Introduction

Why AI Planning? What is it?

Informal Description

Patrik Haslum
Planning is the art and practice of thinking before acting.

Jörg Hoffmann
Selecting a goal-leading course of action based on a high-level description of the world.

Just a bit more formally...
Planning is the reasoning process required to generate a plan – a sequence of action that transforms a given state of a system into a desired one.

Motivation

General Questions Covered

- What are important/interesting properties of algorithms?
- What does it mean that one algorithm is better than another?
- How does one prove such properties? E.g., how does one show:
  - termination?
  - that one algorithm is better than another?

→ Illustrated with AI planning and planning heuristics.
Why AI Planning? What is it?

Games, e.g., Sliding Tile Puzzle, 15 Puzzle, $n^2-1$ Puzzle

![Initial State](Image)

![Goal State](Image)

---

Blocksworld

Start Configuration

Desired Configuration

---

Cranes in a Harbor

---

Standard Planning Benchmark in the International Planning Competition (IPC) and every planning lecture.
Why AI Planning? What is it?

Greenhouse

Source: https://www.lemnatec.com/
Copyright: With kind permission from LemnaTec GmbH
Further reading:
- The IPC Scanalyzer Domain in PDDL (see paper above).

Introduction

Problem Definition

What is Classical Planning?

We focus on the “base case” of AI planning: Classical Planning

- Discrete (no time).
- Deterministic.
- Fully observable.
- Single-agent.

More formally, a classical planning problem consists of:

- A finite set of (deterministic and discrete) actions.
- A (fully known) initial state.
- A set of (fully known) goal states.

A solution (or plan) is any sequence of actions transforming the initial state into a goal state.

Formalism

A classical planning problem $P = \langle V, A, s_I, g \rangle$ consists of:

- $V$ is a finite set of state variables (also called: facts or propositions).

$V = \{\text{CrateAtLoc1}, \text{HoldCrate}, \text{TruckAtLoc1}, \text{TruckAtLoc2}, \text{CrateInTruck}\}$
A classical planning problem $\mathcal{P} = (V, A, s_i, g)$ consists of:

- $A$ is a finite set of actions. Each action $a \in A$ is a tuple $(pre, add, del, c) \in 2^V \times 2^V \times 2^V \times \mathbb{R}_+^+$ consisting of a precondition, add and delete list, and action costs. (We often only give a 3-tuple if there are no action costs.)

\[\text{load pre: } \{\text{HoldCrate, TruckAtLoc1}\} \quad \text{unload pre: } \{\text{CrateInTruck, TruckAtLoc1}\} \quad \text{add: } \{\text{CrateAtLoc1}\} \quad \text{del: } \{\text{HoldCrate}\}\]

\[\text{take pre: } \{\text{CrateAtLoc1}\} \quad \text{put pre: } \{\text{HoldCrate}\} \quad \text{add: } \{\text{CrateAtLoc1}\} \quad \text{del: } \{\text{HoldCrate}\}\]

$V$ is the goal description (encodes a set of goal states).

$s_i$ is the initial state (complete state description).

$g \subseteq V$ is the goal description (encodes a set of goal states).
Problem Definition

Formalism, cont’d I

Action application:
- An action $a \in A$ is called applicable (or executable) in a state $s \in S$ if and only if $\text{pre}(a) \subseteq s$. Often, this is given by a function: $\tau(a, s) \iff \text{pre}(a) \subseteq s$.
- If $\tau(a, s)$ holds, its application results into the successor state $\gamma(a, s) = (s \setminus \text{del}(a)) \cup \text{add}(a)$. $\gamma: A \times S \rightarrow S$ is called the state transition function.

Example: The action $\text{take}$ (pre: $\{\text{CrateAtLoc1}\}$, add: $\{\text{HoldCrate}\}$, del: $\{\text{CrateAtLoc1}\}$) is applicable in state $\{\text{CrateAtLoc1}, \text{TruckAtLoc2}\}$ resulting into $\{\text{TruckAtLoc2}, \text{HoldCrate}\}$.

