Multi-Agent Planning with Planning Graph

The Duy Bui[1] and Wojciech Jamroga[1,2]
[1]Parlevink Group, University of Twente, Netherlands
[2]Institute of Mathematics, University of Gdansk, Poland
{theduy,jamroga}@cs.utwente.nl

Abstract

In this paper, we consider planning for multi-agents situations in STRIPS-like domains with planning graph. Three possible relationships between agents' goals are considered in order to evaluate plans: the agents may be collaborative, adversarial or indifferent entities. We propose algorithms to deal with each situation. The collaborative situations can be easily dealt with the original Graphplan algorithm by redefining the domain in a proper way. Forward-chaining and backward chaining algorithms are discussed to find infallible plans in adversarial situations. In case such plans cannot be found, the agent can still attempt to find a plan for achieving some part of the goals. A forward-chaining algorithm is also proposed to find plans for agents with independent goals.

Keywords: multiagent systems, planning in STRIPS-like domains, planning graph, collaborative planning, adversarial planning best defense.

1 Introduction

In this paper planning for multi-agent situations in STRIPS-like domains is considered. A STRIPS-like planning problem [5] consists of: a set of parameterized operators (action templates), a set of objects to whom the operators can be applied to generate an action, a set of propositions to define the initial state of the environment, and a set of goals (propositions that the planning agent wants to have eventually satisfied). Single-agent planning algorithm in STRIPS-like domains, based on the idea of building and analyzing a so called *Planning Graph*, was proposed in [3]. Although the complexity of STRIPS planning is at least PSPACE-hard, the authors reported very good experimental results on a variety of problems. This approach was revised and extended to suit probabilistic domains in [4]. We now extend the planner in [3] to deal with various situations of planning in multi-agents environments.

1.1 Planning through Planning Graph Analysis

The Graphplan planner [3] builds up a *planning graph* that contains two kinds of nodes: *proposition nodes* and *action nodes*, organized in alternating levels. The first level is a proposition level and refers to the initial facts. Edges in a planning graph (precondition-edges, add-effect-edges and delete-effect-edges) represent relations between actions and propositions. Graphplan also attempts to find *mutually exclusive* actions and facts that cannot happen in parallel. After generating a planning graph of a depth t in a forward-chaining way, the planner looks for a plan using a backward-chaining, level-by-level strategy. If it fails, Graphplan can either detect that no valid plan exists for the problem or generate the $t+1$th level of the graph and extend the search. Figure 1 shows example of operators and a derived planning graph of depth 2.

1.2 Planning in an Environment Inhabited by Multiple Agents

Switching to multi-agent environments poses a number of important problems. First of all, the relationship between the agents' goals must be determined in order to evaluate plans. There are roughly 3 possibilities [7]:

- collaborative agents who have exactly the same goals and can collaborate to bring about them,
- adversarial agents: the goal of the other agents is *not to let* 'our' agent fulfill his goals,
- indifferent agents (or rather agents with independent policies) that cover all the middle ground between both extremes.

```
(define (operator move)
   :parameters ((rocket ?r) (place ?from) (place ?to))
   :precondition (:and (:neq ?from ?to) (at ?r ?from) (has-fuel ?r))
   :effect (:and (at ?r ?to) (:not (at ?r ?from)) (:not (has-fuel ?r))))

(define (operator load)
   :parameters ((rocket ?r) (place ?p) (cargo ?c))
   :precondition (:and (at ?r ?p) (at ?c ?p))
   :effect (:and (:not (at ?c ?p)) (in ?c ?r)))

(define (operator unload)
   :parameters ((rocket ?r) (place ?p) (cargo ?c))
   :precondition (:and (at ?r ?p) (in ?c ?r))
   :effect (:and (:not (in ?c ?r)) (at ?c ?p)))
```

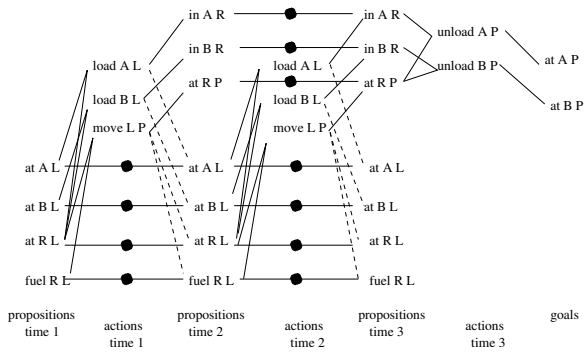

Figure 1: The operators and the planning graph for the *Simple Rocket Domain* [3]. 'No-operation' actions (no-ops) are marked with dots.

