A Planning Graph Heuristic for Forward-Chaining Adversarial Planning

Pascal Bercher and Robert Mattmüller*
Institut für Informatik,
Albert-Ludwigs-Universiät Freiburg

July 18, 2008

Technical Report 238

Abstract

In contrast to classical planning, in adversarial planning, the planning agent has to face an adversary trying to prevent him from reaching his goals. In this report, we investigate a forward-chaining approach to adversarial planning based on the AO* algorithm. The exploration of the underlying AND/OR graph is guided by a heuristic evaluation function, inspired by the relaxed planning graph heuristic used in the FF planner. Unlike FF, our heuristic uses an adversarial planning graph with distinct proposition and action layers for the protagonist and antagonist. First results suggest that in certain planning domains, our approach yields results competitive with the state of the art.

1 Introduction

In many planning problems, the environment in which the agent acts is not static. The exogenous dynamics can be caused by "nature" or by one or more other agents sharing the same environment. Other agents can behave neutrally (simply following their own independent agenda or otherwise acting unpredictably), adversarially, or cooperatively with respect to the protagonist's goals. Here, we focus on adversarial problems. We assume complete observability, i.e., a plan will be a mapping from physical states to applicable actions. A usual approach to conditional (adversarial) planning is planning as Model Checking [CPRT03], whereas planning as heuristic search [BG01] tends to yield best results for static, deterministic problems. Both approaches are also used in General Game Playing [GLP05]. Related work includes the dynamic programming approach by Hansen and Zilberstein [HZ98], multi-agent graphplan-based planning [BJ03], and, for partially observable problems, heuristic search in the belief space as implemented in the POND planner by Bryce et al. [BKS06].

This report is structured as follows. In the Section 2, we introduce the basic definitions. Section 3 shows the variant of the AO* algorithm we used to

*{bercherp,mattmuel}@informatik.uni-freiburg.de

solve adversarial planning problems. Section 4 introduces the adversarial FF heuristic, and in Section 5, we provide some experimental results.

2 Definitions

In this section, we provide the basic definitions we need for the rest of this paper.

Definition 1 (Reachability Game)**.** A *game structure* $G = \langle S, O, \Gamma_1, \Gamma_2, \delta, p \rangle$ consists of the following components:

- Two players 1 and 2, also called *protagonist* and *antagonist*, respectively.
- A finite set S of *states*.
- A finite set O of *operators*.
- Two functions $\Gamma_i : S \to 2^O$, $i \in \{1, 2\}$, assigning to each state $s \in S$ the set of operators available to player i in s. We require that for all $s \in S$, $\Gamma_i(s) = \emptyset$ for exactly one $i \in \{1, 2\}$. $\Gamma(s)$ denotes the union $\Gamma_1(s) \cup \Gamma_2(s)$.
- A *partial transition function* $\delta : S \times O \to S$ which assigns to each state $s \in S$ and each operator $\gamma \in \Gamma(s)$ the state s' resulting from the application of γ in s.
- A *player function* $p : S \to \{1, 2\}$ which assigns to each state s the player who is to move, i.e., $p(s) = i$ iff $\Gamma_i(s) \neq \emptyset$ for $i \in \{1, 2\}$ and $s \in S$. We require that both players move in turn, i.e., $p(s) \neq p(s')$ for all $s, s' \in S$ with $s' = \delta(s, \gamma)$ for some $\gamma \in \Gamma(s)$.

A *reachability game* $\mathcal{G}$ is a triple $\langle G, R, s_0 \rangle$ consisting of a game structure G, an initial state $s_0 \in S$ and a non-empty set of goal states $R \subseteq S$.

The protagonist tries to reach a goal state in a finite number of steps, whereas the antagonist tries to prevent him from doing so. A *winning strategy* for the protagonist is a function mapping states in which he is to move to applicable operators, such that, against each possible strategy of the antagonist, a goal state is reached in a finite number of steps.

