move, MFT means moveFromTable and, MTT means moveToTable.

states \mathbb{S} , set of action A , the initial state s_0 and the goal state g , respectively, are following:

$P = \{(onTable\ A), (on\ A\ B), (on\ A\ C), (onTable\ B), (on\ B\ C), (on\ B\ A), (onTable\ C), (on\ C\ A), (on\ C\ B), (empty\ A), (empty\ B), (empty\ C)\}$, where:

- (onTable A) means that block A is on the table;
- (on A B) means that block A is laid on block B; and,
- (empty A) means that there is no block laid on block A.

$\mathbb{S} = \mathbb{P}(P)$ is a powerset of P ; its cardinality is $|\mathbb{S}| = 2^{|P|} = 4096$.⁴

⁴the number 4096 represents all the possible combination; however, only 13 states are consistent and reachable from the initial state using actions from A , so the search state space is much smaller.

$A = \{(moveToTable\ A\ B), (moveToTable\ A\ C), (moveToTable\ B\ A), (moveToTable\ B\ C), (moveToTable\ C\ A), (moveToTable\ C\ B), (moveFromTable\ A\ B), (moveFromTable\ A\ C), (moveFromTable\ B\ A), (moveFromTable\ B\ C), (moveFromTable\ C\ A), (moveFromTable\ C\ B), (move\ A\ B\ C), (move\ A\ C\ B), (move\ B\ A\ C), (move\ B\ C\ A), (move\ C\ A\ B), (move\ C\ B\ A)\}$, where:

$(moveToTable\ A\ B) = (\{(empty\ A), (on\ A\ B)\}, \{(onTable\ A), (empty\ B)\}, \{(on\ A\ B)\})$ means to move block A from block B to the Table;

$(moveFromTable\ A\ B) = (\{(onTable\ A), (empty\ B)\}, \{(on\ A\ B)\}, \{(empty\ B), (onTable\ A)\})$ means to move block A from a Table onto block B;

$(move\ A\ B\ C) = (\{(on\ A\ B), (empty\ A), (empty\ C)\}, \{(on\ A\ C), (empty\ B)\}, \{(empty\ C)\})$ means to move block A from block B onto block C; and, similarly the rest.

$s_0 = \{(on\ A\ B), (on\ B\ C), (onTable\ C), (empty\ A)\}$.

$G = \{(on\ A\ C), (on\ C\ B)\}$. Note, that it is not necessary to describe the whole goal state – leaving out the propositions (empty A) and (onTable B) – but only the propositions that are wished to hold at the goal state.

An optimal *plan* with the minimal number of actions is $\pi = \langle (moveToTable\ A\ B), (moveToTable\ B\ C), (moveFromTable\ C\ B), (moveFromTable\ A\ C) \rangle$. However, there are more non-optimal plans that solves this problem, in fact, infinitely many.

1.2 Multiagent Planning

Planning for MAP is a search for a plan for each agent, assuming that agents have to cooperate in order to reach the global goal. Problem for set of agents $\mathcal{A} = \{ag_\alpha\}_{\alpha=1}^K$ is given by quadruple

$\mathcal{P} = \langle P, \{A^\alpha\}_{\alpha=1}^K, s_0, G \rangle$, where:

- P is set of all propositions and $P^\alpha \subseteq P$ is a set of propositions that are related to agent ag_α , meaning, that for every $p \in P^\alpha$ exists action $a \in A^\alpha$, such that $p \in \{pre(a) \cup add(a) \cup del(a)\}$;

- A^α is a set of actions that agent ag_α can perform. Every action $a \in A^\alpha$ is defined as $a = \langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$, where $\text{pre}(a), \text{add}(a), \text{del}(a) \subseteq P^\alpha$ and $\text{add}(a) \cap \text{del}(a) \neq \emptyset$;
- $s_0 \subseteq P$ is a set of initial propositions; and,
- $G \subseteq P$ is a set of goal propositions.

Agents' actions are divided into two groups: *public actions* and *private actions*. Public actions are those that can be influenced by other agents in some way. Formally, action $a \in A^\alpha$ is agent's ag_α public action if there is an action $b \in A^\beta$ of an agent ag_β , $\alpha \neq \beta$, such that $(\text{pre}(a) \cup \text{add}(a) \cup \text{del}(a)) \cap (\text{pre}(b) \cup \text{add}(b) \cup \text{del}(b)) \neq \emptyset$; otherwise, action considered to be private action of agent ag_α .

1.3 DisCSP+Planning-based planner

The DisCSP+Planning-based planner, introduced in [13], can be described as two interleaving components, a *coordination component* and a *individual planning component*; both of which require separation actions into private and public actions of each agent. The *coordination component* deals only with the public actions. It searches for a sequence of interaction points between the agents' plans. For a given length of the public part of the plan δ , it tries to assign different public actions of each agent in the different time-steps so that the global goal is satisfied. This is the interaction part, so the resulting plan of each agent has to satisfy the requirements put by the rest of the agents and vice versa. These requirements are described in form of *coordination constraints* in the inner DisCSP problem. Solving this problem effectively means solving the multiagent planning problem. If some agent or the team as a whole could not solve the DisCSP, then δ , the length of the coordination public part of the plan is increased by one and the whole process is repeated.

The *individual planning component* forms the other type of constraints for the DisCSP process, encoding the required local parts of the plan. The individual planning constraints limits usage of

the public actions in the coordination part of the plan such that the gaps between them can be filled by sequences of individual actions of the individual agents.

Nissim et al. experimentally compared DisCSP+Planning-based planner with the centralized *Fast-Forward (FF)* [6] solver over three types of problem domains: LOGISTICS, ROVERS and SATELLITES. Problems varied in required number of agents, from 2 agents up to 18 agents. Centralized *FF* algorithm showed to be much better in LOGISTICS problems; on the other hand, problems ROVERS and SATELLITES were solved about 10x faster by DisCSP+Planning-based planner than (*FF*).

1.4 Multi-Agent A*

The algorithm proposed in [12] is inspired by the well-known A* algorithm. Similarly to centralized A*, the multiagent distributed A* (MAD-A*) maintains *open lists* for all agents, that keeps track of the so far unvisited states, and *closed lists*, that keep the track of the already visited states. Each agent also uses a local heuristic to decide which state from the open list it should expand as next. As stated in the paper, each agent can use different heuristics.

In the MAD-A*, similarly to the DiscCSP+Planning, it is firstly necessary to separate every agents' public and individual actions. The algorithm runs simultaneously for each agent. During the search, the agents send messages to each other to distribute the search at points, where other agents can follow. Effectively, it means the messages are sent only for states achieved by public actions. Each such message consists of a state s , its cost value $g_{ag_\alpha}(s)$ and a heuristic estimate $h_{ag_\alpha}(s)$. In decoupled problems, this principle allows distribution of the knowledge about the entire search space among the agents. When an agent receives a message with a state s , it decides either to: visit the state (by adding it to its *open list*), update its knowledge about s , or discard it if it knows a better path to the state. The search terminates if an agent expands a state which is

compatible with the goals set G and the resulting plan is ensured to be globally optimal.

In [12] MA-A* was tested in its parallel (MAP-A*) and distributed (MAD-A*) setting and compared with parallel and distributed centralized A* over LOGISTICS, ROVERS, SATELLITES and ZENOTRAVEL problems, varying in problems difficulty and number of agents. Authors also compared different heuristics, namely, LM-cut and Merge&Shrink for both, MAP-A* and MAD-A*. Results showed that MAP-A* using LM-cut heuristic outperformed centralized A* in almost all instances of the problems. It showed to be faster in range from 1 up to 19.5 on instances that both algorithms solved. Heuristic Merge&Shrink was less improving, but yet was better than centralized A*. In the distributed setting centralized A* showed to be faster when LM-cut heuristic was used; but when the Merge&Shrink heuristic was used, MAD-A* was usually faster and solved problems that centralized A* could not solve at all.

Nowadays, on-line centralized FF planner⁵ showed to solve these problems instantaneously (in 0.1 seconds). The reason that DisCSP+Planning-based planner and multiagent A* showed to be often better than centralized FF planner is probably due to the fact that authors used an old FF implementation planner.

⁵<http://fai.cs.uni-saarland.de/hoffmann/ff.html>

Chapter 2

Distributed Graphplan

One of the most efficient centralized planners is the Graphplan algorithm. Thus, I would like to begin this chapter by introducing this algorithm and its improvement using CSP solver as a part of its plan extraction. Graphplan is an essential part for both, Distributed Graphplan and Distributed Planning through Graph Merging (DPGM) algorithms, that were mainly studied and implemented in this work.

2.1 Graphplan

Graphplan, developed by Avrim Blum and Merrick Furst in 1995 [2], builds a *planning graph* (also called *plangraph*) structure and analyzes it before generating the plan, rather than blindly searching for a valid solution. The planning graph is not a state-search graph, but rather *flow network*, which encodes in its structure the initial conditions, goals and notion of the time steps. Graphplan's input is a planning problem $\mathcal{P} = \langle P, I, G, A \rangle$, where: P is set of propositions, $I \subseteq P$ is set of initial propositions, $G \subseteq P$ is set of goal propositions, and A is set of actions, where each action is defined as $a = \langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$ and $\text{pre}(a), \text{add}(a), \text{del}(a) \subseteq P$. Output of the graphplan is a partially ordered plan $\pi = \langle A_0, A_1, \dots, A_N \rangle$, where $A_i \subseteq A$ for $i = 1, \dots, N$. Graphplan consists of two phases,

planning graph construction and solution extraction.

First phase, the *planning graph construction* (also called *plangraph construction*), builds the plangraph, which contains alternating *proposition levels*, denoted as S_0, S_1, \dots and *action levels*, denoted as O_0, O_1, \dots . Proposition level is a set of propositions that can be reached in that level. First proposition level S_0 equals to the set of initial propositions, the others are generated. Action level is a set of actions, that could be applicable in that level, under certain conditions. Action level O_i contains all the actions, whose preconditions are met in previous proposition level S_i . Proposition level S_{i+1} is generated as union of S_i and effects of actions in O_i . Thus, $O_i = \{a \in A \mid \text{pre}(a) \subseteq S_i\}$ and $S_{i+1} = S_i \cup \bigcup_{a \in O_i} \text{add}(a) \cup \text{del}(a)$. Additionally, dummy action called *noop action* (also called *maintenance action*) is introduced for every proposition p as $\text{noop}_p = \langle \{p\}, \{p\}, \{\} \rangle$. This actions transfers the propositions from previous propositions level to the next one, reasoning, that if no action changed the truth of the proposition, it remains true.