State Transition Systems

Example

Every classical planning problem is a compact representation of a state transition system, i.e., of how states are transformed into each other.

State Transition Systems

Definition (State Transition System)

A state transition system is a 6-tuple $(S, L, c, T, I, G)$, where
- $S$ is a finite set of states.
- $L$ is a finite set of transition labels.
- $c: L \rightarrow R^+_0$ is a cost function.
- $T \subseteq S \times L \times S$ is the transition relation.
- $I \in S$ is the initial state.
- $G \subseteq S$ is the set of goal states.
State Transition Systems

Size Increase of the State Space in Blocks World

- \( n \) blocks, 1 gripper.
- A single action takes a top-most block with the gripper and
  - puts it immediately onto some other top-most block
  - or onto the table.

<table>
<thead>
<tr>
<th>blocks</th>
<th>states</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>73</td>
</tr>
<tr>
<td>5</td>
<td>501</td>
</tr>
<tr>
<td>6</td>
<td>4051</td>
</tr>
<tr>
<td>7</td>
<td>37633</td>
</tr>
<tr>
<td>8</td>
<td>394353</td>
</tr>
<tr>
<td>9</td>
<td>4596553</td>
</tr>
</tbody>
</table>

Planning problems (very compactly!) define state transition systems (cf. Blocks World).

To solve a planning problem, we construct the underlying state transition system.

Each node in the search space corresponds to a state and a sequence of actions within the state transition system.
Recap: A∗ Search

Example

How to find a(n optimal/good) way from Arad to Bucharest?

Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, $h$ is the linear distance (values see last slide).
Recap: A* Search

Example, cont’d

Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, $h$ is the linear distance (values see last slide).

Progression Algorithm

Classical Planning as Instance of Tree Search

Algorithm: Progression State-based Search

Input: A classical planning problem $\langle V, A, s_I, g \rangle$

Output: A solution $\bar{a}$ or fail if none exists

1. $\text{fringe} \leftarrow \{(s_I, \varepsilon)\}$
2. while $\text{fringe} \neq \emptyset$ do
   3. $(s, \bar{a}) \leftarrow \text{nodeSelectAndRemove}(\text{fringe})$
   4. if $s \supseteq g$ then return $\bar{a}$
   5. for $a \in A$ do
      6. if $\text{pre}(a) \subseteq s$ then
         7. $s' = (s \setminus \text{del}(a)) \cup \text{add}(a)$
         8. $\text{fringe} \leftarrow \text{fringe} \cup \{(s', \bar{a} \circ a)\}$
   9. return fail
What are we interested in? Which properties are of interest?

- Does it always *terminate*? If not, can we make it so?
- How can we make the algorithm *more efficient*?
- What’s the runtime?
- Is it *correct*, i.e., is every plan it returns an actual solution?
- Is it *complete*, i.e., does it always find a solution if one exists?
- Is it *optimal*, i.e., does it always find the best solution?

---

**Algorithm:** Progression State-based Search

**Input:** A classical planning problem \((V, A, s_i, g)\)

**Output:** A solution \(\bar{a}\) or *fail* if none exists

1. \(\text{fringe} \leftarrow \{(s, \bar{a})\}\)
2. while \(\text{fringe} \neq \emptyset\)
3. \((s, \bar{a}) \leftarrow \text{nodeSelectAndRemove}(\text{fringe})\)
4. if \(s \supseteq g\) then return \(\bar{a}\)
5. for \(a \in A\) do
6. if \(\text{pref}(a) \subseteq s\) then
7. \(s' = (s \setminus \text{del}(a)) \cup \text{add}(a)\)
8. \(\text{fringe} \leftarrow \text{fringe} \cup \{(s', \bar{a} \circ a)\}\)
9. return *fail*