The collaborative situation is the easiest to deal with and can be tackled almost directly using the Graphplan algorithm from [3]. One must be careful, however, to define the domain (i.e. the operators, objects etc.) in a proper way so that the resulting Planning Graph and the resulting plan indeed refer to the problem we wanted to solve. Some suggestions about this can be found in section 2.

The adversarial case (discussed in section 3) lies at the heart of Game Theory – in the form of *zero-sum games* theory – with a tradition going back at least to the works by von Neumann and Mogenstern in the 1940s [9]. This reflects a bias of the classical Game Theory: we want our agent to play safe; we want him to be protected against the worst line of events. It is worth noting that such a perspective is not restricted to traditional games with numerical utility function only. Some of the recent work on multi-agent planning through model checking logical formulae derives from the same tradition: a goal is achievable only when it can be enforced for every possible response from the rest of the agents; otherwise no plan can be generated [2, 8]. Two planning algorithms (a forward-chaining and a backward-chaining one) are analyzed for the adversarial planning case.[1]

Since a situation when no plan is generated is not acceptable from the planning agent's perspective, we propose an extension of this approach in section 4. The agent's goals can be satisfied to various degrees, according to a linear utility function. The best plan can be found with a forward-chaining minimaxing algorithm.

The implications of the classical Game Theory standpoint go beyond the zero-sum games. Even for the games where the players can have independent utility functions, most widely accepted rational decision making criteria assume (implicitly or explicitly) that the agent must look for a plan against the worst possible line of events – since such a model refers to the lower bound of his capabilities. Two approaches to the planning problem for agents with independent goals are proposed in section 5. First, we may follow the Game Theory perspective completely (independent search). Alternatively, the agent can look for some coalition with other agents and involve collaborative planning to the greatest possible extent.

A number of simplifying assumptions were adopted within this paper:

- the agents have complete knowledge of the situation (no uncertainty, no probabilistic beliefs),
- the agents have complete knowledge of the outcomes of every action,
- the outcome of every action is deterministic (there are no probabilistic actions or actions with uncertain outcome),
- the planning graph forms a synchronous turn-based structure: time is discrete and at every time point only one agent proceeds with an action or actions (the agents take *turns*),
- the goals of every agent are public.

2 Collaborative Multi-Agent Planning

When all the agents can fully cooperate to bring about some set of goals, the whole coalition can be treated as a single agent trying to search for a single-agent plan. Therefore the original Graphplan planner can be used to find it. However,

[1] a different approach to the adversarial planning seems to be proposed in [6]. The author suggests that a planning graph for multi-agent turn-based situations can be rebuilt to include conditional actions instead of other agents' nodes; then we can run some standard conditional planner on it. According to the available materials, the work is still in progress.

(a)

```
(define (operator move)
  :parameters ((agent ?a) (rocket ?r) (place ?from) (place ?to))
  :precondition (:and (controls ?a ?r) (:neq ?from ?to) (at ?r ?from) (has-fuel ?r))
  :effect (:and (at ?r ?to) (:not (at ?r ?from)) (:not (has-fuel ?r))))

(define (operator take-control)
  :parameters ((agent ?a) (rocket ?r))
  :precondition (uncontrolled ?r)
  :effect (:and (:not (uncontrolled ?r)) (controls ?a ?r)))

(define (operator release)
  :parameters ((agent ?a) (rocket ?r))
  :precondition (controls ?a ?r)
  :effect (:and (:not (controls ?a ?r)) (uncontrolled ?r)))
```