There are two types of edges in the planning graph: (i) *precondition edges*, that connect propositions from O_i with actions in A_i denoting which propositions are the action's preconditions and (ii) *effect edges*, connecting action from A_i with propositions in O_{i+1} , denoting action's add-effects and del-effects.

Graphplan's major effectivity is in maintaining the set of *mutexes* (stands for *mutually exclusive*)—induced knowledge between two propositions or two actions in every level. Mutex in action level is a couple (a, b) , where $a, b \in O_i$, denoting that action a and b cannot be performed together in this level. Set of action mutexes in level i is denoted as μ_{O_i} . Similarly, mutex in proposition level is a couple (p, q) , where $p, q \in S_i$, denoting that both propositions cannot be true in this level. Set of proposition mutexes in level i is denoted as μ_{A_i} . There are several types of mutexes:

- Actions a and b in action level O_i are in:
 - *interference mutex*, if either of them deletes either precondition or an add-effect of the other. Thus, if $\text{pre}(a) \cup (\text{add}(b) \cap \text{del}(b)) \neq \emptyset$ or $\text{pre}(b) \cup (\text{add}(a) \cap \text{del}(a)) \neq \emptyset$; or,

- *competing needs mutex*, if they have any preconditions in proposition level S_{i-1} , which are in mutex.
- Propositions p and q from the proposition level S_i (noop actions are also considered) are in mutex if:
 - p is negation of q ; or,
 - all actions in A_{i-1} , leading to p (that have p in their add-effect) are mutexes with all the actions, leading to q . Formally, p and q are in mutex, if every action in $\{a \in A_{i-1} | p \in \text{add}(a)\}$ is in mutex with every action in $\{b \in A_{i-1} | q \in \text{add}(b)\}$.

The plangraph $pg = \langle S_0, \mu_{S_0}, O_0, \mu_{O_0}, \dots, O_{N-1}, \mu_{O_{N-1}}, S_N \rangle$ consists of sequences of level and its mutex pairs, either until it reaches the proposition level S_N that contains the goal propositions, such that no two of them are in mutex; or no change in propositions levels was detected, e.g., $S_N = S_{N-1}$ and $\mu_{S_N} = \mu_{S_{N-1}}$, which is called *fixed point*. In the former case, it continues with the *plan extraction phase*; in the latter case, no solution exists, and algorithm terminates.

In the *solution extraction* phase the backsearch algorithm is run to find a plan. For every proposition level, starting from S_N and ending at S_0 , backsearch algorithm sets new subgoal. Subgoal is a set of propositions that consequently have to be supported by actions in the immediately preceding action level. Subgoal $S_{i+1}^{subgoal} \subseteq S_{i+1}$ in level $i+1$ is supported by set of actions $O_i^{supp} \subseteq O_i$ from the previous level i , if for every $p \in S_{i+1}^{subgoal}$: exists action $a \in O_i^{supp}$ so that $p \in \text{add}(a)$ and no two actions in O_i^{supp} are in mutex. New subgoal $S_i^{subgoal}$ is computed as $S_i^{subgoal} = \{\text{pre}(a) | a \in O_i^{supp}\}$. If no set of supporting actions can be generated in level O_i , algorithm backtracks to the level O_{i+1} , and there tries to find a different set of supporting actions. Algorithm continues in this manner either until it runs out of all possibilities of supporting sets in all levels, which means that no solution exists; or proposition level S_0 is reached and partially ordered plan $\pi = \langle O_0^{supp}, O_1^{supp}, \dots, O_N^{supp} \rangle$ is returned.

In Algorithm 3 is presented the pseudo-code of the Graphplan algorithm. As an input it receives the planning problem $\mathcal{P} = \langle P, I, G, A \rangle$, where P is a set of propositions, $I \subseteq P$ is a set of initial propositions, $G \subseteq P$ is a set of goal propositions and A is set of actions. First, the Graphplan algorithm initializes first proposition level. Then it continues to build a new action level, action mutex level, proposition level and proposition mutex level until the *fixed point is reached*. After each newly built level, if goal propositions were such that no two goal propositions are in mutex, plan extraction phase is run through backsearch algorithm. Backsearch pseudo-code is illustrated in Algorithm 2. As an input it gets a set of propositions $S^{subgoal}$ as a subgoal and the plangraph with N levels. If plangraph has zero levels, empty plan is returned. Otherwise, it attempts to find set of support actions O_{N-1}^{supp} in the last action level O_{N-1} , such that no two actions in O_{N-1}^{supp} are in mutex. Next, compute new subgoal propositions $S_{new}^{subgoal}$ that is a set of preconditions of actions in O_{N-1}^{supp} . Recursively call this procedure with new subgoal $S_{new}^{subgoal}$ and the plangraph without the last level.

Figure 2.1 depicts the partial planning graph of the block-world problem, presented in Example 1.1.1. Lines from the propositions in S_i to the actions in O_i represent the actions' preconditions; solid lines from the actions in O_i to the preconditions in S_{i+1} represent the add-effects, and dotted lines represent the delete-effects. Big dots on solid lines in the action level denote the noop actions of the connected proposition. Finally, the green curves that link two actions or two propositions together denote those actions or propositions that are in mutex. Solution of the problem from 1.1.1 is actually in 4th level, which requires big plangraph. Assume a different goal state: to put all the blocks on the table next to each other. This solution can be found in the depicted plangraph, as it requires only two actions. After building this plangraph, backsearch algorithm attempts to find this plan. The search begins with the goal propositions in last proposition level S_2 (propositions in S_2 in red color). Next, it finds the supporting actions that have as an effect the these propositions. The support actions are the action level in red color in action level O_1 (two noop actions

Algorithm 1: Graphplan

```

input :  $\mathcal{P} = \langle P, I, G, A \rangle$ ; // planning problem
output:  $\pi = \langle A_0, A_1, \dots, A_N \rangle$ ; // resulting plan

 $S_0 \leftarrow I$ ;
 $\mu_{S_0} \leftarrow$  generate mutexes in proposition level  $S_0$ ;
 $N \leftarrow 0$ ;
repeat
  |  $O_N \leftarrow$  build new action level  $O_N$ ;
  |  $\mu_{O_N} \leftarrow$  generate mutexes in action level  $O_N$ ;
  |  $N \leftarrow N + 1$ ;
  |  $S_N \leftarrow$  generate new proposition level;
  |  $\mu_{S_N} \leftarrow$  generate mutexes in proposition level  $S_N$ ;
  | if  $G \subseteq S_N$  and no two propositions in  $G$  are in mutex then
  | |  $\pi \leftarrow$  Backsearch( $G, \langle S_0, \mu_{S_0}, O_0, \mu_{O_0}, S_1, \mu_{S_1}, O_1, \mu_{O_1}, \dots, S_{N-1} \rangle$ );
  | | if plan  $\pi$  found then
  | | | return  $\pi$ ;
  | | end
  | end
until fixed point is not reached;
No solution exists

```

Algorithm 2: Backsearch

```

input :  $S^{subgoal}, \langle S_0, \mu_{S_0}, O_0, \mu_{O_0}, \dots, S_N \rangle$ 
output:  $\pi = \langle A_0, A_1, \dots, A_N \rangle$ ; // resulting plan

if  $N$  equals 0 then
  | return  $\emptyset$ ;
end

while has not ran out of the support action sets to support the  $S^{subgoal}$  do
  |  $O_{N-1}^{supp} \leftarrow$  get set of the support actions to support  $S^{subgoal}$ , so that no two actions are
  | in mutex;
  |  $S_{new}^{subgoal} \leftarrow$  compute new subgoal as  $\{\text{pre}(a) | a \in O_{N-1}^{supp}\}$ ;
  |  $\pi \leftarrow$  Backsearch( $S_{new}^{subgoal}, \langle S_0, \mu_{S_0}, O_0, \mu_{O_0}, \dots, S_{N-1} \rangle$ );
  | if returned plan is not failed then
  | | return  $\langle \pi, O_{N-1}^{supp} \rangle$ ;
  | end
end
return failed;

```

and the (MTT B C) actions). A set of preconditions of these three actions will be new subgoal (the propositions in red color in S_1). After one more repetition algorithm terminates, as it reaches first proposition level S_0 . Thus, the plan is $\pi = \langle \{(MTT A B)\}, \{(MTT B C)\} \rangle$ (the noop actions can be excluded from support actions in the plan as they neither add nor delete any proposition).

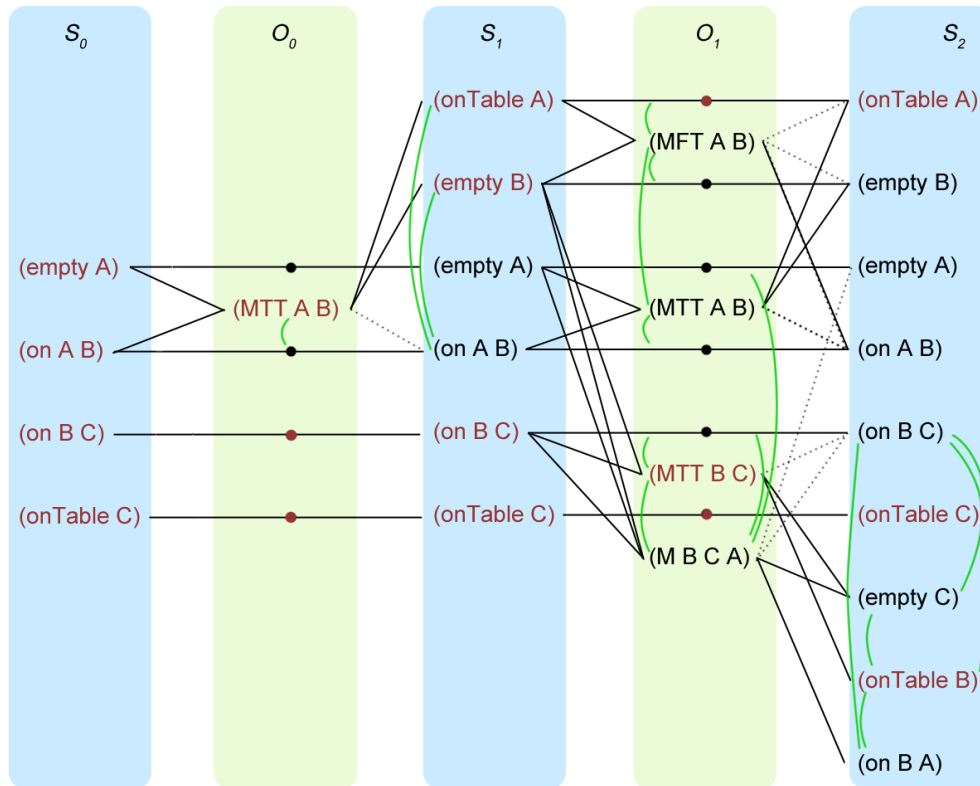


Figure 2.1 Graphplan of a simple block-world planning problem.

