Principles of AI Planning

16. Determinization and Hints for Project 3

Bernhard Nebel and Robert Mattmüller

Albert-Ludwigs-Universität Freiburg

February 3rd, 2012

Principles of AI Planning

February 3rd, 2012 — 16. Determinization and Hints for Project 3

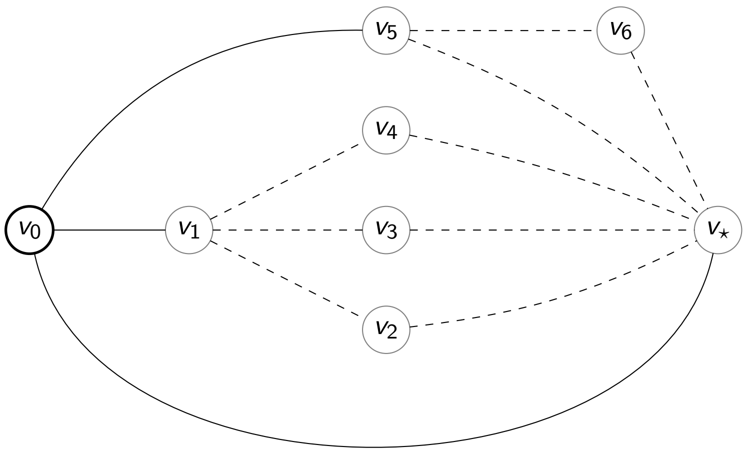
1 Determinization

- Example
- Motivation
- All-Outcomes Determinization

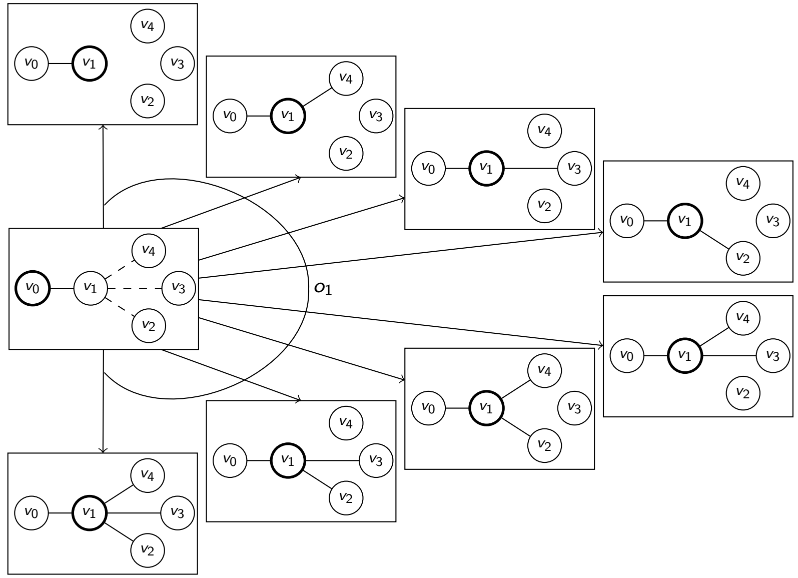
Example: Canadian Traveler's Problem (CTP)

- Objective: Get from initial location v_0 to goal location $v_\star$.
- Roads may be blocked due to snow.
- Status of road is only observed once the agent arrives at one of the end-points of the road.
- Formally: Moving to a location nondeterministically assigns blocking status (blocked or unblocked) to incident, yet unseen, roads.