A *history* σ is a sequence of states $s_0, \dots, s_n$ where s_0 is the initial state of $\mathcal{G}$ and for all $0 \leq j < n$ with $n \in \mathbb{N} \cup \{\infty\}$, there is a $\gamma \in \Gamma(s_j)$ such that $s_{j+1} = \delta(s_j, \gamma)$. Let H_i^+ be the set of all finite histories in which player i has to move, i.e., $H_i^+ := \{ \sigma = (s_0, \dots, s_n) \mid \sigma \text{ is a finite history and } p(s_n) = i \}$. A *strategy* π_i for player $i \in \{1, 2\}$ is a (partial) mapping $H_i^+ \to O$ which assigns to history $\sigma = (s_0, \dots, s_n) \in H_i^+$ a possible move $\gamma \in \Gamma_i(s_n)$. A *memoryless strategy* for player $i \in \{1, 2\}$ is a strategy π_i, such that for histories $\sigma s, \tau s \in H_i^+$, $\pi_i(\sigma s) = \pi_i(\tau s)$, i.e., the move prescribed by a memoryless strategy is only dependent on the current state and not on the history leading there. Thus, we will simply write $\pi_i(s)$ instead of $\pi_i(\sigma s)$ for history $\sigma s \in H_i^+$.

Since in each state $s \in S$, the player i who is to move has a non-empty set of moves Γ_i, it follows that each history that is induced by a pair of two strategies of the protagonist and antagonist is infinite. We call the history induced by two memoryless strategies π_1 and π_2 the outcome $O(\pi_1, \pi_2) = s_0, s_1, s_2, \dots,$ defined as the unique history where s_0 is the initial state and

$s_{j+1} = \delta(s_j, \pi_{p(s_j)}(s_j))$ for all $j \in \mathbb{N} \cup \{0\}$. The set of states $\{s_0, s_1, s_2, \dots\}$ in this history is denoted by *states*$(O(\pi_1, \pi_2))$. The strategy π_1 wins against strategy π_2 iff *states*$(O(\pi_1, \pi_2)) \cap R \neq \emptyset$. The protagonist wins the reachability game $\mathcal{G}$ if and only if he has a strategy π_1, which wins against all strategies π_2 of the antagonist. To such a strategy we will refer as a *winning strategy*. It is sufficient to restrict our attention to memoryless strategies, since in every reachability game one of the players wins memorylessly [GTW02, follows from Theorem 2.14].

In order to find such winning strategies, we will perform an AO* search [Nil98] on the AND/OR graph induced by the reachability game.

Definition 2 (AND/OR Graph)**.** An *AND/OR graph* is a triple $\langle V_{AND}, V_{OR}, E\rangle$, consisting of:

- A finite set of *AND vertices* V_{AND}.
- A finite set of *OR vertices* V_{OR}.
- A set $E \subseteq (V_{AND} \times V_{OR}) \cup (V_{OR} \times V_{AND})$ of *edges*.

V_{AND} and V_{OR} have to be disjoint, i.e., $V_{AND} \cap V_{OR} = \emptyset$.

The AND/OR graph induced by the reachability game, called the *game graph*, is defined as follows:

$$
\begin{aligned}
V_{OR} &= \{\ s \in S \mid p(s) = 1\ \} \\
V_{AND} &= \{\ s \in S \mid p(s) = 2\ \} \\
E &= \{\ (s, s') \mid s \in V_{OR},\ s' \in V_{AND} \text{ and there is a } \gamma \in \Gamma_1(s) \text{ with } \delta(s, \gamma) = s'\} \\
&\quad \cup \{\ (s, s') \mid s \in V_{AND},\ s' \in V_{OR} \text{ and there is a } \gamma \in \Gamma_2(s) \text{ with } \delta(s, \gamma) = s'\}
\end{aligned}
$$

Although the game graph is finite, it is in general too large to be represented explicitly, so we also consider *partial game graphs*, which are connected subgraphs of a game graph containing the initial state, such that where all vertices are either unexpanded, i.e., there are no outgoing edges to child vertices, or fully expanded, i.e., all successor states are represented.

3 AO* Algorithm

The AO* algorithm [Nil98] is an informed search algorithm that searches for solutions in AND/OR graphs. We first give an informal account of the algorithm before presenting pseudocode describing it in more detail.

The AO* algorithm iteratively generates a game graph starting with the partial game graph only consisting of the initial state. Then, the algorithm alternates between expansion and update steps as follows. In an expansion step, the algorithm first determines a most promising subgraph of the current partial game graph by starting at the initial state s_0 and tracing down all *marked* edges going out of OR vertices and *all* edges going out of AND vertices. The algorithm maintains the invariant that for all expanded OR vertices, always exactly one outgoing edge is marked. Once the most promising subgraph has been determined, one of its unexpanded leaf vertices is expanded, where the choice of the vertex can be made dependent on an evaluation function applied to the leaf vertices.