It is worth mentioning a few important properties of a plangraph. Proposition levels monotonically increase, i.e., $S_i \subseteq S_{i+1}$; and so does the action levels, i.e., $O_i \subseteq O_{i+1}$. But proposition mutex relationships monotonically decrease, meaning, that if propositions p and q are not in mutex in level S_i , they will never become mutexes in any future proposition level. Similarly, the action mutex relationships decrease. These properties and the fact that P and A are finite sets guarantee that there exists the *fixed point* for every plangraph.

2.1.1 Plan extraction using dynamic CSP

Kambhampati and Subbarao presented in [8] a method for converting the plan extraction problem into the dynamic CSP problem. This method can replace the backsearch algorithm in the Graphplan. The conversion is following: every proposition from the plangraph will represent a variable and every action will represent a value. Variable's domain are the values of the support actions of the proposition represented by that variable. There are two types of constraints in this CSP, *mutex constraints* and *activation constraints*. The mutex constraints forbid assigning to two variables the values of the actions, that are in mutex in the original plangraph. Activation constraints represent the computation of new subgoals from the support actions in the backsearch. In dynamic CSP if variable is deactivated, no assignment is required for valid solution; if variable is activated, one of the values from the domain has to be assigned to it. At the beginning all variables are deactivated except the variables that represent the goal propositions. The activation constraint states that if value is assigned to the activated variable, then all variables—that represent the preconditions of the action which value is assigned—are also activated. Actions that are represented by the values of the activated variables in the CSP solution, will be in the solution plan. From the knowledge which variables represented which proposition can be induced in which level each action is performed.

According to [3] this method does not improve the plan extraction phase too much, but it is required by algorithms Distributed Graphplan and DPGM, since they add a few more constraints into the CSP assignment. In my work I compared both methods for plan extraction, the backsearch and using CSP conversion, and CSP conversion was noticeably faster and required significantly less memory. This was not the subject of this particular work; hence no statistical measurements were made.

2.2 Distributed Graphplan

Distributed Graphplan (DGP) algorithm was introduced by Iwen et al. in [7]. This algorithm solves the problem by decomposing it into two subproblems. Each subproblem is solved separately by the Graphplan algorithm. Consequently, algorithm attempts to find plans that are not conflicting. Note, that DGP is not multiagent planner, since it does not work with the agents, that could have limited actions, but searches two plans in which both can perform all the actions.

DGP algorithm expects on its input manually decomposed two subproblems $\mathcal{P}_1 = \langle I, G_1, A_1 \rangle$ and $\mathcal{P}_2 = \langle I, G_2, A_2 \rangle$, where $G_1 \cup G_2 = G$ and $A_1 \cup A_2 = A$; and $\mathcal{P} = \langle I, G, A \rangle$ is the original problem, $I, G, G_1, G_2 \subseteq P$ and $A_1, A_2 \subseteq A$. First, it constructs two plangraphs, PG_1 that solves subproblem \mathcal{P}_1 and PG_2 that solve subproblems \mathcal{P}_2 , from which plans π_1 and π_2 are extracted, respectively. If length of the plans differ, the shorter one is extended with *noops* actions until their length are equal. Next, it constructs global plan π , as union of π_1 and π_2 in their respective levels, and checks its validity by progression. If it is valid, algorithm terminates; otherwise conflict resolution step is followed. The conflict between the plans can be caused only due to *static mutex*¹, which have to be resolved. One of the two conflict actions are replaced by a new action from the planning graph where the conflict action came from. The action is chosen so that the plan still solves the individual goal. This process is repeated until it finds non-conflict plans; or, runs out of new actions, after which new level is generated in both plangraphs and whole process repeats.

DPG algorithms was implemented and tested in this work. However, this algorithm is not suited for multiagent planning problems, so no statistical result were carried out. Yet some parts of it were reused for DPGM implementation.

¹*static mutex* is found by examining the preconditions and effects of actions in one level; whereas *dynamic mutex* is propagated static mutex through more levels.

Chapter 3

Distributed Planning through Graph

Merging (DPGM)

DPGM algorithm, presented by Damien Pellier in 2010 [14], was the main subject of this work. Primarily this algorithm was studied into the details, consequently implemented and even improved. Aim of this work was to compare it to DisCPS+Planning-based planner and MA-A* over standard multiagent planning problems.

3.1 Definitions

Planning problem is defined as $\mathcal{P} = \langle P, \mathcal{A}, I, G \rangle$, where:

- P is set of propositions and $P_\alpha \subseteq P$ is set of propositions concerning the agent ag_α ;
- $\mathcal{A} = \{ag_\alpha\}_{i=1}^K$ is set of agents, where ag_α can perform action A_α ;
- $I \subseteq P$ is set of propositions that hold in the initial state; and,
- $G \subseteq P$ is set of propositions, that must hold in a goals state.

The set of propositions $G_\alpha \subseteq G$ is agent's ag_α individual goal, and for every goal proposition $p \in G_\alpha$ must exist supporting action $a \in A_\alpha$, so that $p \in \text{add}(a)$; additionally, $G_1 \cup \dots \cup G_K = G$.

The plan $\pi_\alpha = \langle A_0^\alpha, A_1^\alpha, \dots, A_N^\alpha \rangle$ is agent's $ag_\alpha \in \mathcal{A}$ *individual solution plan* in form of partially ordered set if every $A_i^\alpha \in \pi_\alpha$ is applicable on state s_i^α defining sequence of states $\langle s_0^\alpha, \dots, s_N^\alpha \rangle$ such that $G_\alpha \subseteq s_N^\alpha$.

The plan $\Pi = \langle A_0, A_1, \dots, A_N \rangle$ is a *global solution* in form of partially ordered set solving the problem $\mathcal{P} = \langle P, \mathcal{A}, I, G \rangle$ if:

- individual plan $\pi_\alpha = \langle A_0^\alpha, A_1^\alpha, \dots, A_N^\alpha \rangle$ of every agent ag_α reaches its individual goal G_α ; and
- every $A_i = \bigcup_{\alpha=1}^K A_i^\alpha$.

3.2 Algorithm

The planner as a whole can be described by following five phases: *global goal decomposition, expansion, planning graph merging, individual plan extraction, coordination*. The result is in form of a *coordinated individual solution plan*.

3.2.1 Global Goal Decomposition

In the first phase, the *global goal decomposition*, each agent ag_α creates an individual goal $G_\alpha \subseteq G$, i.e., a subset of the propositions from the global goal that it can reach. Proposition $p \in G$ is in the individual goal G_α of an agent ag_α if exists an action $a \in A^\alpha$ such that $p \in \text{add}(a)$. If any proposition from the global goal cannot be assigned to any agent, then problem \mathcal{P} has no solution. Algorithm 3 illustrates the procedure. For every proposition p in the goal it tests searches which agents have an action, that would have p in add-effect. Those agents will have p in their individual goals.

Algorithm 3: Goal Decomposition

```

input :  $\mathcal{A}$ ; // set of all agents
         $G$ ; // set of goal propositions
output:  $\langle G_0, G_1, \dots, G_N \rangle$ 

foreach  $p \in G$  do
  foreach agent  $ag_\alpha \in \mathcal{A}$  do
    if exists  $a \in A_\alpha$ , so that  $p \in \text{add}(a)$  then
       $G_\alpha \leftarrow G_\alpha \cup p$ ;
    end
  end
end

```

3.2.2 Expansion and Planning Graph Merging

In next two phases, the *expansion* and *planning graph merging*, every agent builds an individual planning graph via Graphplan algorithm. First, every agent builds a new level in their planning graphs. Afterwards, actions from the new level are shared with the rest of the agents. Other agents include into their respective plangraphs the *relevant* actions from actions, that are shared with them. Agent's ag_α action $a_\alpha \in A^\alpha$ is *relevant* to agent ag_β in two cases:

- if $\text{add}(a_\alpha)$ contains a proposition that another action $a_\beta \in A^\beta$ uses in $\text{pre}(a_\beta)$ or $\text{add}(a_\beta)$, then the action a_α *promotes* the action a_β ; or,
- if $\text{del}(a_{ag_\alpha})$ contains a proposition that another action $a_\beta \in A^\beta$ uses in $\text{pre}(a_\beta)$ or $\text{add}(a_\beta)$, then the action a_α *threats* the action a_β .

The threats and promotions are relevant to the agent, because they may help him to achieve its individual goal. This is the coordination point of agents if agent uses the threat or promotion action in its individual plan. The algorithm alternates between the *expansion* phase, where all the agents build new levels in their planning graphs and the *planning graph merging* phase, when all the agents share their actions. This is done until all agents reach their individual goals in their planning graphs or all planning graphs reach their *fixed points*. Note, that all plangraphs will

always have equal number of levels. Although some agent's plangraphs may have reached goal propositions earlier, they will keep expanding their plangraphs until all plangraphs reach individual goals. Algorithm 4 illustrates the pseudo-code of the both phases. In the first for-cycle the each agent's planning graph is extended by one level and in the second for-cycle agents share their actions from the last level among each other. The *relevant* actions—*threats* and *promotions*—are included into the agent's planning graphs.

Algorithm 4: Expansion and Merging Planning Graphs

```

input :  $\mathcal{A}$ ; // set of all agents
 $\langle G_1, G_2, \dots, G_N \rangle$ ; // set of individual goals for each agent
output:  $\langle pg_1, \dots, pg_N \rangle$ ; //  $pg_\alpha = \langle pg_\alpha^1, \dots, pg_\alpha^L \rangle$  is agent  $ag_\alpha$ 's plangrap,
    and  $pg_\alpha^i$  is  $i^{th}$  level of  $pg_\alpha$ 

 $L \leftarrow 1$ ;
while until all plangraphs  $pg_\alpha^L$  contain  $G_\alpha$  in non-mutex way do
  for every agent  $ag_\alpha \in \mathcal{A}$  do
    | build level  $pg_\alpha^L$ ;
  end
  for  $\alpha_1 \leftarrow 1, \dots, K$  do
    | for  $\alpha_2 \leftarrow \alpha_1 + 1, \dots, K$  do
      | agent  $ag_{\alpha_2}$  selects relevant actions from  $pg_{\alpha_1}^L$  and introduces them into its
      | plangraph  $pg_{\alpha_2}^L$ ;
      | agent  $ag_{\alpha_1}$  selects relevant actions from  $pg_{\alpha_2}^L$  and introduces them into its
      | plangraph  $pg_{\alpha_1}^L$ ;
    | end
  end
   $L \leftarrow L + 1$ ;
end

```

3.2.3 Individual Plan Extraction

In the *individual plan extraction* phase, each agent extracts plan(s) from its plangraph. In the centralized Graphplan algorithm, this is done by compilation of the problem into a CSP and solved by a CSP solver. Each result from the solver is then one resulting plan as Kambhampati showed in [8].