Example: Canadian Traveler's Problem (CTP)



Example: Part of the State Transition System



Motivation: Determinization

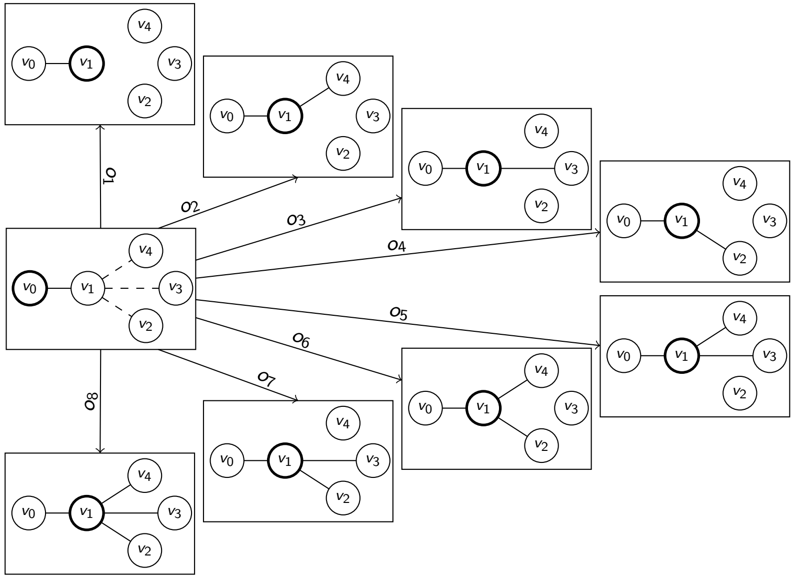
Determinizations are useful in nondeterministic planning:

- To plan on the determinized task, combined with replanning if unexpected outcome occurs.
- To plan on the determinized task and use the result as a heuristic for, e.g. AO*.
- To enable the use of heuristics from classical planning for, e.g. AO*.

Determinization Strategies

- Single-Outcome Determinization
  - One deterministic operator per nondeterministic operator (e.g. the most likely outcome, if we have some knowledge about likelihoods).
  - Task in determinization potentially unsolvable.
- All-Outcomes Determinization:
  - All potential outcomes in determinization.
  - Solution preserving.
  - Might lead to exponentially many operators if nesting of conjunctions and nondeterminism is allowed.
    E.g., $\langle \chi, (a_1 \mid \neg a_1) \wedge (a_2 \mid \neg a_2) \wedge \cdots \wedge (a_n \mid \neg a_n) \rangle$ induces exponentially many deterministic operators).

Example: All-Outcomes Determinization



2 Hints for Project 3

Overview

Objective: Strong planning in `pyperplan` using AO* search.

What you need to know:

- (P)PDDL encoding of nondeterministic operators
- The AO* algorithm
- Which heuristics to use to guide AO*
- Output format for strategies
- How to verify strategies
- Which benchmark problems to use
- How to evaluate the performance of AO* and various heuristics

(P)PDDL Encoding of Nondeterministic Operators

Nondeterminism in operators encoded with keyword `oneof`.

- In general: arbitrary nesting of `oneof` with `and`, `when`, ...
- In `pyperplan`: One outermost `oneof` separating deterministic STRIPS effects, or no `oneof` at all (if the operator is deterministic).

Example ((P)PDDL encoding of a schematic operator)

$o(\overline{x}) = \langle \chi(\overline{x}), \{e_1(\overline{x}), \dots, e_n(\overline{x})\} \rangle$ encoded as

(:action o
 :parameters ($\overline{x}$)
 :precondition ($\chi(\overline{x})$)
 :effect (oneof $e_1(\overline{x}) \dots e_n(\overline{x})$))

(P)PDDL Encoding of Nondeterministic Operators

Example

```
(:action put-on-block
  :parameters (?b1 ?b2 - block)
  :precondition (and (holding ?b1) (clear ?b2))
  :effect (oneof (and (on ?b1 ?b2)
                      (emptyhand)
                      (clear ?b1)
                      (not (holding ?b1))
                      (not (clear ?b2)))
                 (and (on-table ?b1)
                      (emptyhand)
                      (clear ?b1)
                      (not (holding ?b1))))
)
```


(P)PDDL Encoding of Nondeterministic Operators

We have prepared a modified version of `pyperplan` that is able to parse and ground nondeterministic planning tasks in this format.