**Does it always terminate?**

- No, due to cycles in state-space.
Properties: Proof Sketches

Algorithm: Progression State-based Search
Input: A classical planning problem \((V, A, s_i, g)\)
Output: A solution \(\bar{a}\) or fail if none exists

\[
\begin{align*}
1 & \text{ fringe } \leftarrow \{(s_i, c)\} \\
2 & \text{ while } \text{ fringe } \neq \emptyset \text{ do} \\
3 & \quad (s, \bar{a}) \leftarrow \text{nodeSelectAndRemove}(\text{fringe}) \\
4 & \quad \text{if } s \supseteq g \text{ then return } \bar{a} \\
5 & \quad \text{ for } a \in A \text{ do} \\
6 & \quad \quad \text{ if } \text{pre}(a) \subseteq s \text{ then} \\
7 & \quad \quad \quad s' = (s \setminus \text{del}(a)) \cup \text{add}(a) \\
8 & \quad \quad \quad \text{ fringe } \leftarrow \text{ fringe } \cup \{(s', \bar{a} \circ a)\} \\
9 & \text{ return fail }
\end{align*}
\]

How can we make it always terminate?

- Ensure that every search node (state) is explored only once.
- Check the current plan length. Discard nodes of a certain length.

Properties: Proof Sketches

Algorithm: Progression State-based Search
Input: A classical planning problem \((V, A, s_i, g)\)
Output: A solution \(\bar{a}\) or fail if none exists

\[
\begin{align*}
1 & \text{ fringe } \leftarrow \{(s_i, c)\} \\
2 & \text{ while } \text{ fringe } \neq \emptyset \text{ do} \\
3 & \quad (s, \bar{a}) \leftarrow \text{nodeSelectAndRemove}(\text{fringe}) \\
4 & \quad \text{if } s \supseteq g \text{ then return } \bar{a} \\
5 & \quad \text{ for } a \in A \text{ do} \\
6 & \quad \quad \text{ if } \text{pre}(a) \subseteq s \text{ then} \\
7 & \quad \quad \quad s' = (s \setminus \text{del}(a)) \cup \text{add}(a) \\
8 & \quad \quad \quad \text{ fringe } \leftarrow \text{ fringe } \cup \{(s', \bar{a} \circ a)\} \\
9 & \text{ return fail }
\end{align*}
\]

How can we make an algorithm more efficient?

- By including (and studying properties of) heuristics. See later.
- Much more! → I offer research projects and PhD theses!
Analyzing the Planning Algorithm

Properties: Proof Sketches

Algorithm: Progression State-based Search

Input: A classical planning problem \((V, A, s_i, g)\)
Output: A solution \(\bar{a}\) or fail if none exists

1. \(\text{fringe} \leftarrow \{(s_i, \varepsilon)\}\)
2. while \(\text{fringe} \neq \emptyset\) do
3. \((s, \bar{a}) \leftarrow \text{nodeSelectAndRemove}(\text{fringe})\)
4. if \(s \supseteq g\) then return \(\bar{a}\)
5. for \(a \in A\) do
6. if \(\text{pre}(a) \subseteq s\) then
7. \(s' = (s \setminus \text{del}(a)) \cup \text{add}(a)\)
8. \(\text{fringe} \leftarrow \text{fringe} \cup \{(s', \bar{a} \circ a)\}\)
9. return \(\text{fail}\)

Is it complete, i.e., does it always find a solution if one exists?

- This depends on the node selection strategy. And on the fact whether duplicates are considered again.

Is it optimal, i.e., does it always find the best solution?

- Yes, if used with \(A^*\) and an admissible heuristic (see AI lecture/handbook).