Expanding a vertex s means adding all its successors s'_i and edges (s, s'_i) to the partial game graph unless already present. Newly added AND vertices are expanded immediately, whereas for newly added OR vertices only an evaluation function is computed and the algorithm enters the update phase.

During an update phase, the *cost estimates* of interior vertices are updated given new cost estimates of their children. The cost estimate $c(s)$ of a leaf vertex is the result of the evaluation function applied to s, whereas the cost estimate of an interior vertex is an aggregation of the cost estimates of its children:

$$\begin{aligned} c(s) &= 1 + \max_{s' \in children(s)} c(s') \qquad \text{if } s \text{ is an AND vertex, and} \\ c(s) &= 1 + \min_{s' \in children(s)} c(s') \qquad \text{if } s \text{ is an OR vertex.} \end{aligned}$$

The constant 1 added at AND and OR vertices represents the unit cost of an operator application. For all expanded OR vertices, during the update phase an outgoing edge to a vertex *minimizing* the cost estimate is marked.

When the cost estimate of a vertex s is updated, this can recursively trigger updates of the cost estimates of the parent vertices of s. Therefore, after each expansion, the new cost estimates are propagated backwards until no more updates are necessary or the update procedure would run into a cycle. Information about whether one of the players possesses a winning strategy in the subgame starting at a given vertex is propagated through the partial game graph along with the cost estimates.

We call a vertex s *proven* if the protagonist has a winning strategy in the subgame starting in s and the relevant part of the winning strategy is completely represented by the current partial game graph. This is the case iff $s \in R$ or if s is an OR vertex (AND vertex) and at least one of its successors is proven (all its successors are proven).

Once the initial state s_0 is proven, the algorithm returns a *solution graph*, i.e., a subgraph of the current partial game graph containing (a) the initial state, (b) for each contained non-goal AND vertex all outgoing arcs and their target vertices, (c) for each contained non-goal OR vertex exactly one outgoing arc and its target vertex which has to represent the action prescribed by the winning strategy, and no further vertices or arcs, such that all leaf vertices are goal states. The solution graph has to be acyclic.

Our implementation of the AO* algorithm differs slightly from the version described in the textbook by Nilsson [Nil98]. Standard AO* does not cope with cycles. While there are no cycles in the solution graphs, there might be some (depending on the problem domain) in the game graph. Those cycles could have the effect that cost estimates are updated infinitely often. Therefore, we need to keep track of which vertices have already been updated during an update run (cf. line 5 of the pseudocode). The set U contains all vertices that still have to be updated, and $C \supseteq U$ contains all vertices that have already been scheduled for updates before. Vertices are only scheduled for an update unless they are contained in C.

In the description of the AO* algorithm, we did not address the issue of how to initialize the cost estimates of leaf vertices. We will make up for this omission in the following section.

Input : Reachability Game $\mathcal{G} = \langle\langle S, O, \Gamma_1, \Gamma_2, \delta, p\rangle, R, s_0\rangle$ with game graph $\mathcal{G}^{A/O} = \langle V_{AND}, V_{OR}, E\rangle$ (given implicitly)

Output: partial game graph $\mathcal{G}'^{A/O} = \langle V'_{AND}, V'_{OR}, E'\rangle$ and (if existent) a solution graph $\mathcal{G}^{*A/O} := \langle V^*_{AND}, V^*_{OR}, E^*\rangle \subseteq \mathcal{G}'^{A/O}$

Create the initial explicit graph $\mathcal{G}'^{A/O} = \langle\emptyset, \{s_0\}, \emptyset\rangle$, only consisting of the initial vertex s_0. Calculate the heuristic estimate $h(s_0)$ and the initial cost estimate $c(s_0) := h(s_0)$. Mark s_0 as proven, if $s_0 \in R$.