(b)

```
(define (operator load)
  :parameters (agent ?a) (cargo ?c) (rocket ?r) (place ?p))
  :preconditions (:and (at ?a ?p) (acting ?a) (at ?r ?p)(at ?c ?p))
  :effect (:and (in ?c ?r) (:not (at ?c ?p))))

(define (operator unload)
  :parameters (agent ?a) (cargo ?c) (rocket ?r) (place ?p))
  :preconditions (:and (in ?a ?r) (acting ?a) (at ?r ?p) (in ?c ?r))
  :effect (:and (at ?c ?p) (:not (in ?c ?r))))

(define (operator move)
  :parameters ((agent ?a) (rocket ?r) (place ?from) (place ?to))
  :precondition (:and (acting ?a) (in ?a ?r) (:neq ?from ?to) (at ?r ?from) (has-fuel ?r))
  :effect (:and (at ?r ?to) (:not (at ?r ?from)) (:not (has-fuel ?r))))
```

Figure 2: Modifying the domain to allow the control over actions in multi-agents situations: (a) by defining the control over objects; (b) by defining which agent is acting.

the definition of the domain (and the resulting planning graph) should reflect the multi-agent nature of the problem: the agent that performs an action should be explicitly represented in its parameters. This is necessary because in a multi-agent plan the tasks must be distributed among the coalition members. It has also a very welcome implication: the same activity cannot be performed by two different agents at the same time; the actions (and the objects involved) should be controlled by a single agent: for instance, agents A and B cannot move the same rocket at the same moment, even if they want to move it to the same place. Actions `(move A R Warsaw Hanoi)` and `(move B R Warsaw Hanoi)` are mutually exclusive, because they delete each other's preconditions. Thus, only one agent can control the rocket at a time. Unfortunately, it holds only for the most dynamic operations, namely the operations that 'consume' their own preconditions. If the only effect of the operation `fill-tank` is that the rocket has some fuel in its tank (and *not* that there is less fuel than before at the petrol station – the station must have infinite resources of fuel!), then `(fill-tank A R Warsaw)` and `(fill-tank B R Warsaw)` are not mutually exclusive. In a similar way, two agents may simultaneously drill a hole in the tank and make it empty.

The easiest way to provide the agents with a complete control over actions is to introduce an additional predicate used in the preconditions of operations concerned. An example domain definition that makes use of propositions (`controls Agent Object`) is shown on figure 2a. Note that with a slightly changed definition the agents can be designed to control actions instead of objects. A more radical way of dealing with the problem is proposed in the example on figure 2b: a concept of "acting agent" is added by the algorithm at each turn. A predicate like `(acting ?a)` can be used to represent situations such as: "agent a_1 can do action α but agent a_2 cannot". For example, while a_1 is at home, a_2 is at work. Apparently, if the car stands in front of the house, only agent a_1 can drive the car but not agent a_2.

We will assume that the structure of the problem is strictly turn-based throughout the rest of the paper, i.e. that every proposition level of the corresponding planning graph is added exactly one fact of type `(acting ?a)`.

It is worth emphasizing that the agents share their knowledge and cooperate completely in this kind of planning. Other kinds of (bounded) cooperation can also be considered, like planning for hierarchically organized military units [1], for instance.

3 Planning against Best Defense in Adversarial Games

Much of Game Theory is devoted to decision making in a situation when all the other agents are powerful enemies. Searching for such infallible plans in STRIPS-like domains is discussed in this section. In order to avoid confusion, we will use the term *goal* to denote one of the propositions that the agent wants to make true, *goals* to denote a complete set of goals he wants to bring about (i.e. a set of facts from the same level). A state s at time t is a set of propositions