3.2.4 Coordination

In the last *coordination* phase, before an agent ag_α generates an individual plan, it includes *requirement constraints* and *commitment constraints*—induced by the other agents’ plans—into its individual plan extraction CSP problem.

The *requirement constraints* are couples (a, i) which describe that action a has to be performed in a level i . The function $req(\pi_{1,\dots,\alpha})$ denotes a set of requirement constraints induced by the current partial plan $\pi_{1,\dots,\alpha}$ of the agent ag_α . A partial plan $\pi_{1,\dots,\alpha}$ in this phase contains finished parts from agents ag_1, \dots, ag_α and future actions required by the agent ag_α (and possibly from previous agents) for the following agents $ag_{\alpha+1}, \dots, ag_K$.

The *commitment constraints* are again couples (a, i) denoting that no action b , which is in *mutex* with action a , can be performed in level i . Function $com(\pi_{1,\dots,\alpha})$ denotes a set of the commitment constraints induced by the current partial plan $\pi_{1,\dots,\alpha}$.

A couple $c = (com(\pi_{1,\dots,\alpha}), req(\pi_{1,\dots,\alpha}))$, besides representing the partial plan $\pi_{1,\dots,\alpha}$ in form of action commitments and requirements, describes which agents have already contributed to the plan $\pi_{1,\dots,\alpha}$ with their individual plans—the performers of the actions in $com(\pi_{1,\dots,\alpha})$ —and which agents have not—the performers of the actions in $req(\pi_{1,\dots,\alpha})$ but not in $com(\pi_{1,\dots,\alpha})$. Such common partial plan in the form of requirement couple c is passed from one agent to another. Each agent extends it with its individual plan and possibly new requirements for following agents. After c passes all the agents, it contains a global plan consisting of individual plans of all the agents.

3.2.5 Example

Assume the agents are ordered ag_1, ag_2, \dots, ag_K , as illustrated in Figure 3.1, and that they have already generated their plangraphs, so phases individual plan extraction and coordination are to be approached. It starts with an agent ag_1 which generates its first plan π_1 —lower index indicates the owner of the individual plan, here it is the agent ag_1 . The agent computes commitment and

requirement constraints, denoted as $c_1 = (com(\pi_1), req(\pi_1))$ and sends them to agent ag_2 . Agent ag_2 includes constraints c_1 into its CSP problem, as additional constraints and generates plan π_2 , which is compatible with plan π_1 . Next, ag_2 computes new constraints $c_{1,2} = (com(\pi_1) \cup com(\pi_2), req(\pi_1) \cup req(\pi_2)) = (com(\pi_{1,2}), req(\pi_{1,2}))$, and passes them to agent ag_3 . Generally, when agent ag_α receives constraints $c_{1,\dots,\alpha-1} = (com(\pi_{1,\dots,\alpha-1}), req(\pi_{1,\dots,\alpha-1}))$, it includes it into its CSP problem and generates its individual plan π_α . Then agent ag_α computes new constraints $c_{1,\dots,\alpha} = (com(\pi_{1,\dots,\alpha-1}) \cup com(\pi_\alpha), req(\pi_{1,\dots,\alpha-1}) \cup req(\pi_\alpha)) = (com(\pi_{1,\dots,\alpha}), req(\pi_{1,\dots,\alpha}))$, and passes them to agent $ag_{\alpha+1}$, and so on. In Figure 3.1, the search would actually end with generating ag_K 's plan π_{ag_K} , which illustrates the ideal way of the solution, since no agent had to generate a variance of its plan more than once to fulfill all the requirements from previous agents. If agent ag_3 could not generate plan π_3 , algorithm would backtrack to agent ag_2 , who would generate an alternative plan π'_2 , as shown in the figure. If all agents ran out of plans, algorithm returns to the phases *expansion* and *planning graph merging*, and whole search repeats again.

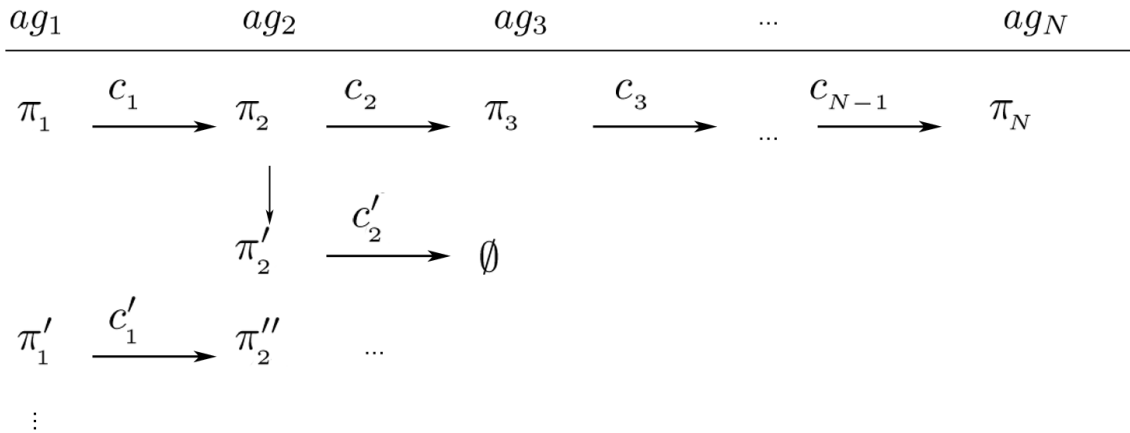


Figure 3.1 DPGM plan search process.

3.3 Optimizations

In [14], the author introduces theoretically the algorithm, yet no experiments were carried out to verify its efficiency. Besides the implementation of the algorithm in Java programming language, some implementation improvements to speed up the algorithm are included.

3.3.1 Constraint cache

To reduce the search tree, as illustrated in Figure 3.1, each agent memorizes the constraints it passes to the next agents. If the constraints caused that one of the following agents is not able to generate any plan, the requesting agents do not require such constraints for the next agents any more.

For an instance depicted in Figure 3.1, at point when agent ag_2 generates plan π_2 together with its constraints $c_{1,2}$, it memorizes the constraints $c_{1,2}$ before passing them to ag_3 . Let us assume the constraints caused ag_3 unable to generate any individual plan depicted as \emptyset . The agent ag_2 memorizes this information and introduces new constraint into its CSP assignment that will eliminate generating of such plans in future, that restricts the agent ag_3 as $c_{1,2}$ did.

For an instance, $c_{1,2} = (\{\}, \{(a_3, 1), (a_4, 2)\})$, where $(a_3, 1)$ restricts ag_3 to perform action a_3 in level 1 and $(a_4, 2)$ restricts ag_4 to perform action a_4 in level 2. Since ag_3 could not find any plan that satisfies restriction $(a_3, 1)$, agent ag_2 should not generate such plans. Thus, ag_2 introduces new constraint into its CSP assignment that will eliminate plans having action a_3 in level 1. This will cause agent ag_2 to eliminate generating of the plans that agent ag_3 cannot satisfy.

3.3.2 Ordering of agents

Ordering of the agents turned out to be crucial for the DPGM algorithm to work efficiently. It is necessary to separate agents that have an individual goal from those that have no goal, but whose

cooperation might be required somewhere during the plan. Let \mathcal{A}_g be a set of all agents having an individual goal and let \mathcal{A}_s be a set of all agents without their own goals, so that $\mathcal{A}_g \cup \mathcal{A}_s = \mathcal{A}$ and sets \mathcal{A}_g and \mathcal{A}_s are disjunctive. As the plan search progresses the actual agents' ordering dynamically changes. The ordering starts with a random agent from the set \mathcal{A}_g , lets call him ag_1 . Note, that the set \mathcal{A}_g cannot be empty at this point, if it was, we have no goal, therefore the problem would be unsolvable. Afterwards, a plan π_1 is generated together with the constraints c_1 . Next agent, lets call him ag_2 , is selected by looking which cooperation is required in $req(\pi_1)$ from the c_1 . If there are more such agents, then those in \mathcal{A}_g are prioritized over the agents from \mathcal{A}_s . After generating new π_2 and $c_{1,2}$, the hole process is repeated. If at any point $c_{1,\dots,\alpha}$ has no requirements on the other agents, although \mathcal{A}_g or \mathcal{A}_s are not empty yet, it means that there exists a goal reaching plan without cooperation of the agents remaining in \mathcal{A}_g and \mathcal{A}_s sets. Notice that this may happen even for \mathcal{A}_g , although I said that these are agents with goals. This may happen in the case, if a goal proposition can be reached by more than one agent and an agent that is not in \mathcal{A}_g have reached it. Algorithm 5 shows the selection of the next agent. As an input it receives the last agent's plan π_α , constraints $c_{1,\dots,\alpha-1}$ of agents $ag_1, \dots, ag_{\alpha-1}$, and the two disjunctive sets of agents: set of agents with goal \mathcal{A}_g and set of agents without goals \mathcal{A}_s . First it computes new constraint $c_{1,\dots,\alpha}$ and extracts $\mathcal{A}_{required}$, the set of agents whose cooperation is required there in $req(\pi_{1,\dots,\alpha-1})$. Then the algorithm selects one of the agents from this set, prioritizing the agents from \mathcal{A}_g over the agents in \mathcal{A}_s .