while s_0 *not proven and explicit graph not completely expanded* **do**

- Traverse the partial game graph, using only marked edges until no more leaves can be reached. Select from those leaves a vertex v with minimal cost estimate.
- Expand this vertex v by adding all its successors to the set of AND vertices and by adding the corresponding edges. Expand each newly generated AND vertex in the analogous way, unless it is already proven. Compute the heuristic of the new OR vertices, followed by updating the cost estimates and the proof status of all new AND vertices.
- Let $U = C = \{v\}$
- **while** $U \neq \emptyset$ **do**
  - Pick and remove a vertex u from U. Update the cost estimate and the proof status of u. Remove marks of all outgoing edges of u. Then mark an edge leading to a best successor, giving proven vertices precedence over unproven ones.
  - **if** *proof status of* u *changed* **then**
    - add all unproven parents of u to U and to C.
  - **else if** *cost estimate of* u *changed* **then**
    - add all unproven parents of u which are not contained in C to U and to C.
  - **end**
- **end**

end

Algorithm 1: AO* Algorithm

4 Adversarial FF Heuristic

Since we decided to employ a variant of the heuristic used in the FF planning system [HN01], we need to make some assumptions about how the states and operators are encoded. We assume a STRIPS [FN71] encoding with operators divided in two sets controlled by the protagonist and antagonist, respectively. More precisely, there is a set of propositions P over which the set of states $S = 2^P$ is defined. The state $s_0 \subseteq P$ is the initial state and $R \subseteq S$ is the set of goal states. We assume that whenever $r \in R$ and $r' \supseteq r$, also $r' \in R$. Additionally, we assume that in each state the player to move is known. An operator o has the form $\langle \mathit{pre}, \mathit{add}, \mathit{del} \rangle$, where $\mathit{pre} \subseteq P$ is the *precondition* and $\mathit{add}, \mathit{del} \subseteq P$ are the *add* and *delete lists* of o. Each player $i \in \{1, 2\}$ controls a finite set of operators O_i. O_1 and O_2 need not be disjoint. An operator $o \in O_i, i \in \{1, 2\}$, is *applicable* in a state $s \subseteq P$ iff the player to move is i and $\mathit{pre} \subseteq s$. If applied, o leads to the successor state $s' = (s \setminus \mathit{del}) \cup \mathit{add}$, in which player $\bar{i} = 3 - i$ is to move. A state s is a goal state iff $s \in R$.

The *adversarial FF heuristic* is based on the heuristic used in the FF planning system [HN01], to which we will refer as the *standard FF heuristic*. Both use a relaxation of the operators to solve a relaxed form of the planning problem. This is done by ignoring the delete lists of all operators. While in classical planning there is only one agent and thus only one set of (relaxed) operators, the adversarial FF heuristic uses two relaxed sets of operators, one for each player. These sets of relaxed operators is $O_i^+ := \{ \langle \mathit{pre}, \mathit{add}, \emptyset \rangle \mid \langle \mathit{pre}, \mathit{add}, \mathit{del} \rangle \in O_i \}$. We abbreviate $o^+ = \langle \mathit{pre}, \mathit{add}, \emptyset \rangle$ as $(\mathit{pre} \rightarrow \mathit{add})$.

The computation of the adversarial FF heuristic consists of the construction of a relaxed planning graph (with alternating fact and proposition layers for the protagonist and the antagonist), the extraction of a relaxed plan (with relaxed operators partitioned into those selected for execution by the protagonist and those selected for execution by the antagonist), and a final post-processing step in which rules controlled by both players, but selected for execution by a particular one of them may be re-distributed among the players. Based on the re-arranged sets of relaxed operators, the heuristic value is computed, taking into account that both players move in turn.

4.1 Computation of the Adversarial FF Heuristic

Note that the pseudocode implementation assumes that a solution always exists. In the following, we will describe the pseudocode line by line.

Line 1 is the base case and returns zero if the initial state s_0 is a goal state.

Lines 2 through 9 correspond to the forward step of the standard FF heuristic. Starting in the first proposition layer, all operators that are applicable in that layer are executed, leading to a new layer which contains all propositions of the previous layer plus all propositions that are produced by the applicable operators. This procedure continues until no additional operators are applicable or until a goal state is found. In line 8, this forward step differs from the forward step of the standard FF heuristic, since we always select operators controlled by exactly one of two players.

Lines 10 through 27 correspond to the backward step of the standard FF heuristic. We are interested in finding a subset of all operators found during the forward step that is sufficient to reach the goal state r. To that end, we start

Input : For each player $i \in \{1, 2\}$ a set of operators O_i^+, each operator o^+ having the form $(pre \rightarrow add)$, an initial state s_0, and a non-empty set of goal states R.
Output: An estimate of the cost to reach R.