Your task in the project

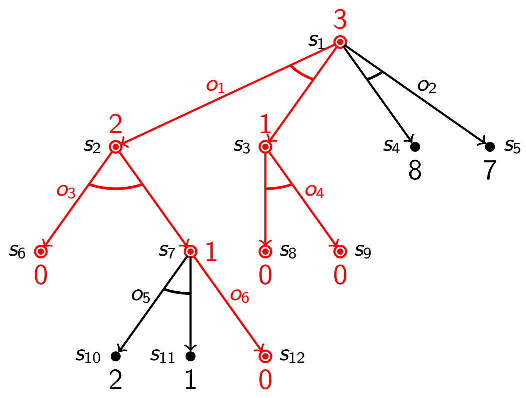
Simply download the modified version of `pyperplan` from the course website.

The AO* Algorithm

Procedure ao-star

def ao-star($\mathcal{T}$):
- let $\mathcal{T}_e$ initially consist of the initial state s_0.
- **while** $\mathcal{T}_p$ has unexpanded non-goal node:
  - expand an unexpanded non-goal node s of $\mathcal{T}_p$
  - add new successor states to $\mathcal{T}_e$
  - **for all** new states s' added to $\mathcal{T}_e$:
    - $f(s') \leftarrow h(s')$
  - $Z \leftarrow s$ and its ancestors in $\mathcal{T}_e$ along marked actions.
  - **while** Z is not empty:
    - remove from Z a state s w/o descendant in Z.
    - $f(s) \leftarrow \min_{o \text{ applicable in } s}(1 + \max_{s \xrightarrow{o} s'} f(s'))$.
    - mark the best outgoing action for s
      (this may implicitly change $\mathcal{T}_p$).
- **return** an optimal solution graph.

The AO* Algorithm



- red: optimal solution graph with f-values, marked outgoing hyperarcs (red), and solution labeling (circled nodes).
- solution: $\pi(s_1) = o_1$, $\pi(s_2) = o_3$, $\pi(s_3) = o_4$, $\pi(s_7) = o_6$.

The AO* Algorithm

Implementation details:

- Represent $\mathcal{T}_e$ explicitly as an AND/OR tree.
  (A state reachable along different operator sequences should be represented by one node corresponding to each such sequence.)
- Each node n of $\mathcal{T}_e$ should contain:
  - the state s represented by n,
  - a reference to the parent node of n,
  - references to the child nodes of n, sorted by the operator that leads there (represent each applicable operator o by an outgoing hyperarc with "head" node n and "body" nodes representing all the successor states in $app_o(s)$), and
  - the following data, possibly updated during dynamic programming steps ("**while** Z is not empty"):
    - a distinguished outgoing hyperarc that is marked,
    - the f-value of n, and
    - the solution status of n (solved, unsolved – not shown in pseudocode).

The AO* Algorithm

Implementation details (ctd.):

- Find unexpanded non-goal nodes by tracing down marked hyperarcs.
- Expand an unexpanded non-goal node of $\mathcal{T}_p$ with maximal h-value.
- Work on a topological ordering of Z in the dynamic programming step.
- Node is labeled as solved if it is a goal node or there exists an outgoing hyperarc such that all successor nodes along that hyperarc are labeled as solved.

Your task in the project

Implement AO* tree search in `pyperplan` as described above.

Which Heuristics to Use to Guide AO*

To guide AO*, one should estimate the remaining (strong/weak) distances to the goal.

Your task in the project

Implement an interface between your AO* implementation and the heuristics defined in `pyperplan` for deterministic problems by applying the heuristics from the deterministic setting to the all-outcomes determinization of the planning task at hand. Experiment with several of these heuristics.

You may also experiment with running an optimal classical planner (e.g., the `pyperplan` implementation of A* with a PDB heuristic) on the determinization and use the resulting plan length as a heuristic for AO*.

