AIP: Artificial Intelligence and Planning

Important Notes About These Notes

These notes were written by me, Mark Ormesher, during my revision for Computer Science exams at King's College London. I am publishing them here to help my classmates and other students in the years below me. To the best of my knowledge they are correct.

- These notes are **not endorsed** by King's College London, the module lecturers, or any member of College staff.
- These notes are **not checked** by any qualified individual, and I offer no guarantee as to how accurate or complete these notes are. They are offered for free, as-is, and I accept no responsibility for how you choose to use them.
- These notes were relevant for my year (2016/17) but **the content for your course may be different**. Check your lecture slides, syllabi, tutorial guides, etc.
- These notes were produced for my own personal use (it's how I like to study and revise). That means that some annotations may by irrelevant to you, and some topics might be skipped entirely.
- Feel free to **share** these notes, however please only share a link to the repo (see the link below), not individual files.

Notes are originally from **https://github.com/markormesher/CS-Notes** and remain the copyright-protected work of Mark Ormesher.

Contents

Recap: Planning Basics

A planner is a **non-domain-specific** tool that can solve problems in the following format:

- An **initial state**, I, composed of facts.
  - **Static** facts do not change.
  - **Dynamic** facts can change during planning.
- A **goal state**, G, also composed of the facts.
  - We define the goal state in terms of only the facts we care about.
- A set of **actions**.
  - Actions have **preconditions** (the facts that must be true for the action to take place) and **effects** (the facts that are set or unset by applying the action).
  - Actions can be described as **grounded** (with instantiated terms) or **lifted** (as a non-instantiated template).

A planner aims to produce some set of actions that will transform I into some state that conforms to G. They do this by performing a **search over the possible problem states**.

Planning vs. Scheduling

Planning means determining which actions to apply; **scheduling** means determining an order and timeline for pre-known actions.

Recap: Searching

A planner searches in a **space of states** of the world we are working in. We are looking at **forward search** (working from I to find some G, rather than the reverse).

A state can be **expanded** by applying the action(s) that are applicable to it (based on their preconditions), which will create new states. Some permutations of actions may produce **duplicate states**; these should be **merged** to avoid wasted effort.

To avoid searching in circles, a **closed list** of states that have already been generated or expanded (both options covered later) and ignoring them when computing successors of any given state. This effectively turns the search space into a tree.

Forward Search Recap

Forward search follows a simple model using a **closed list** and an **open list** (states that still need to be dealt with).

```
closed = { initial state }
open = { initial state }

while (open list is not empty):
    s = next state from open
    newStates = expand s with all applicable actions
    foreach (state in newStates):
        if (state is not in closed):
            add state to open
```

Note: expansion order is arbitrary but must be constant, so that the planner is deterministic.

This is **satisfying planning** - a solution will be found if there is one, and it will be correct, but it may not be optimal.

- What if the open list was a **queue**? We'd have breadth-first search.
- What if the open list was a **stack**? We'd have depth-first search.

Both of these options are **too slow**.

Heuristic Search Recap

A heuristic **estimates the distance** from a given state **to a goal** and is used to select the next state from the open list to expand.

- A heuristic is **admissible** if it never **over-estimates** the distance to the goal.

- A heuristic is **consistent** if moving forwards at a cost of c decreases the heuristic value by at most c.

The heuristic value of a state S is $h(S)$.

In an ideal world, all non-helpful states have a high heuristic value and helpful states have a decreasing value as they approach the goal. For the following examples, we use the following 'dumb' heuristic:

- $h(S) = 0$ for all goal states
- $h(S) > 0$ for all non-goal states

See more: Heuristic Search, page 11

Best-First Search

The open list is a **min-priority queue** with **stable insertion**. The next state to expand is chosen by the lowest heuristic value, with ties broken by preferring the 'oldest' expanded state (and then by some stable arbitrary method, such as left-to-right).

Best-first is **not guaranteed to be optimal** because it ignores the distance travelled so far when considering any given state.