/* Forward step: Generate a goal state r. */
if $s_0 \in R$ **then return** 0
$i = -1$
$S[-1] := s_0$
$O[-1] := \emptyset$
while $r \not\subseteq S[i]$ *for all* $r \in R$ **do**
  /* Go to next layer. */
  $i++$
  /* Calculate all propositions in layer i. */
  $S[i] := S[i-1] \cup \{ \ p \in add \mid (pre \rightarrow add) \in O[i-1] \ \}$
  /* Calculate all operators that are applicable in $S[i]$. */
  $O[i] := \{ \ (pre \rightarrow add) \in O^+_{i\%2+1} \mid pre \subseteq S[i] \ \}$
end

/* Backward step: Find operator sequence to generate r. */
$SO_1 = \emptyset$ /* selected operators of player 1 */
$SO_2 = \emptyset$ /* selected operators of player 2 */
$m := i$ /* the last processed layer */
$G[m] := r$ /* with $r \subseteq S[m]$, the found goal state */
for $j = m - 1$ **to** 0 **do**
  $G[j] := \emptyset$
  $SO[j] := \emptyset$
  /* Select for each proposition that we would like to have in the next layer an operator to produce it (if necessary). */
  foreach $g \in G[j+1]$ **do**
    if $g \in S[j]$ **then**
      $G[j] := G[j] \cup \{g\}$
    else
      pick $o^+ = (pre \rightarrow add) \in O[j]$, such that $g \in add$
      $SO[j] := SO[j] \cup \{ \ o^+ \ \}$
      $G[j] := G[j] \cup pre$
    end
  end
  /* Sort operators selected for execution by player. */
  $SO_{j\%2+1} := SO_{j\%2+1} \cup SO[j]$
end

Algorithm 2: Computation of the Adversarial FF heuristic **(part 1)**

/* Re-distribution step: If possible, shift operators from SO_1 to SO_2 (or vice versa) to ensure that the difference between $|SO_1|$ and $|SO_2|$ is as small as possible. */

/* Let max be the player contributing more operators to a plan. */

28 **if** $|SO_1| > |SO_2|$ **then** $max := 1$ **else** $max := 2$

/* Calculate h on the basis of $n \in \mathbb{N}$, the number of operators that *have* to be played by the max player, since the other player does not control these operators. */

29 $n := |SO_{max}| - |SO_{max} \cap O^{+}_{\overline{max}}|$, where $\overline{max} := 3 - max$

30 $h := \max\{2n, |SO_1| + |SO_2|\}$

Algorithm 2: Computation of the Adversarial FF heuristic **(part 2)**

with the goal state r just found and check for each proposition g in r, whether or not it already exists one layer earlier (line 18). If not, g has been generated in the current layer. Thus, we have to select an operator that produces g (lines 22 and 23). This selection can be based on various strategies. We achieved best results with the strategy that always selects an operator with smallest precondition. So far, the backward step of he adversarial FF heuristic does not differ from the standard FF heuristic. The only difference is line 26, where we collect all operators that are selected for the same player.

Lines 28 through 30 do not have a counterpart in the standard FF heuristic and handle the fact that there are two players moving in turn. They re-distribute the relaxed operators among the players if this is possible. Since the two players take alternating turns, the number of turns until the goal can be reached is at least twice the number of operators that the player who contributes more operators to a plan (called max player) controls. Assume that the second player contributes ten operators to a plan and the first player none. This would lead to a relaxed plan of length twenty, including ten artificial no-ops by the first player. Assume that the ten chosen operators could also be executed by the first player. Then we might as well re-distribute them among the players, giving five to each player, for an overall relaxed plan length of ten. Since player 2 is the adversary of player 1, he will usually try to avoid executing operators that contribute to a plan. Thus our heuristic calculation is optimistic and assumes that the adversary would cooperate.

To compute a heuristic value, in line 28 we define max, the player who contributes more operators to the plan. Next (line 29), we calculate n, which is the number of operators that can only be played by the max player, since he is the only one who controls these operators. Since both players move in turn, the number of turns that are needed to execute the relaxed plan is $2n$, as long as $2n$ is not less than the number of operators $|SO_1| + |SO_2|$ that belong to the plan.

One can obtain a more pessimistic (and possibly more realistic) heuristic value by assuming that the antagonist plays only those operators, that *only he* controls. All remaining operators are played by the protagonist. This alternative re-distribution step can be formulated as shown in Algorithm 2 (part 2, alternative).