3.3.3 Removing unnecessary actions

Since I used pure PDDL parser to read the domain and the problem from the files, we could end up with useless actions. For example, in the simplest LOGISTICS problem these actions are following: (drive car1 place1 place1) or (fly plane1 airport1 airport1). The reason that these actions are useless is because they do not change the state in any way. These actions can be generalized as

Algorithm 5: Select Next Agent

```

input :  $\pi_\alpha$ ; // last agent's plan
         $c_{1,\dots,\alpha-1} = (com(\pi_{1,\dots,\alpha-1}), req(\pi_{1,\dots,\alpha-1}))$ ; // requirements that agent
         $ag_\alpha$  received
         $\mathcal{A}_g$ ; // agents with goals
         $\mathcal{A}_s$ ; // agents without goals
output:  $ag_{selected}$ ; // next agent, to whom  $c_{1,\dots,\alpha}$  will be sent

 $c_\alpha \leftarrow (com(\pi_\alpha), req(\pi_\alpha))$ ;
 $c_{1,\dots,\alpha} \leftarrow (com(\pi_{1,\dots,\alpha-1}) \cup com(\pi_\alpha), req(\pi_{1,\dots,\alpha-1}) \cup req(\pi_\alpha))$ ;
 $\mathcal{A}_{required} \leftarrow$  agents whose cooperation is required in  $req(\pi_{1,\dots,\alpha})$  but are not in
 $com(\pi_{1,\dots,\alpha})$ ;
 $\mathcal{A}_{contributed} \leftarrow$  agents who contributed to  $com(\pi_{1,\dots,\alpha})$ ;
if  $\mathcal{A}_{required} \cap (\mathcal{A}_g \setminus \mathcal{A}_{contributed}) \neq \emptyset$  then
    |  $ag_{selected} \leftarrow$  select arbitrary agent from  $\mathcal{A}_{required} \cap (\mathcal{A}_g \setminus \mathcal{A}_{contributed}) \neq \emptyset$ ;
    | return  $ag_{selected}$ ;
else
    |  $ag_{selected} \leftarrow$  select arbitrary agent from  $\mathcal{A}_{required} \cap (\mathcal{A}_s \setminus \mathcal{A}_{contributed}) \neq \emptyset$ ;
    | return  $ag_{selected}$ ;
end

```

$a = \langle pre(a), add(a), del(a) \rangle$, where $add(a) = del(a)$. Moreover, actions as STRIPS defines them, have a requirement $add(a) \cap del(a) = \emptyset$. But a pure PDDL parser cannot hold this requirement while parsing. Hence, these actions had to be removed in the algorithm.

3.4 Implementation

Implementation of the DPGM algorithm was carried out in Java programming language on NetBeans IDE 7.1.2. Flowchart of the whole algorithm is illustrated in Figure 3.2. Blue rectangles represent the phases of the procedures, while white rectangles comment under what conditions their paths are selected in the run. The complex procedure phases are explained as follows (the self-explanatory procedures are skipped):

PDDL parser takes on its input the domain and problem files and returns the problem description

$\mathcal{P} = \langle P, \mathcal{A}, I, G \rangle$, where P is set of all propositions, \mathcal{A} is set of agents, $I \subseteq P$ is set of initial

propositions and $G \subseteq P$ is set of goal propositions. This process is done externally by calling the PDDL4J parser.

Remove agent's unnecessary actions - the optimization described in the section 3.3.3.

Goal decomposition phase - the exact implementation of the Algorithm 3.

Planning graph expansion phase - expands the planning graph of every agent by one level as described in section 3.2.2; if the fixed point is reached, then algorithm terminates.

Planning graph merging phase - in this phase each agent shares with the other agents the actions of its planning graph's last level; others select the *threats* and *promotions* and include them into their planning graphs.

Select next agent ag_i who has not contributed to the plan - this selection follows the rules described in the section Ordering of Agents 3.3.2.

Include constraint c into its CSP - in this phase agent ag_i builds the CSP problem from its planning graph, and additionally includes there the constraint $c = (req(\pi), com(\pi))$; it is implementation of the coordination section 3.2.4.

Extract a plan - in this phase the algorithm calls an external CSP solver (in my implementation Minion Solver or Choco Solver) with the CSP problem; the returned result converts into the plan

Compute new constraint c and memorize it - new constraint is computed as described in the Example 3.2.5 and memorization is the Constraint Cache optimization described in 3.3.1.

Backtrack to the previous agent - sets the previous agent as the current agent to extract new plan

Check the plan's validity by progression - in this verification phase algorithm runs the plan to check if it actually reaches the goal without mutexes.

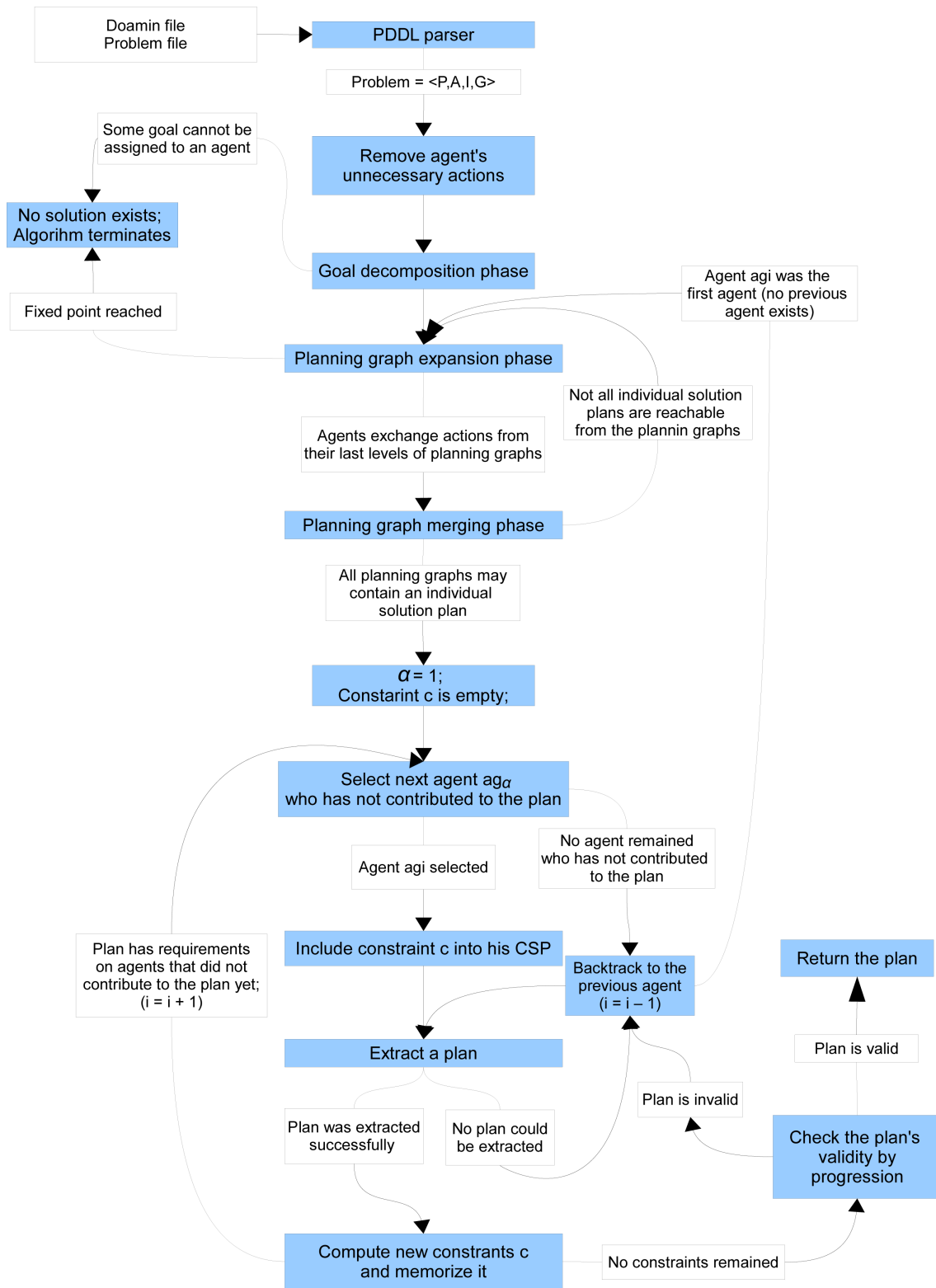


Figure 3.2 Flowchart of the DPGM implementation.

Chapter 4

PDDL Adaptation for Multiagent Planning

4.1 PDDL

Planning Domain Definition Language (PDDL) is language for describing the planning problems. It was inspired by previous languages and formalisms such as ADL, SIPE-2, Prodigy-4.0, UMCP, Unpop and UCPOP [1]. Each planning problem consists of the *domain* description, the world in which the problem occurs and the *problem* description, that states what is the actual task.

In the *domain* descriptions the mandatory and most important definitions are following:

Types defines types of the objects, that occur in the problem. Types are defined hierarchically with the basic built-in types `object` and `number`. The typical types could be `locations`, `packages`, `vehicles`, `cities`, `trucks`, `air planes`, `agents`, etc.

Predicates represent a features, a property or any true/false statement about one or more objects.

Predicates are in parenthesis, where first string is the name of the predicate, followed by input variables. Typical predicates are `(at ?car - vehicle ?loc - location)`, `(at ?a - (either package vehicle location) ?c - city)`, etc. Variables starts with `?` and must be followed by dash and its type. Construction (either `<type1>` `<type2>` ...)

denotes, that variable may be any of the listed types.

Actions define actions that can be performed in this world. Every action $a = \langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$ is defined by preconditions $\text{pre}(a)$, the positive effects (or add-effects) $\text{add}(a)$ and negative effects (or del-effects) $\text{del}(a)$. Notion of every actions in PDDL is following (without the angle brackets):

```
(:action_<action-name>
  :parameters_(<?<v_1>-<type-of-v_1>...<?<v_n>-<type-of-v_n>)
  :preconditions_(and_(<pre_1>)<pre_2>...<pre_n>))
  :effect_(and_(<add_1>)...<add_n>)<del_1>...<del_n>))
)
```

The *problem* description defines:

- which *domain* description it uses,
- the objects used in the problem,
- the initial state by listing the preconditions that hold there, and
- the goal state by listing the preconditions that must hold there.