Problems of progression search:

- Often very huge branching factor (many actions are applicable to a state).
- The search space size increases exponential with search depth (cf. blocks world!)
- Thus, how we implement the node selection (line 3) has a huge impact on efficiency! (We rather explore the exact path from the initial state to a goal state rather than the entire search space.)
- Which state to explore next is decided by heuristics!
Heuristic Example: Sliding Tile Puzzle

**Initial State**

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>7</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

**Goal State**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

**How far are we still away?**

- Number of misplaced tiles: 13
- “Distance” (horizontal and vertical distance) per tile to goal position → *Manhattan distance*: $1 + \ldots + (2+1) + \ldots$

---

Planning Heuristic Construction

**How to come up with heuristics in a domain-independent way?**

- Perform a problem relaxation.
- Solve the relaxed problem.
- Use the cost of the solution in the relaxed problem as approximation (i.e., heuristic) of the actual problem.

**Example Sliding Tile Puzzle:**

- *Number of misplaced tiles*. Relaxation: We can always move tiles to any location, i.e., ignore all preconditions.
- *Manhattan distance*. Relaxation: We can move a tile, even if the neighbor tile is not free, i.e., ignore some preconditions.
- *Ignore tiles*. Some tiles (i.e., state variables) do not exist.
Formal Definitions and Properties of Heuristics

Definitions

**Definition (Heuristic)**

Given a state transition system \( ts = (S, L, c, T, I, G) \), a heuristic \( h \) is a function \( h: S \rightarrow \mathbb{R}^+ \cup \{\infty\} \).

**Definition (Perfect Heuristic)**

A heuristic \( h^* : S \rightarrow \mathbb{R}^+ \) is called perfect, if for all states \( s \in S \) \( h^*(s) \) is the cost of the cheapest transition from \( s \) to a goal \( s' \in G \). Further, \( h^*(s) = \infty \) for all states \( s \) for which no goal state can be reached.

**Definition (Safe Heuristic)**

A heuristic \( h \) is called safe, if for all states \( s \in S \) \( h(s) = \infty \) implies \( h^*(s) = \infty \).

**Definition (Goal-aware Heuristic)**

A heuristic \( h \) is called goal-aware, if all goal states, i.e., \( s_G \in G \) holds \( h(s_G) = 0 \).

**Definition (Admissible Heuristics)**

A heuristic \( h \) is called admissible, if for all states \( s \in S \), it holds \( h(s) \leq h^*(s) \).

**Definition (Dominance)**

A heuristic \( h_1 \) is said to dominate another heuristic \( h_2 \) if for all states \( s \in S \), \( h_1(s) \geq h_2(s) \).

Analysis of Properties

Why analyzing these properties?

- Because they tell us how “good” (well-informed) they are, and whether one heuristic is better than another.
- The more accurate heuristic estimates, the smaller the explored search space!
- Better-informed heuristics might be harder to compute, so smaller search space does not imply better runtime.
- Every well-informed heuristic should be goal-aware.
- Admissibility guarantees optimality when used with tree search.
- If \( h_1 \) and \( h_2 \) are admissible, and \( h_1 \) dominates \( h_2 \), then \( h_1 \) is more accurate than \( h_2 \) and should create smaller search spaces.
  → We will analyze these properties for heuristics in the assignments.
Delete Relaxation

Definitions, Delete Relaxation

What's the core idea behind delete relaxation?
→ What's true once stays true!

Consider Sokoban after: moving left, down,

These positions are also free! (Since they were free before or have become so.)

[@] = the figure  [$$] = a crate  [/] = a goal position
Delete Relaxation

Heuristic(s) based on Delete-Relaxation

Why delete-relaxation and how to exploit it?

- Solving delete-free planning problems can be done in polynomial time!
- (Whereas solving arbitrary planning problems normally requires exponential space and time.)
- Many heuristics are based on delete-relaxation:
  - $h^{max}$ (shown next!)
  - $h^{Add}$ (improves $h^{max}$ by incorporating all preconditions)
  - $h^{FF}$ (compute a plan for the delete-relaxation)
  - $h^{max}$ and $h^{FF}$ might be covered in the assignments.