A* Search

This redefines a given state's heuristic as its original cost-to-goal estimate **plus the total cost incurred so far**.

Now, the open list is a stable-insertion min-priority queue on $f(S) = h(S) + g(S)$ where g gives the cost incurred so far, and h is the original cost-to-goal estimate.

This method is finished when the goal is **expanded**, not just added to the open list. This is because other, cheaper goal states may be encountered after finding the first goal.

The first goal to be expanded is an optimal solution, if h is both **admissible** and **consistent**.

- Proof 1: at any point, the lowest $f(S) = h(S) + g(S)$ on the open list estimates the cheapest plan that can solve the problem. g is perfect, and h never over-estimates, so neither will their sum.
- Proof 2 (by contradiction): search always expands the lowest $f(S) = h(S) + g(S)$; if there was a cheaper path to some goal state S' then $f(S') < f(S)$, so it would have been expanded first. If S' isn't on the open list yet, then its parent must have had a more expensive path than S, so S' can't be cheaper than S.

Planning Languages

PDDL

The **Problem Domain Description Language (PDDL)** specifies a syntax for planning problems that are split into two halves:

- The **domain** specifies the world in which the planner will work (the static facts, the actions, etc.).
- The **problem** specifies an instance of the initial and goal states within the domain.

Different versions of PDDL offer different features:

- **PDDL1** offers predicates only (statements that can be true or false at any given point) and is sometimes used as a benchmark
- **PDDL2.1** added temporal and numeric effects
- **PDDL3** added preferences (soft goals) and trajectory constraints (e.g. always P, sometimes P, eventually P, etc.)
- **PDDL+** allows for modelling of hybrid systems with discrete and continuous constraints

More PDDL notes will be added in the future, if required.

SAS+

Operations in SAS+ look different, compared to PDDL:

- **Prevail conditions** are values that don't change during application of an action.
- **Pre-post conditions** specify that the value of a variable changes from A to B during application.
  - The first value can be a special value equivalent to `anything`.

This representation can create a **domain transition graph**:

- One domain exists per variable, v, we care about.
- The graph is a **directed graph** of possible transitions (changes to the value of v).
- Edges: an edge from A to B labelled with O exists iff an operation O exists with the pre-post condition v, A, B.

Intuition: Mutual Exclusion

Simple intuitions can be encoded into some planners. The most basic intuition that can be encoded is **mutual exclusion**: 'facts A and B cannot be true at the same time' (e.g. I cannot be at university and at home at the same time).

This kind of logic can be encoded into **SAS+**, by creating a variable like this:

```
1 at = one of {home, university, work}
```

Mutual exclusion can apply to facts as well as fact values.

Hybrid Domains

Some processes represent **flows**, rather than discrete actions. An action initiates a flow, rather than executing it entirely (e.g. 'charge battery').

Temporality also introduces another dimension to states: the time spent in one, and the duration of an action. Some effects are said to be **time-dependent**, if their effect depends on their duration (e.g. 'charge battery' or 'run heater').

One solution is to use discretisation: use discrete units of time and other continuous quantities (battery charge, heat, etc.), solve the problem based on the discrete units, validate the plan, and then refine the discretisation if necessary.

Refining Plans

Evaluating Plans

Modelling the entire world of a problem is often difficult because it would overwhelm the planner, so only the important elements are considered. Using an **approximate world** produces a **approximate plans** though, so they should be evaluated against real-world data or complete simulators and refined as necessary.

Improving Plans and Planning

Several techniques exist for improving plan quality and/or planner speed:

- **Tightening constraints** allows a planner to reduce the search space (pruning) by discarding more unusable states.
- **Symmetries** in problem domains can be exploited to reduce complexity (i.e. if 10 entities are identical then they can be grouped; they no longer need to be considered as 10 separate entities, as an action can be applied to any random item from the group with the same effect).

Policies

Planning finds a single solution for a single scenario, but in some applications policies can be more useful. **Policies** work as rules or **decision trees** to indicate actions that should be taken for any given world state.