The main difference between the optimistic re-distribution step and the pes-

/* Pessimistic re-distribution step: Let the protagonist play all operators that he controls and the antagonist the remaining. */

/* Calculate how many operators $n_1 \in \mathbb{N}$ *could* be played by pl. 1. */

$n_1 := |SO_1 \cup (SO_2 \cap O_1^+)|$

/* Calculate how many operators $n_2 \in \mathbb{N}$ *can only* be played by pl. 2. */

$n_2 := |SO_1| + |SO_2| - n_1$

/* Return the heuristic value. */

return $2 \max\{n_1, n_2\}$

Algorithm 2: Computation of the Adversarial FF heuristic **(part 2, alternative)**

simistic re-distribution step is that in the latter the protagonist plays all selected operators he can possibly play (and the antagonist only the remaining ones), whereas in the former the antagonist helps as much as he can.

It is worth noting that the heuristic values of the (optimistic) adversarial FF heuristic do not differ from the heuristic values of the standard FF heuristic if both players control the same set of operators. Using the pessimistic adversarial FF heuristic, one would obtain heuristic values that are exactly twice as large as the values obtained by the standard FF heuristic.

4.2 Example

Assume that the sets of relaxed operators of the protagonist and antagonist are

$$O_1^+ := \{\ (1 \to 2)\ ,\ \dots\ ,\ (1 \to 8)\ ,\ (9 \to 10, 11)\ \} \quad \text{and}$$
$$O_2^+ := \{\ (1 \to 2)\ ,\ (1 \to 3)\ ,\ (8 \to 9)\ \}, \quad \text{respectively,}$$

the current state s we want to evaluate is $s_0 := \{1\}$, and the goal states are given by the goal propositions $1, \dots, 10$.

During the forward step, we obtain the following operator and rule sets:

$$\begin{aligned}
S[0] &= \{\ 1\ \}, & O[0] &= \{\ (1 \to 2)\ ,\ \dots\ ,\ (1 \to 8)\ \},\\
S[1] &= \{\ 1\ ,\ \dots\ ,\ 8\ \}, & O[1] &= \{\ (1 \to 2)\ ,\ (1 \to 3)\ ,\ (8 \to 9)\ \},\\
S[2] &= \{\ 1\ ,\ \dots\ ,\ 9\ \}, & O[2] &= \{\ (1 \to 2)\ ,\ \dots\ ,\ (1 \to 8)\ , (9 \to 10, 11)\ \},\\
S[3] &= \{\ 1\ ,\ \dots\ ,\ 11\ \}.
\end{aligned}$$

The plan extraction in the backward step gives the following sets of rules and unsatisfied preconditions:

$$\begin{aligned}
G[3] &= \{\ 1\ ,\ \dots\ ,\ 10\ \}, & SO[2] &= \{\ (9 \to 10, 11)\ \},\\
G[2] &= \{\ 1\ ,\ \dots\ ,\ 9\ \}, & SO[1] &= \{\ (8 \to 9)\ \},\\
G[1] &= \{\ 1\ ,\ \dots\ ,\ 8\ \}, & SO[0] &= \{\ (1 \to 2)\ ,\ \dots\ ,\ (1 \to 8)\ \},\\
G[0] &= \{\ 1\ \}.
\end{aligned}$$

The sets SO_1 and SO_2 are therefore $SO_1 = \{\ (1 \to 2)\ ,\ \dots\ ,\ (1 \to 8)\ ,\ (9 \to 10, 11)\ \}$ and $SO_2 = \{\ (8 \to 9)\ \}$. Finally, the optimistic heuristic value is computed as follows:

$$
\begin{aligned}
|SO_1| &= 8 > 1 = |SO_2|, \text{ thus } max := 1 \\
n &:= |SO_1| - |SO_1 \cap O_2^+| \\
&= |SO_1| - |\{\ (1 \to 2)\ ,\ \dots\ ,\ (1 \to 8)\ ,\ (9 \to 10, 11)\ \} \cap \\
&\qquad\qquad \{\ (1 \to 2)\ ,\ (1 \to 3)\ ,\ (8 \to 9)\ \}| \\
&= |SO_1| - |\{\ (1 \to 2)\ ,\ (1 \to 3)\ \}| \\
&= 8 - 2 = 6 \\
h &:= \max\{2n, |SO_1| + |SO_2|\} = \max\{12, 9\} = 12
\end{aligned}
$$