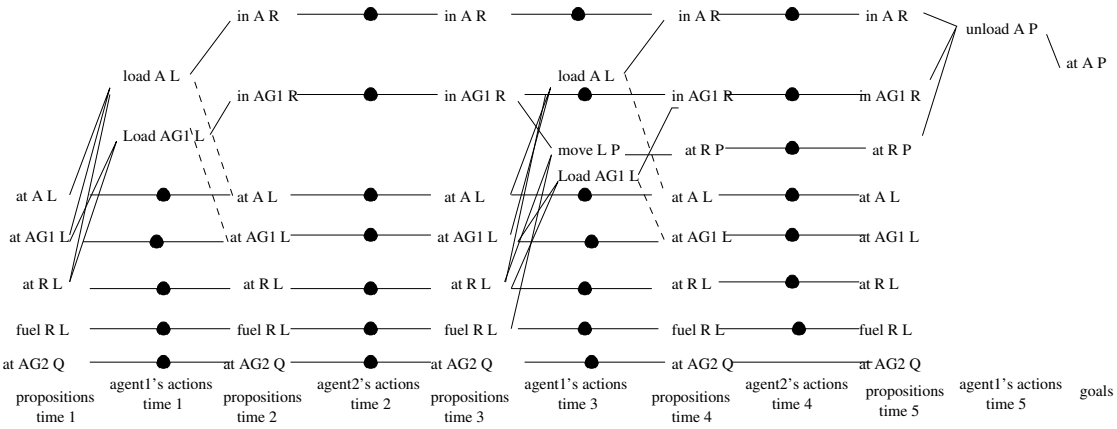

Figure 3: A trivial example of adversarial planning when the other agent cannot do anything to prevent the planning agent from reaching his goal

which are true at that time. A move m is a set of actions from the same action level, in which no two actions are labeled mutually exclusive. A *subplan* is a tuple $\langle t, s, m\rangle$ so that the move m is specified to be executed from state s at time t; Finally, a *plan* is a set of subplans for all the relevant states. Note that a plan in the multi-agent non-collaborative setting is substantially more complex than a single-agent plan, exactly in the same way as for the probabilistic planning problem [4]: since the agent has no complete control over the course of action, subplans starting from various states at the same time must be considered. Thus, the size of a complete plan in the multi-agent setting is exponential in time, while for a single agent it is only linear.

Searching for such a plan can be done with forward-chaining and backward-chaining algorithms. We first define a 'good' state at time t as a set of preconditions so that there exists a subplan $\langle t, preconditions, move\rangle$. The forward-chaining algorithm checks whether the initial state is a 'good' state – the plan is established during this checking. The backward-chaining algorithm, on the other hand, finds and stores all possible 'good' states back from the goals. If the initial state is inside the set of 'good' states then a plan is found. We now describe and discuss the forward-chaining and backward-chaining algorithms.

The forward-chaining algorithm

Let T be the number of levels in the planning graph. A state s at time t is determined as a 'good' one as follows:

- if $t = T$ then s is 'good' if it contains all the final goals.
- if $t < T$ and it is the planning agent's turn then s is 'good' if there exists a legal move m from s leads to a 'good' state in time $t + 1$. Remember $\langle t, s, m\rangle$ as a *subplan*.
- if $t < T$ and it is the other agent's turn then s is 'good' if every legal move from state s leads to a 'good' state in time $t + 1$.

The advantage of the forward-chaining algorithm is that it can terminate without searching the whole subtree in some cases. First, if *any* action leading to a 'good' state has been found at the planning agent's turn then the search is already successful and can be terminated immediately. Second, checking actions from a state at the other agent's turn can be terminated immediately (with failure) when even one action does not lead to a good state. These premature terminations depend greatly on the order of propositions in the state being checked.

The backward-chaining algorithm

The backward-chaining level-by-level planning algorithm, presented below, is based directly on the Graphplan idea in an attempt to benefit from the mutual exclusion labeling. For every level t a list of 'good' states G_t is derived from G_{t+1}.

We start describing the algorithm with some observations. First, it must be the planning agent's turn at $T - 1$; if it is the other agent's turn and a plan exists, then a shorter plan can be found. Therefore, with the assumption that the planning agent moves first and there are just two agents taking turns one by one,[2] we consider only even T's $(0, 2, 4..)$. Second,

[2]note that this can be done without any loss of generality. First, even when there is more than just one opponent, we can consider the enemies as a single player in the adversarial case (therefore assuming full cooperation between them). Second, if any agent is allowed to make several subsequent moves in the game, we can divide the nodes in the graph into particular players' partitions, and run the search partition-by-partition instead of level-by-level.