4.1.1 PDDL Example

Lets show a simple example of the logistics planning problem. This simple logistics planning problem consists of locations, airports, cities, trucks and airplanes. Each location and airport is assigned to a city. Trucks can drive only between the locations (including the airport) in the same city and airplanes can fly only between the airports between the cities. The problem consists of transporting one or more packages from their initial location to a goal location. Typical plan is that car has to drive the package from initial location to the airport within one city, where the package is passed to the airplane, who will fly to the airport in a different city, and another truck takes over the package again, and drives it to the desired location. The domain description is following:

```

(define (domain logistics)
  (:requirements :strips :typing)
  (:types package location vehicle - object
           truck airplane - vehicle
           city airport - location)

  (:predicates
   (at ?vehicle-or-package - (either vehicle package) ?location - location)
   (in ?package - package ?vehicle - vehicle)
   (in-city ?loc-or-truck - (either location truck) ?city - city))

  (:action load-truck
   :parameters (?obj - package ?truck - truck ?loc - location)
   :precondition (and (at ?truck ?loc) (at ?obj ?loc))
   :effect (and (not (at ?obj ?loc)) (in ?obj ?truck)))

  (:action load-airplane
   :parameters (?obj - package ?airplane - airplane ?loc - airport)
   :precondition (and (at ?obj ?loc) (at ?airplane ?loc))
   :effect (and (not (at ?obj ?loc)) (in ?obj ?airplane)))

  (:action unload-truck
   :parameters (?obj - package ?truck - truck ?loc - location)
   :precondition (and (at ?truck ?loc) (in ?obj ?truck))
   :effect (and (not (in ?obj ?truck)) (at ?obj ?loc)))

  (:action unload-airplane
   :parameters (?obj - package ?airplane - airplane ?loc - airport)
   :precondition (and (in ?obj ?airplane) (at ?airplane ?loc))
   :effect (and (not (in ?obj ?airplane)) (at ?obj ?loc)))

  (:action drive-truck
   :parameters (?tr - truck ?loc-from - location ?loc-to - location ?city - city)
   :precondition (and (at ?tr ?loc-from) (in-city ?loc-from ?city)
                     (in-city ?loc-to ?city))
   :effect (and (not (at ?tr ?loc-from)) (at ?tr ?loc-to)))

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

```

The problem description could be following:

```

(define (problem pb3)
  (:domain logistics)
  (:requirements :strips :typing)
  (:objects
   package1 package2 package3 - package

```

```

airplane1 airplane2 - airplane
pgh bos la - city
pgh-truck bos-truck la-truck - truck
pgh-po bos-po la-po - location
pgh-central bos-central la-central - location
pgh-airport - (either airport location)
bos-airport - (either airport location)
la-airport - (either airport location)
(:init
  (in-city pgh-po pgh)
  (in-city pgh-airport pgh)
  (in-city pgh-central pgh)
  (in-city bos-po bos)
  (in-city bos-airport bos)
  (in-city bos-central bos)
  (in-city la-po la)
  (in-city la-airport la)
  (in-city la-central la)
  (at package1 pgh-po)
  (at package2 pgh-po)
  (at package3 pgh-po)
  (at airplane1 pgh-airport)
  (at airplane2 pgh-airport)
  (at bos-truck bos-po)
  (at pgh-truck pgh-po)
  (at la-truck la-po))
(:goal (and
  (at package1 bos-po)
  (at package2 la-po)
  (at package3 bos-po))))

```

Figure 5.1 illustrates the problem’s initial state. Green arrows depicts the tasks, that trucks and airplanes have to accomplish, i.e., move the packages from pgh-po to bos-po and la-po.

4.1.2 PDDL Adaptation for Multiagent Planning

PDDL language does not support the ability to define agents and actions they can perform as it is required in multiagent planning. There is an extension for PDDL 3.1 that enables it [9], however, the parser itself is not available for downloading as far as I know. Thus, I came up with a method how to alter the domain and problem description as well as the algorithm, that will enable the multiagent planning problem assignments. It requires two steps: (i) assigning an action to an

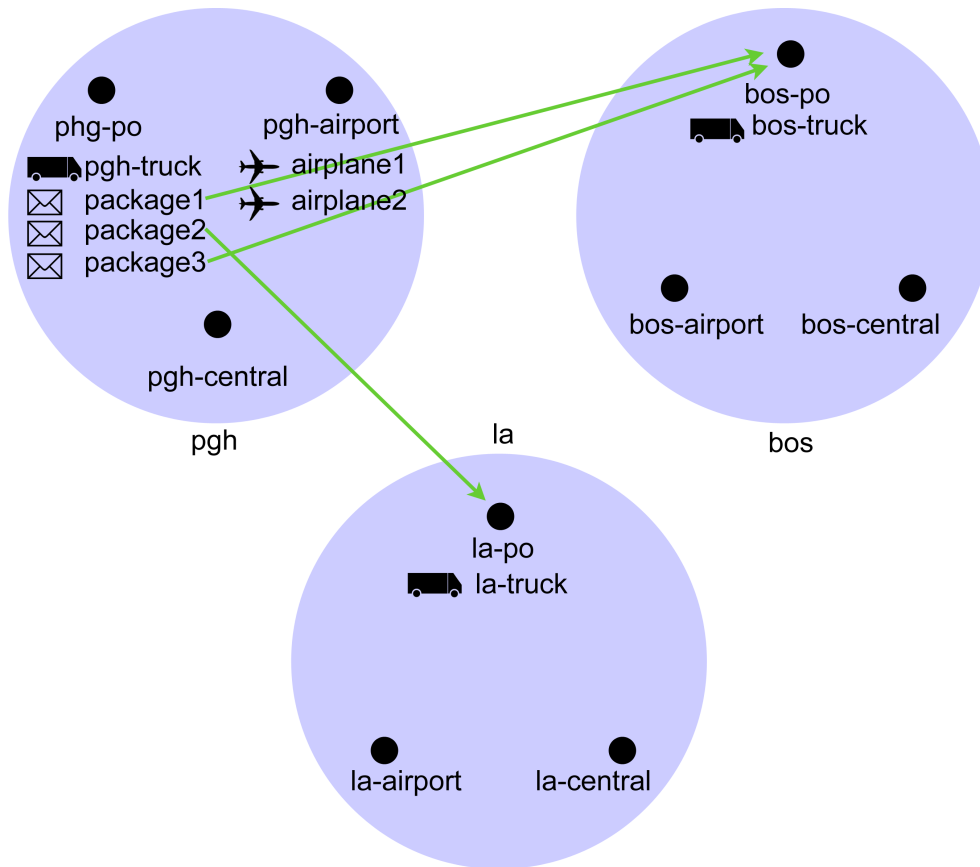


Figure 4.1 An example of a simple logistics problem.

agent and (ii) limiting an action as necessary. Assigning an action to an agent is an easy part. In the domain description an action can be assigned to a specific agent by adding string `agent<#>-` at the beginning of the name of that action. The algorithm will interpret it in such a way that this action can be performed only by the agent `ag#`. If name of the action does not start with `agent<#>-` it means that all agents can perform that action. In the second part the action has to be limited if necessary. Lets say agent `ag1` is a truck that can drive only within a specific city, thus, agent's action `agent1-drive` has to be disallowed to drive within other cities or between the cities. This is done by introducing new type of object that is common for all locations where agent can drive. It does not have to represent a real object, but will be included in all actions of that

agent. Consequently, the object has to be properly defined in the problem description.

Following example will clarify the procedure. Assume I want to alter a simplified version of the logistics problem from the section 4.1.1 PDDL Example for the multiagent planner. Simplified version will contain only two cities, pgh and bos with their trucks and only one plane airplane1. Each mean of transport will represent an agent. Agent ag_1 is bos-truck that can drive, load and unload only within the bos city, similarly agent ag_2 is the pgh-truck that can drive, load and unload only within the pgh city and agent ag_3 , the airplane1, can fly, load and unload only in locations pgh-airport and bos-airport. The domain description of this planning problem is following:

```
(define (domain logistics)
  (:requirements :strips :typing)
  (:types agent package location vehicle city - object
    truckbos truckpgh airplane1 - vehicle
    truck1 truck2 plane1 - agent
    airport locationbos locationpgh - location)
  (:predicates (at ?vehicle-or-package - (either vehicle package) ?location - location)
    (in ?package - package ?vehicle - vehicle)
    (in-city ?locOrTruck - (either location truckbos truckpgh) ?city - city))
  (:action agent1-load-truck
    :parameters (?ag - truck1 ?obj - package ?truck - truckbos ?loc - locationbos)
    :precondition (and (at ?truck ?loc)(at ?obj ?loc))
    :effect (and (not (at ?obj ?loc))(in ?obj ?truck)))
  (:action agent1-unload-truck
    :parameters (?ag - truck1 ?obj - package ?truck - truckbos ?loc - locationbos)
    :precondition (and (at ?truck ?loc)(in ?obj ?truck))
    :effect (and (not (in ?obj ?truck))(at ?obj ?loc)))
  (:action agent1-drive-truck
    :parameters (?ag - truck1 ?tr - truckbos ?from - locationbos
      ?to - locationbos ?city - city)
    :precondition (and (at ?tr ?from) (in-city ?from ?city) (in-city ?to ?city))
    :effect (and (not (at ?tr ?from)) (at ?tr ?to)))
  (:action agent2-load-truck
    :parameters (?ag - truck2 ?obj - package ?truck - truckpgh ?loc - locationpgh)
    :precondition (and (at ?truck ?loc)(at ?obj ?loc))
    :effect (and (not (at ?obj ?loc))(in ?obj ?truck)))
  (:action agent2-unload-truck
    :parameters (?ag - truck2 ?obj - package ?truck - truckpgh ?loc - locationpgh)
    :precondition (and (at ?truck ?loc)(in ?obj ?truck))
    :effect (and (not (in ?obj ?truck))(at ?obj ?loc)))
  (:action agent2-drive-truck
    :parameters (?ag - truck2 ?truck - truckpgh ?from - locationpgh
```

```

        ?to - locationpgh ?city - city)
:precondition (and (at ?truck ?from)(in-city ?from ?city)(in-city ?to ?city))
:effect (and (not (at ?truck ?from))(at ?truck ?to)))
(:action agent3-load-airplane
:parameters (?ag - plane1 ?obj - package ?airplane - airplane1 ?loc - airport)
:precondition (and (at ?obj ?loc)(at ?airplane ?loc))
:effect (and (not (at ?obj ?loc))(in ?obj ?airplane)))
(:action agent3-unload-airplane
:parameters (?ag - plane1 ?obj - package ?airplane - airplane1 ?loc - airport)
:precondition (and (in ?obj ?airplane)(at ?airplane ?loc))
:effect (and (not (in ?obj ?airplane))(at ?obj ?loc)))
(:action agent3-fly-airplane
:parameters (?ag - plane1 ?airplane - airplane1 ?from - airport ?to - airport)
:precondition (at ?airplane ?from)
:effect (and (not (at ?airplane ?from))(at ?airplane ?to))))

```

The common object for all locations in ny is the locationny. And since locationny is included in every action that the truck ny-truck can perform, its actions will be limited for the ny city. Similarly for the other agents. The problem description then is as follows:

```

(define (problem logistics-e2)
  (:domain logistics)
  (:requirements :strips :typing)
  (:objects
    package1 - PACKAGE
    airplane1 - AIRPLANE
    pgh - CITY
    bos - CITY
    bos-truck - TRUCKNY
    pgh-truck - TRUCKPGH
    pgh-po - LOCATIONPGH
    bos-po - LOCATIONNY
    pgh-airport - (either LOCATIONPGH AIRPORT)
    bos-airport - (either LOCATIONNY AIRPORT)
    ag1 - truck1
    ag2 - plane1
    ag3 - truck2)
  (:init (in-city pgh-po pgh)
    (in-city pgh-airport pgh)
    (in-city bos-po bos)
    (in-city bos-airport bos)
    (at package1 pgh-po)
    (at airplane1 pgh-airport)
    (at pgh-truck pgh-po)
    (at bos-truck bos-po))
  (:goal (at package1 ny-po)))

```


Chapter 5

Experiments

5.1 Domains

The experiments were carried out on five different domain, three of which were taken from the International Planning Competition benchmarks adapted for multiagent planning: ROVERS, LOGISTICS, SATELLITES. The additional two are: LINEAR LOGISTICS and DECONFLICTION. Each domain was tested on several problems with various numbers of agents. In the following sections I will describe each domain [11] and each problem from that domain that were used in the experiments.