The pessimistic heuristic value, on the other hand, would be computed as follows:

$$
\begin{aligned}
n_1 &:= |SO_1 \cup (SO_2 \cap O_1^+)| \\
&= |\{\ (1 \to 2)\ ,\ \dots,\ (1 \to 8)\ ,\ (9 \to 10, 11)\ \} \cup (\{\ (8 \to 9)\ \} \cap O_1^+)| \\
&= |\{\ (1 \to 2)\ ,\ \dots,\ (1 \to 8)\ ,\ (9 \to 10, 11)\ \} \cup \emptyset| \\
&= 8 \\
n_2 &:= |SO_1| + |SO_2| - n_1 = 8 + 1 - 8 = 1 \\
h &:= 2\max\{n_1, n_2\} = 2\max\{8, 1\} = 16
\end{aligned}
$$

5 Experimental Results

We modified the simple rocket domain from [BF95] as follows: The task is to transport a set of packages from initial to destination cities using a single airplane with infinite capacity. A state is described by the current package positions and the fuel level of the airplane tank (full or empty) plus an additional auxiliary variable *nop* needed to encode the fairness condition that the antagonist does not only perform no-ops. Possible actions are *flying* from one city to another one if the tank is full, *loading* a package into the plane, *unloading* a package from the plane unless the same package has just been loaded without an intermittent flying action, *fueling* the plane if necessary, and performing *no-op*s. Flying and loading can only be done by the protagonist, fueling only by the antagonist, and unloading and no-ops by both, with the antagonist being barred from two consecutive no-ops without a flight in between. The goal of the protagonist is to transport the packages to specified target cities. The agents take turns, starting with the protagonist.

There always exists a winning strategy for the protagonist since neither *destructive* unload actions nor repeated no-ops by the antagonist are allowed, so that the antagonist is eventually forced to contribute to the plan by fueling or unloading.

Assume two cities Paris and London, one package to be transported from London (packageInLondon) to Paris (packageInParis), and the plane initially in London (airplaneInLondon) with its tank empty (¬full). The variable “nop” is true iff the adversary has already performed a no-op since the last flight. A winning strategy for the protagonist is depicted in Figure 1.

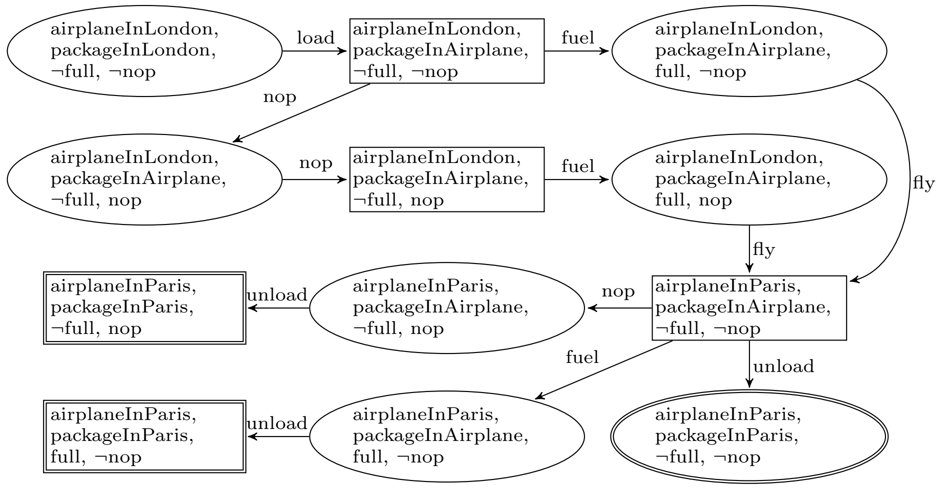

Figure 1: Cargo transport from London to Paris. The initial state is depicted on the upper left hand side, goal states are doubly framed. The protagonist moves in elliptic, the antagonist in rectangular vertices.

We experimented with solvable problems from this example domain with varying numbers of cities and packages. We compared running times, memory usage and vertex creations for uninformed breadth-first search, AO* search with the standard FF heuristic under the assumption of full cooperation (only one agent, controlling all operators), and AO* search with the optimistic and pessimistic adversarial FF heuristic. In addition, we encoded the same tasks as conditional planning problems under full observability in NuPDDL and solved them using MBP [CPRT03]. The results are summarized in Table 1.