### Relaxed Planning Graph: Example from the Crane in the Harbors Domain

```
\( V_0 \quad A_1 \quad V_1 \quad A_2 \quad V_2 \quad A_3 \quad V_3 \quad A_4 \quad V_4 \)
```

Delete Relaxation

### Relaxed Planning Graph: Definition

#### Definition (Relaxed Planning Graph)

Let \( \langle V, A, s_i, g \rangle \) be a (delete-free) planning problem.

Then, a relaxed planning graph (rPG) is a graph \( \langle \bar{V}, \bar{A} \rangle \) consisting of:

- \( \bar{V} = V^0 \ldots V^n, V^i \subseteq V, 0 \leq i \leq n \), a sequence of variable layers.
- \( \bar{A} = A^1 \ldots A^n, A^i \subseteq A, 1 \leq i \leq n \), a sequence of action layers.
- \( V^0 = s_i \).
- \( A^i = \{ a \in A \mid \text{pre}(a) \subseteq V^{i-1} \}, 1 \leq i \leq n. \)
- \( V^i = V^{i-1} \cup \bigcup_{a \in A^i} \text{add}(a), 1 \leq i \leq n. \)

Choose \( n = i \), such that \( V^i = V \) holds.

**Questions:**

- Why is “delete-free” in the problem description put in parentheses?
- Why is \( n \) chosen as is? Is there a bound on \( n \)?

### h^{max} Heuristic

Let \( \mathcal{P} = \langle V, A, s_i, g \rangle \) be a classical planning problem and \( \mathcal{G} = \langle \bar{V}, \bar{A} \rangle \) its rPG.

- \( h^{max}(s) \) returns the first layer number in which all goal variables hold. Meaning: Number of action layers required in \( \mathcal{P}^+ \) to make the hardest variable in \( g \) true (starting in some \( s \in S \), e.g., \( s_i \)).

  - Formally, \( h^{max} \) can be calculated as follows:
    - **action vertex** The cost of an action vertex \( a \in A^i \) is 1 plus the maximum of the predecessor vertex costs.
    - **variable vertex** The cost of a variable vertex \( v \) is 0 if \( v \in V^0 \).
      - For all \( v \in V^i, i > 0 \), the cost of \( v \) equals the minimum cost of all predecessor vertices (these might be either action or variable vertices).
    - **vertex set** For a set of state variables \( \bar{V} \subseteq V \), the cost equals the most expensive variable in \( \bar{V} \).
    - **heuristic** For a state \( s \in S \), \( h^{max}(s) \) equals the cost of \( g \).
The $h_{\text{max}}$ Heuristic: Example

Calculate $h_{\text{max}}$ for the Cranes in the Harbor domain.

\[ s_I = \{\text{CrateAtLoc} 1, \text{TruckAtLoc} 2\}, \quad g = \{\text{CrateInTruck}, \text{TruckAtLoc} 2\} \]

\[ h_{\text{max}}(s_I) = 2, \quad h^*(s_I) = 4, \quad h_{\text{makespan}}(s_I) = 3 \]

Properties of $h_{\text{max}}$ (proofs (trivial) given in lecture talk)

- Perfect?
  - No.
- Safe?
  - Yes.
- Goal-aware?
  - Yes.
- Admissible?
  - Yes.
- "Well-informed"?
  - Not at all. Almost all other heuristics dominate that one.

Theoretical Research Methods ... Illustrated in AI Planning

In this lecture, we ...

- Analyzed (planning) search algorithms, i.e., we ...
  - investigated runtime behavior,
  - investigated space requirements,
  - analyzed heuristics to improve performance, and
- learned that heuristics base on special cases that are computationally easier to compute (we aim at poly-computable heuristics, whereas most planning problems – practically – require exponential time and space)
- Possible outlook: computational investigation of (planning) problems and heuristics. We normally investigate
  - complexity of a problem (with/without relaxation)
  - runtime of algorithms/heuristics
- Literature and material: see Wattle! (soon)

→ Thank you for your attention!