the search cannot be restricted to minimal good subgoals only, because it may happen that a good set $s_1 \in G_t$ has a 'bad' superset $s_2 \notin G_t$ (s_2 may contain preconditions for a dangerous action, which is inexecutable from s_1 because some of the preconditions are lacking). Worse still, bad s_2 may have a good superset s_3. Thus, neither the property of being a good set nor the property of being a bad one is monotonic and we cannot restrict the algorithm to the minimal (or maximal) good set only, unlike in the original planner. Third, a set of propositions representing a state at time t (opponent's turn) can be 'good' only if it is 'good' at time $t+1$, too – otherwise the opponent can choose 'do nothing' move (containing only no-op actions) and get to a 'bad' state. Therefore the list of candidates at time t is simply G_{t+1}. This observation is not completely true for last move of the opponent since there he has no moves after that. Instead, the requirement for a set of facts to be a 'good' state at this time is that it must be superset of any 'good' state at the next moment.

Based on the above observations, G_t is created as follows:

- if $t = T$ then G_t contains only one state that is the goals.
- if $t < T$ and it is the planning agent's turn:

  for every state s' in G_{t+1}

  Derive all the candidate states s from s' in the same manner as original Graphplan does.

  if $t = T - 1$ **then** add s to G_t.

  if $t < T - 1$ **then** verify that s lead to exact s' but not any superset of s'. If so, add s to G_t.
- if $t < T$ and it is the other agent's turn:

  if $t = T - 2$ **then**

  for every possible states s at time t

  if s is superset of at least one s' in G_{t+1} **then**

  if every legal move from state s leads to one of states in G_{t+1}

  then add s to G_t.

  if $t < T - 2$ **then**

  for every states s' in G_{t+1}

  Copy s' to s in time t

  if every legal move from state s leads to one of states in G_{t+1}

  then add s to G_t.

The speed of the backward-chaining algorithm depends much on the checking step at time $t = T - 2$. This step is really time-consuming if the number of possible propositions is big. This is the disadvantage when compared to the forward-chaining one. The set of possible good states G_t at any time t can also be further refined by checking if any state s in G_t is reachable with a single-agent plan in t steps. If not, the other agent can just "do nothing" and state s cannot be achieved at time t. The sets of good states at previous time get smaller as $G_t \subseteq G_{t+1}$. The backward-chaining algorithm, for a large T, can be advantageous over the forward-chaining one.

We have extended the rocket domain to design four test domains in order to compare the forward-chaining and the backward-chaining algorithms. The first test domain is simple: "shallow" (the goals can be achieved in a short time) and "narrow" (the number of possible facts at each time t is small). The second test domain is "shallow" and "wide" (the number of possible facts at each time t is large). The third one is similar to the second one except that some "redundant" facts are added, so it's even "wider". The last test domain is a "deep" (the goals can be achieved in a long time) and "narrow" one. The result of running both algorithms on a Pentium II 350MHz computer with 256MB RAM is shown in table 1.

	test1	test2	test3	test4
forward-chaining	1 second	2 seconds	5 seconds	120 seconds
backward-chaining	1 second	12 seconds	$>$ 10 minutes	6 seconds

Table 1: The result of running the forward-chaining and backward-chaining on the four test domains: test1 – "shallow" and "narrow"; test2 – "shallow" and "wide"; test3 – "shallow" and "wider"; test4 – "deep" and "narrow"

From the result in table 1 we can see that the forward-chaining algorithm is very effective on "shallow" problems, while the backward-chaining one performs better on problems with a "narrow" planning graph. The backward-chaining algorithm can still be further improved by removing the redundant facts somehow.

4 Planning to Achieve Partial Goals in Adversarial Situations

Infallible plans have one important drawback: they seldom exist in practice. Thus, an agent looking for a perfect plan in an adversarial situation will fail to do so for many practical problems. Since all the plans that may fail to achieve all the goals are considered equally insufficient, no plan will be chosen. If we assign a value of 1 to success, and 0 to failure, then most plans are equivalent because the set of all valuations contains only two values. In this section we propose a way of enriching the set: a plan that leads to achieving a large part of the goals is considered better than a plan that satisfies only a small portion of them. For instance, an agent trying to deliver two pieces of cargo C_1 and C_2 should be most satisfied when he succeeds to achieve the whole set of goals. If this is not possible, however, he should choose a plan that delivers only one of the pieces rather than a plan that fails to deliver anything.