5.1.1 Rovers

This planning problem is inspired by NASA Mars Exploration Rovers, where each rover has a mission to collect specific data about the environment on Mars and communicate them back to the lander. The STRIPS version of this planning problem consists of multiple rovers that represent agents. Each rover is equipped with tools for soil analysis, rock analysis and imaging, and a storage. An analysis or an image has to be taken from specific waypoints, to which rovers have to transport

themselves through the other visible waypoints. After rover collects all required data about the objectives, it communicates them to the lander. Only one rover can communicate at a time, which makes the situation even more difficult. Every agent can perform following actions: `navigate`, `sample-soil`, `sample-rock`, `drop`, `calibrate`, `take-image`, `prepare-to-communicate-soil-data`, `prepare-to-communicate-rock-data`, `prepare-to-communicate-image-data` and `communicate-all-data`.

In the problem `rover-a2` there are two rovers, 8 waypoints and 4 objectives; in `rover-a3` there are three agents, 12 waypoints and 6 objectives, and in `rover-a4` there are four rovers, 15 waypoints and 8 objectives.

5.1.2 Satellites

This problem is inspired—as the name indicates—by the satellites, that collect spectrographic and thermographic images of the planet they orbit. Satellites take images under different directions using different instruments set on one of the multiple modes. The problem is to plan the most efficient way to cover the observation given the satellite capabilities. Satellites can perform following actions: `turn-to` - turns to a new direction; `switch-on` and `switch-off` - switches on and off the instrument; `calibrate` - calibrates chosen instrument to a specific direction; and `take-image` - takes an image with an instrument under certain mode over particular direction.

The `satellite-a6` problem contains 6 satellites and 12 directions, the `satellite-a8` problem contains 8 satellites and 16 directions, and problem `satellite-a10` contains 10 satellites and 20 directions.

5.1.3 Logistics

As logistics problem was discussed generally in 4.1.2 PDDL Adaptation for Multiagent Planning, in this section only specific planning problem used in experiments will be described.

The `log-a4` contains two cities: `pgh` and `ny`, both of which have two locations, a `po` locations

and an `airport` locations. Both cities have one truck that can drive between the two locations within the city, and there are two air-planes that can fly between the airports. The goal is to deliver two packages, one package from `pgh-po` to `ny-po`, and the other one from `ny-po` to `pgh-po`. The `log-a6` contains four cities: `pgh`, `ny`, `def` and `abc`, each of which has two locations, a `po` locations and an `airport` locations. Each city has one truck that can drive between the two locations, and there are two air-planes in total. The goal is to deliver two packages, one from `pgh-po` to the `ny-po` and one from `abc-po` to the `def-po`.

5.1.4 Linear Logistics

In this problem the locations are situated in a chained-wise manner one after another, and a package has to be transported from the location on one end of the chain to the location on the another end of the chain. Situation is complicated by the fact, that every truck can drive only between neighbouring locations in this chain, thus, package has to be passed over from one truck to another in every location.

Problems differ in the numbers of the locations and a corresponding number of the trucks, which is one less than the number of locations. Problem `log-lin-a6` consists of 6 trucks and 7 locations, problem `log-lin-a8` of 8 trucks and 9 locations, problem `log-lin-a10` of 10 trucks and 11 locations and finally, `log-lin-a15` of 15 trucks and 16 locations. Problems are depicted in 5.1.

5.1.5 Deconfliction

In this problems the robots, who represent the agents, initially are located on one of the squares in a three times three grid. Its task is to switch its positions with the position of an another robot in non-collision way.

The problem `deconf-a2` consists of two robot that have to switch their positions, as illustrated on the left in 5.2. In the problem `deconf-a3` three robots have to switch positions with the neighbouring

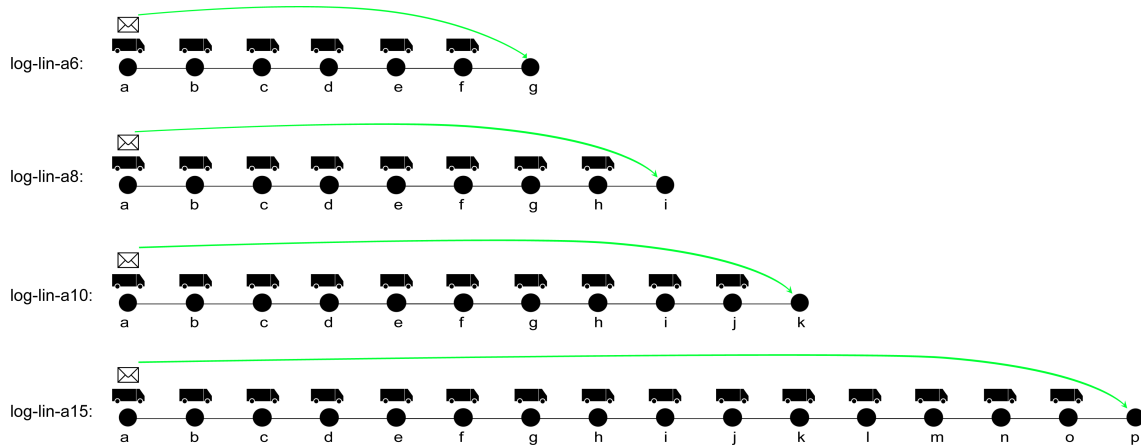


Figure 5.1 Used linear logistics problems in experiments.

robot, as illustrated on the right in 5.2.

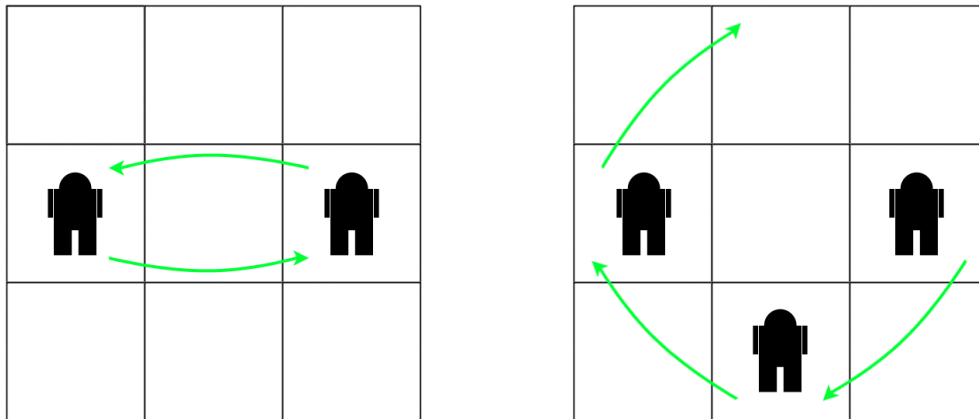


Figure 5.2 Used deconfliction problems in experiments.

5.2 Experiment settings

All experiments were run on 8-core processor at 3.6GHz with 2.5GB limit on memory and 10 minutes time limit. I used time and communicated bytes as metrics for the comparison of the algorithms. Each problem was run 10 times and time averages were calculated.

domain-agents	M	M sdf	M sac	M sdf sac	M srf	Choco	comm.
rover-a2	8.9s	13.3s	7.3s	7.7s	3.8s	11.7s	69kB
rover-a3	6.6s	6.0s	11.9s	11.6s	20.3s	17.4s	234kB
rover-a4	–	–	–	–	–	76.1s	234kB
log-a4	0.9s	0.4s	0.4s	0.3	0.3s	1.2s	34kB
log-a6	0.7s	0.7s	0.7s	0.7	0.6s	1.3s	136kB
log-lin-a6	0.5s	0.5s	0.5s	0.5	0.5s	0.3s	167kB
log-lin-a8	0.7s	0.7	0.8	0.7	0.7s	0.5s	417kB
log-lin-a10	0.9s	0.9s	1.0s	1.0s	0.9s	0.7s	849kB
log-lin-a15	1.6s	1.6s	2.0s	2.0s	1.6s	1.8s	2.9MB
deconf-a2	–	6.4s	–	3.2s	1.3s	(OoM)	18kB
deconf-a3	0.2s	–	0.3s	199.6s	0.2s	0.1s	13kB
deconf-a4	–	–	–	–	–	(OoM)	–
satellite-a6	1.6s	1.4s	1.3s	1.4	1.5s	4.6s	266kB
satellite-a8	5.0s	4.4s	4.5s	4.5s	4.3s	24.8s	793kB
satellite-a10	14.3s	13.2s	13.8s	14.0s	12.7s	101s	1.8MB

Table 5.1 Comparison of CSP solvers used in DPGM. The dash – means that the time limit was exceeded and OoM that memory was exceeded.

5.2.1 Comparison of Used CSP Solvers in DPGM

As DPGM algorithm uses CSP solver for the local plan extraction, I had to decide which solver would serve the best. Two CSP solvers were tested: Choco CSP Solver¹ and Minion CSP Solver². Choco solver was tested with its basic setting, while Minion was tested over several settings: (M)—the default setting; (M sdf)—with smallest-domain-first variable order; (M sac)—with SAC preprocessing; (M srf sac)—combination of the previous two settings, and (M srf)—smallest-ratio-first order. Table 5.1 shows the times DPGM took to solve the problems using certain CSP solvers and settings. Although Minion solver showed to be sometimes unstable and did not return any result over the longer period of time, which is represented in the table as a dash – it was faster, than Choco solver. Another Minion’s disadvantage is that if the problem is unsolvable, it is inefficiently detected, since it has to go through all the possibilities in the search space. Last column (comm.)

¹<http://www.emn.fr/z-info/choco-solver/>

²<http://minion.sourceforge.net/>

in Table 5.1 shows the communicated bytes among the agents. As the CSP solver is used only for local extraction of a plan, the number are the same for all the solvers. In the deconf-a3 problem, although the number of agents is higher than in deconf-a2, the results are better. This is caused by the particular problem instance, where the agents in the a2 case has to pass by each other, and therefore the solution is found not before 4th level, however, in a3 the agents only rotates and therefore the solution is found in 2nd level.