The results in Table 1 suggest that in domains where the the antagonist controls operators that may contribute to a plan, AO* search with optimistic and pessimistic adversarial FF heuristic often outperforms AO* search with standard FF heuristic and uninformed forward search. It is competitive with the symbolic approach used in MBP.

ℓ/p	BFS time	BFS mem.	BFS vertices	AO* + h_{FF} time	AO* + h_{FF} mem.	AO* + h_{FF} vertices	AO* + $h_{\text{opt. adv. FF}}$ time	AO* + $h_{\text{opt. adv. FF}}$ mem.	AO* + $h_{\text{opt. adv. FF}}$ vertices	AO* + $h_{\text{pes. adv. FF}}$ time	AO* + $h_{\text{pes. adv. FF}}$ mem.	AO* + $h_{\text{pes. adv. FF}}$ vertices	MBP pre	MBP search	MBP BDD
2/1	0.014	**1**	*44*	0.046	**1**	**37**	0.023	**1**	**37**	**0.022**	**1**	**37**	*0.064*	*0.004*	14822
2/2	**0.048**	*2*	*152*	0.064	**0**	96	0.088	**0**	96	0.077	*2*	**84**	*0.384*	*0.084*	290495
3/3	0.354	*6*	*2106*	0.311	4	1131	0.571	*6*	1106	**0.226**	**1**	**285**	*4.128*	*3.668*	166012
3/4	0.870	*49*	*8211*	0.696	38	2766	0.781	42	2499	**0.538**	**9**	**1053**	*39.890*	*82.073*	654147
3/5	*5.556*	*159*	*43785*	2.599	67	12676	3.019	60	11644	**0.672**	**12**	**1836**	–	–	–
3/6	*87.691*	*987*	*237264*	12.421	252	61154	11.896	238	54469	**2.526**	**111**	**10333**	–	–	–
4/6	–	–	–	*203.678*	*1572*	*408768*	37.762	700	129362	**3.973**	**214**	**14115**	–	–	–
4/7	–	–	–	*756.138*	*3389*	*1006666*	131.505	1725	341093	**1.375**	**60**	**4262**	–	–	–
4/8	–	–	–	–	–	–	–	–	–	**29.129**	**921**	**100263**	–	–	–
4/9	–	–	–	–	–	–	–	–	–	**129.305**	**1729**	**361899**	–	–	–
4/10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

ℓ: cities, p: packages, $h_{\text{opt. adv. FF}}$: optimistic adv. FF heuristic, $h_{\text{pes. adv. FF}}$: pessimistic adv. FF heuristic, BDD: BDD nodes

Table 1: Experimental results for the airplane benchmark problems. We used a Java implementation, running on a machine with an Intel Core 2 Duo processor with 2.4 GHz and a Windows Vista x64 system. The problem instances for MBP were run on a machine with two Quad Xeon processors with 2.66 GHz and a Linux x64 kernel. The time-out, indicated by dashes, was set to fifteen minutes. Times are given in seconds, memory usage in MB. Bold entries highlight best results and italic entries worst, respectively.

References

[BF95] Avrim L. Blum and Merrick L. Furst. Fast Planning Through Planning Graph Analysis. In *Proc. of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1636–1642, 1995.

[BG01] Blai Bonet and Héctor Geffner. Planning as Heuristic Search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[BJ03] The Duy Bui and Wojciech Jamroga. Multi-Agent Planning with Planning Graph. In *Eunite 2003*, pages 558–565, 2003.

[BKS06] Daniel Bryce, Subbarao Kambhampati, and David E. Smith. Planning Graph Heuristics for Belief Space Search. *Journal of Artificial Intelligence Research*, 26:35–99, 2006.

[CPRT03] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1–2):35–84, 2003.

[FN71] Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[GLP05] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26(2):62–72, 2005.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke. *Automata, Logics, and Infinite Games. A Guide to Current Research*, chapter 2, pages 23–40. Springer-Verlag, 2002.

[HN01] Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[HZ98] Eric A. Hansen and Shlomo Zilberstein. Heuristic Search in Cyclic AND/OR Graphs. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 412–418, 1998.

[Nil98] Nils J. Nilsson. *Principles of Artificial Intelligence*, chapter 3, pages 99–129. Springer-Verlag, 1998.