Output Format for Strategies

General strategy output format:

```
<n> <atom-list>
%%
<m> <action-list>
%%
policy <k> <map-list>
```

- `<n>`: size of `<atom-list>`,
- `<atom-list>`: space-separated list of atoms,
- `<m>`: size of `<action-list>`,
- `<action-list>`: space-separated list of operators,
- `<k>`: size of `<map-list>`,
- `<map-list>`: space-separated list of $(s, \pi(s))$ pairs of the form "`<l> <st> <action>`":
  - `<l>`: number of atoms true in s,
  - `<st>`: space-separated list of indices of the atoms true in s,
  - `<action>`: index of $\pi(s)$ (indexing starts with 0).

Output Format for Strategies

Example

```
4 (on A B) (clear A) (clear B) (on B A)
%%
4 (pick A) (putdown A) (pick B) (putdown B)
%%
policy 2 2 0 1 0 2 3 2 2
```

denotes the strategy π such that

$$\pi(\{\texttt{(on A B)}, \texttt{(clear A)}\}) = \texttt{(pick A)} \text{ and}$$
$$\pi(\{\texttt{(on B A)}, \texttt{(clear B)}\}) = \texttt{(pick B)}$$

Your task in the project

Implement the strategy format and make sure to output the strong plans you find in that format.

How to Verify Strategies

To check the correctness of your implementation, you should verify the plans your planner produces.

Your task in the project

Download the plan verifier from `http://ippc-2008.loria.fr/wiki/images/f/f0/Verifier.tgz` and use it to verify the plans your planner produces.

You can use the `verify` binary you get by compiling the above code instead of the `validate` tool already called from within `pyperplan` (`validate` only supports deterministic problems, whereas `verify` specializes on nondeterministic/probabilistic planning).

In case you have trouble compiling the verifier, we also provide a Linux binary at `http://www.informatik.uni-freiburg.de/~ki/teaching/ws1112/aip/verify`.

Which Benchmarks Problems to Use

We have prepared two toy domains for testing, experimenting and benchmarking. The domain files and generators for problem files of different sizes are available from the course website.

- The `coinFlip` domain: n coins, initially untouched. Agent can toss each coin once, nondeterministically leading to the coin showing heads or tails. After tossing a coin, the agent can turn the coin (as often as the agent wants). In the goal, all coins should show heads.

Which Benchmarks Problems to Use

We have prepared two toy domains for testing, experimenting and benchmarking. The domain files and generators for problem files of different sizes are available from the course website.

- The `chainOfRooms` domain: sequence of n rooms, sequentially connected by doors. Initially, robot is in the left room, status of doors (open or closed) is unknown, light is off in all rooms. Robot can open closed doors, turn on light in the current room (observation action: nondeterministically, the robot will see that the door to the next room to the right is open or closed), and move left and right (if door is open). Goal: being in the rightmost room.

Which Benchmarks Problems to Use

We have prepared two toy domains for testing, experimenting and benchmarking. The domain files and generators for problem files of different sizes are available from the course website.

Your task in the project

Download the domains and problem generators from the course website and familiarize yourself with the domains.

Evaluation

Your task in the project

Generate `coinFlip` and `chainOfRoom` problems of various sizes, run your AO* implementation with various determinization-based heuristics on them, and measure and report run times, node expansions, and strategy sizes.

You may additionally do some more research, tune your algorithm based on the findings from your experiments, . . .

Summary

- AO* tree search for strong planning in `pyperplan`.
- Reuse of classical heuristics via determinization.
- Infrastructure code, verification and benchmarking.

Summary

- We know that this looks like a lot of work.
- You may want to restrict yourselves to a few heuristics, limited experiments, a rather preliminary implementation of AO*, . . .
- On the other hand, if you have a lot of motivation, here are some ideas how to continue:
  - more experiments, more benchmarks
  - AO* for DAG shaped or generally graph shaped transition systems
  - dealing with the symmetries in the two benchmark domains
  - . . .