5.2.2 Comparison of the Multiagent Planners

domain-agents	DPGM	DisCSP+Plan.	MAD-A*
rover-a2	3.8s/69kB	1.4s/0.8kB	22.4s/52kB
rover-a3	20.3s/234kB	7.9s/1.7kB	230s/2.5MB
rover-a4	–	62.3s/3.1kB	–
log-a4	0.3s/34kB	0.6/15kB	0.8s/77kB
log-a6	0.6s/136kB	38.5/7.1MB	2.0s/320kB
log-lin-a6	0.5s/167kB	–	1.7s/87kB
log-lin-a8	0.7s/417kB	–	4.7s/254kB
log-lin-a10	0.9s/849kB	–	15.4s/589kB
log-lin-a15	1.6s/2.9MB	–	217s/4.2MB
deconf-a2	1.3s/18kB	N/A	0.9s/15.3kB
deconf-a3	0.2s/13kB	N/A	1.2s/187kB
deconf-a4	–	N/A	3.8s/2.1MB
satellite-a6	1.5s/266kB	4.4/6.5kB	7.4s/270kB
satellite-a8	4.3s/793kB	–	37.5s/964kB
satellite-a10	12.7s/1.8MB	–	189s/2.5MB

Table 5.2 Comparison of DPGM, DisCSP+Planning and MAD-A* with set-additive heuristics. The dash – means that the time or memory limit was exceeded. N/A means the planner did not return a sound plan.

In the final experiment, the DPGM algorithm was compared to other two cited algorithms. Table 5.2 shows results. As the srf setting of Minion showed the best results—especially because of its ability to solve most of the presented problems—I chose it for comparison with the other

algorithms.

The results show that DPGM performs well in decoupled domains, which are rather combinatorially easy (LOGISTICS, LINEAR LOGISTICS, and SATELLITES). The DisCSP+Planning is efficient in problems which are combinatorially hard from perspective of individual planning (ROVERS), as the internally used planner is highly efficient FastForward. The used implementation of MAD-A* with set-additive heuristics was most effective in highly coupled domains (DECONFLICTION).

5.3 Discussion

DPGM showed its strength based on efficient factorization of the problems. However problems as DECONFLICTION, which are coupled and require high combinatorial search, DPGM solves rather inefficiently, if at all. The way CSP generated the plans hindered the performance of the algorithm. For instance, the first plan that the agent ag_1 generated in 4th level of the deconf-a4 problem, consisted of its own actions, leading the agent to the goal, additionally the agent generated requirements for the agent ag_2 to prevent future collisions. However, the requirements were unreasonable: instead of requiring one action that would suffice for agent ag_1 , it built a whole plan for the ag_2 . And since ag_1 did not know the ag_2 's goal, the plan was usually invalid. I tried to avoid this, by stating to minimize requirements put on other agents in the CSP Solver. This approach helped to lower the number of constraints; however, the solution of such CSP became combinatorially more complex and therefore did not bring much improvement in efficiency. The DPGM efficiency depends very much on the CSP solver that it uses. In table 5.1 it illustrates the ROVER-A4 problem, where Minion solver failed to find a result, however Choco solver found it in a comparable time as DisCSP+Planning from the table 5.2. On the other hand, Minion only with certain settings was able to solve the DECONF-A2 problem very fast, while other settings and the Choco solver failed to solve it at all. Additionally, the result shows that Minion with sdf and sac

solves the problem DECONF-A3 in 199.6s, while other settings (except the M sdf) solve it under 0.4s. This number 199.6s is due to the more runs, where sometimes Minion solved it fast, but the other times it was unstable and took more then 270s to solve it. In my opinion, if good CSP solver is found and adjusted to run ina desired way, much better results could be achieved. Deeper study of CSP selection, its adjustment and other phenomenons remains for the future work.

Chapter 6

Conclusion

This work focused on three distinctive algorithms for solving the distributive multiagent planning problems. Two of which, the DisCSP+Planning and MA-A* algorithms, were already implemented and experimentally tested, while the third, DPGM algorithm, was studied in details and implemented. Moreover, I introduced a few optimizations that improved the algorithm. The algorithm was experimentally tested over five different problem domains, that first had to be adapted for the multiagent purposes. Experimental results show the DPGM algorithm to be very effective in problems where agents are not tightly coupled, however less effective in tightly coupled problems.

Appendix A

List of used symbols

In planning in general:

$p \in P$ — is a propositions from P , the set of propositions.

$s \in \mathbb{S}$ — is a state in search space. Each state s is represented as a set of propositions from P . So $\mathbb{S} = \mathbb{P}(P)$, is the powerset of P .

A — is set of action, where each action $a \in A$ is a tripe $a = \langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$ and $\text{pre}(a), \text{add}(a), \text{del}(a) \subseteq P$.

app — is an application function $S \times A \rightarrow S$. For a given state and action it return a new state.

$\Sigma = (\mathbb{S}, A, \text{app})$ — represents the *planning domain*.

\mathcal{P} — represents the *planning problem*. Generally $\mathcal{P} = (\Sigma, s_0, G)$. But for planning problems for STRIPS-like syntax $\mathcal{P} = (P, I, G, A)$ is used. For the multiagent planning $\mathcal{P} = (P, \mathcal{A}, I, G)$.

$G \subseteq P$ — is a set of goal propositions.

$I \subseteq P, s_0 \in \mathbb{S}$ — is a set of propositions that hold in the initial state.

π — is an individual plan. In totally ordered plan $\pi = \langle a_0, \dots, a_N \rangle, a_i \in A$, and in partially ordered plan $\pi = \langle A_0, \dots, A_L \rangle, A_i \subseteq A$.

In multiagent planning:

L — is the length of the plan.

π_α — is an individual plan of agent ag_α .

Π — is a global plan that contains actions for all agents.

K — is the number of agents in multiagent planning domain.

$\mathcal{A} = \{ag_\alpha\}_{\alpha=1}^K$ — is set of agents.

A_α — is set of actions that agent ag_α can perform.

$noop_p$ — is an noop, or maintenance action, for propositions $p \in P$ if $noop_p = \langle \{p\}, \{p\}, \{\} \rangle$.

$O_i \subseteq A$ — is an i-th action level in plangraph

$S_i \subseteq P$ — is an i-th proposition level in graphplan

μ_{O_i} — is set of mutexes between actions in i-th level of the plangraph.

μ_{S_i} — is set of mutexes between actions in i-th level of the plangraph.

G_α — is individual goals of agent ag_α .

$pg_\alpha = \langle pg_\alpha^1, \dots, pg_\alpha^L \rangle$ — is agent ag_α 's plangraph and pg_α^i is i-th level of pg_α plangraph.

$\pi_{1, \dots, \alpha}$ — is union of plans from agents ag_1, \dots, ag_α in respective levels.

(a, i) — is a commitment or requirement constraint that restricts action a to be performed in i-th level.

$com(\pi_{1, \dots, \alpha})$ — is set of commitment constraints of agents ag_1, \dots, ag_α .

$req(\pi_{1, \dots, \alpha})$ — is set of requirement constraints from agents ag_1, \dots, ag_α to agents $ag_{\alpha+1}, \dots, ag_K$.

$\mathcal{A}_g \subseteq \mathcal{A}$ — is set of agents that have non-empty set of individual goals ($G_\alpha = \emptyset$)

$\mathcal{A}_s \subseteq \mathcal{A}$ — is a set of agents that have at least one individual goal ($G_\alpha \neq \emptyset$)

Appendix B

Content of the CD

CD

- | - DPGM/ (*NetBean project of the DPGM algorithm*)
- | - DPGMRunner/ (*NetBean project of framework that runs DPGM over all problems*)
- | - domains/
 - | - rovers/ (*rover problems*)
 - | - deconfliction/ (*deconfliction problems*)
 - | - lin-logistics/ (*lin-logistics problems*)
 - | - logistics/ (*logistics problem*)
 - | - satellites/ (*satellites problems*)
- | - karelDurkota.pdf (*this Diploma Thesis*)
- | - choco-solver-2.1.5.jar (*used Choco solver*)
- | - minion-0.15-windows.tar.gz (*used Minion solver*)
- | - pddl4j.jar (*used PDDL parser*)

Bibliography

- [1] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. Pddl the planning domain definition language. 1998.
- [2] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. Technical report, DTIC Document, 1995.
- [3] Minh Binh Do and Subbarao Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into csp. *Artificial Intelligence*, 132(2):151–182, 2001.
- [4] NASA Facts. Mars science laboratory. *National Aeronautics and Space Administration Jet Propulsion Laboratory, California Institute of Technology*, 2006.
- [5] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann, 2004.
- [6] J Hoffmann and B Nebel. The ff planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.(JAIR)*, 14:253–302, 2001.
- [7] Mark Iwen and Amol Dattatraya Mali. Distributed graphplan. In *Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence, ICTAI '02*, pages 138–, Washington, DC, USA, 2002. IEEE Computer Society.
- [8] Subbarao Kambhampati. Planning graph as a (dynamic) csp: exploiting ebl, ddb and other csp search techniques in graphplan. *Journal of Artificial Intelligence Research*, 12(1):1–34, 2000.
- [9] Daniel L Kovacs. A multi-agent extension of pddl3. *WS-IPC 2012*, page 19, 2012.
- [10] Ugur Kuter and Dana Nau. Using domain-configurable search control for probabilistic planning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 1169. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.
- [11] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *J. Artif. Intell. Res. (JAIR)*, 20:1–59, 2003.

- [12] Raz Nissim and Ronen I Brafman. Multi-agent a* for parallel and distributed systems. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1265–1266. International Foundation for Autonomous Agents and Multiagent Systems, 2012.
- [13] Raz Nissim, Ronen I Brafman, and Carmel Domshlak. A general, fully distributed multi-agent planning algorithm. *Proceedings of AAMAS'10*, 2010.
- [14] Damien Pellier. Distributed planning through graph merging. In *ICAART (2)*, pages 128–134, 2010.
- [15] Ralph B Roncoli and Jan M Ludwinski. Mission design overview for the mars exploration rover mission. In *2002 Astrodynamics Specialist Conference*, 2002.
- [16] Stuart Jonathan Russell, Peter Norvig, Ernest Davis, Stuart Jonathan Russell, and Stuart Jonathan Russell. *Artificial intelligence: a modern approach*. Prentice hall Upper Saddle River, NJ, 2010.