The idea is very natural: to determine the extent to which the goals are achieved in some state, we can simply count how many of the goals are true in this state. Then the agent can look for a plan that guarantees to achieve the maximal fraction of the goals. The setting can be generalized to allow the agent express some preferences over the goals: every fact f can be assigned a weight (or utility value) w_f; we assume that the utility of making some facts true is the sum of elementary utilities. In other words, the scoring function is linear now.

The partial goals algorithm

Let w_f be a real number that represents the utility of making f true. The utility of a state $s = \{f_1, ..., f_k\}$ is defined as $u(s) = w_{f_1} + ... + w_{f_k}$.[3] The planning agent searches for a plan that maximizes u, while the other agents try to minimize it. The search can be done with a forward-chaining minimaxing algorithm. Let T be again the number of levels in the planning graph. The valuation of a reachable state s at time t is computed as follows:

- if $t = T$ then return $u(s)$.
- if $t < T$ and it is the planning agent's turn:

  Compute valuations $v_1, ..., v_k$ for all the states $s_1, ..., s_k$ at $t+1$ such that for every s_i there is a move m_i providing a transition from s to s_i. Choose s_i with maximal v_i.

  if $v_i > u(s)$ **then** remember $\langle t, s, m_i \rangle$ as a subplan. Return v_i

  else remember $\langle t, s, stop \rangle$ as a subplan. Return $u(s)$.
- if $t < T$ and it is the other agent's turn:

  Compute the valuations $v_1, ..., v_k$ in the same way. Return the minimal v_i.

Note that the above algorithm finds a perfect plan if such a plan exists. Moreover, the forward-chaining algorithm from section 3 is a special case of this one: the utility of a state can be only 1 (for a 'good' state) or 0 (for a 'bad' one), a fact has a weight of 1 if it's one of the goals and 0 otherwise, and the composite utility u is a boolean function: $u = w_{f_1} \wedge ... \wedge w_{f_k}$. Note also that the planning agent here not only determines the time horizon T; he can also decide to finish the interaction (action $stop$) before T if it's beneficial for him.

5 Planning for Agents with Independent Goals

When the agents inhabiting some environment follow their individual goals, and the goals are not adversarial by principle, they can either look for a way of coordinating their plans or simply try to achieve their goals by themselves. The first case is when the agents are willing to communicate and cooperate to search for a plan that suits everybody, even if in some situations the coalition's plan may take more steps than each individual's own plan. We will call this a coalitional approach. If this is not the case – the planning agent must look for a plan on his own, considering all the possible responses from the rest of agents (independent planning). Algorithms for independent and coalitional planning are presented below.

The independent planning algorithm

Let $\Sigma = \{a_1, ..., a_n\}$ be the set of all agents. Like in the previous section, we assume that it is better to achieve some part of the goals than no goals at all. Utility (weight) of a fact is defined for every agent a separately: w_f^a, and the composite utility is linear again: $u^a(s) = \sum_{f \in s} w_f^a$. The valuation of a reachable state s at time t is computed as follows:

- if $t = T$ then return the utilities for all the agents $\langle u^{a_1}(s), ..., u^{a_n}(s) \rangle$.

[3] states in a planning graph are just sets of facts

- if $t < T$ and it is the agent a's turn:

  Compute valuations $v_1, ..., v_k$ for all the states $s_1, ..., s_k$ at $t+1$ such that for every s_i there is a move m_i providing a transition from s to s_i. Note that every valuation is a tuple of values, one per agent: $v_i = \langle v^{a_1}(s_i), ..., v^{a_n}(s_i)\rangle$.

  Choose s_i with maximal valuation for the acting agent: $v^a(s_i)$.

  if a is the planning agent and $v_i^a > u^a(s)$ **then** remember $\langle t, s, m_i\rangle$ as a *subplan*. Return v_i.

  if a is the planning agent and $v_i^a \leq u^a(s)$ **then** remember $\langle t, s, stop\rangle$ as a *subplan*. Return $\langle u^{a_1}(s), ..., u^{a_n}(s)\rangle$.

  else return v_i.

The coalitional planning algorithm

The independent planning algorithm can be easily extended for coalitional planning in multi-agent environments:

- find all the maximal coalitions $\mathscr{C} \subseteq \Sigma$ such that the planning agent $a \in \mathscr{C}$ and the goals within $\mathscr{C}$ are non-conflicting. This can be done by running Graphplan on the set of collective goals of the coalition: $Goals(\mathscr{C}) = \bigcup_{c\in\mathscr{C}} Goals(c)$;
- for every such coalition $\mathscr{C}$:
  - treat the agents from $\mathscr{C}$ as a single agent trying to achieve the collective goals;
  - search for the best plan $\mathscr{P}(\mathscr{C})$ with the **independent planning algorithm**;
- choose $\mathscr{C}$, $\mathscr{P}(\mathscr{C})$ for which $a's$ valuation of the initial state is maximal.

The plan formation in the above algorithms is based on forward-chaining search similar to the algorithm from section 4. The coalition formation phase for coalitional planning, on the other hand, employs the original Graphplan algorithm. Of course, coalition formation is computationally expensive in the general case (when the number of agents is large). However, for two or three agents it is quite feasible since the backward-chaining single-agent Graphplan runs faster than the forward-chaining algorithms for multi-agent planning. Note also that in some cases the situation is trivial: if goals of agents c_i are more restricted than a's goals ($Goals(c_i) \subseteq Goals(a)$ for all i) then $Goals(\{c_1, c_2, ..., a\})$ are non-conflicting from a's perspective (there is no conflict unless $Goals(a)$ are self-conflicting). Moreover, if two propositions from $Goals(\mathscr{C})$ are labeled mutually exclusive at level t then the goals are obviously conflicting in t steps. However, the reverse is not true: if there are no such labels in the set of collective goals, it does not necessarily guarantee that the goals are non-conflicting.

6 Conclusions and Further Research

In this paper, we considered planning for multi-agents situations in STRIPS-like domains with planning graph. Three possible relationships between agents' goals were considered in order to assess plans. The goals may be just the same (collaborative planning), opposite (adversarial planning) or there may be no direct relation between them. We have proposed algorithms to deal with each situation. The collaborative case can be easily dealt with by redefining the domain in a proper way. Forward-chaining and backward chaining algorithms have been discussed to find infallible plans in adversarial situations; in case such plan cannot be found, the agent can still produce a plan to achieve the largest possible fraction of the goals. A forward-chaining algorithm was also proposed to find plans for agents with independent goals.

The algorithms presented in this paper can still be improved to obtain higher performance. It may be also interesting to investigate how heuristics can be used to speed up the algorithms. Moreover, we would like to explore planning in multi-agent situations in which some of the assumptions from section 1.2 don't hold any more, i.e. when the agents may have incomplete information, the goals aren't public and/or the domain is probabilistic, as well as for different best defense models.

References

[1] E. Alonso and D. Kudenko. Logic-based multi-agent systems for conflict simulations. In *UKMAS*, 2000.

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49:672–713, 2002. Updated, improved, and extended text.

[3] A.L. Blum and M.L. Furst. Fast planning through graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[4] A.L. Blum and J.C. Langford. Probabilistic planning in the Graphplan framework. In *Proceedings of ECP'99*, pages 319–332, 1999.

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

[6] W. Lue. Adversarial planning in multiagent systems based on Graphplan. 2002. Abstract and slides from a presentation at Dagstuhl Seminar 02481: Programming Multi Agent Systems based on Logic. Available at `http://www.cs.man.ac.uk/~zhangy/dagstuhl/`.

[7] S. Sen and G. Weiss. Learning in multiagent systems. In Weiss G., editor, *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*, pages 259–298. MIT Press: Cambridge, Mass, 1999.

[8] W. van der Hoek and M. Wooldridge. Tractable multiagent planning for epistemic goals. In *Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*, 2002.

[9] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behaviour*. Princeton University Press: Princeton, NJ, 1944.