The process of converting a planning-based solution to a policy-based solution is usually as follows:

- Using real-world or probabilistically accurate data, a variety of representative problem definitions are created.
- Planning is used to create a solution for each problem.
- Classification algorithms (a branch of machine learning) are applied to convert problem/-solution patterns into policies. The WEKA framework and J48 algorithm are suited to this task.

Policies often run in a 'feedback loop': observe state, consider policies, apply action(s), repeat.

To be fully effective, policies need sensible **default actions**: when the observed state is outside the range of scenarios they can reason with, a default action should be applied (e.g. 'when load balancing data is outside recognised bounds, just use the battery with the most charge').

Heuristic Search

During forward heuristic search, every decision of which state to expand is based on an **evaluation function** $f(n)$. This function is a **cost estimate of reaching the goal** from that state.

The choice of $f(n)$ determines the search strategy (*see more: Heuristic Search Recap, page 4*). One component of $f(n)$ is usually $h(n)$, or the **heuristic function**. $h(n)$ is the estimated cost of reaching the goal from the state n, and should be zero if n is a goal state.

Heuristics may be described as **admissible**, **informative** and/or **consistent** (*see more: Heuristic Search Recap, page 4*). The challenge is finding a **domain-independent heuristic function that is admissible *and* informative**.

Finding Good Heuristics

A 'good' heuristic is:

- Admissible, for optimality;
- Informative, for guided, efficient search;
- Easy to calculate (because it has to be computed at every state).

Good heuristics often come from **relaxed problems**: some constraints are removed, making the problem easier to solve. The cost of an optimal solution to the relaxed problem is an admissible heuristic for the original problem, because the relaxed problem has inherently cheaper solutions.

Relaxed Planning Graph

The relaxed planning graph (RPG) is a **domain-independent heuristic** - it can be applied to any problem with no knowledge of the world in which is operates. The RPG heuristic involves finding a path from the current state s to the goals G whilst **ignoring the delete effects of each action**. The length of the path is used as a heuristic value for the state s.

An RPG is made of alternative **fact layers** and **action layers**. The fact layer $f(n)$ determines the actions that are available in the action layer $a(n+1)$ (i.e. those with preconditions that are satisfied in $f(n)$). The next fact layer $f(n+1)$ is then constructed by applying all of the actions in $a(n+1)$ in parallel, ignoring any delete effects. As a result, **fact layers never shrink**.

In short, $f(0) = s$ and $f(n+1) = f(n) + a(n+1)$, ignoring delete effects.

To construct the relaxed plan from the RPG, we **work backwards** through the graph, knowing that at each fact layer $f(n)$ we have to achieve some goals $g(n)$.

- Start from the last fact layer, containing the problem goals.

- For each fact in $g(n)$:
  - If it is in $f(n-1)$, add it to $g(n-1)$;
  - Otherwise, choose an action from $a(n)$ that adds the fact; add it to $O(n)$ and add its preconditions to $g(n-1)$.
- Stop at $g(0)$.
- The relaxed solution is the sequence of actions $< O_1, ...O_n >$.
- The **heuristic value is the count of actions** in the relaxed solution.

Example: Constructing an RPG

- The $pack$ can be at A or B, or $loaded$.
- The $truck$ can be at A or B.
- Actions:
  - $loadX$ (req $truckAtX$, $packAtX$, del $packAtX$, add $packLoaded$)
  - $unloadX$ (req $truckAtX$, $packLoaded$, del $packLoaded$, add $packAtX$)
  - $driveXY$ (req $truckAtX$, del $truckAtX$, add $truckAtY$)
- Initial state: $truckAtB$, $packAtA$
- Goal state: $packAtB$

$f(0)$	$a(1)$	$f(1)$	$a(2)$	$f(2)$	$a(3)$	$f(3)$
$packAtA$	$driveBA$	$packAtA$	$loadA$	$packAtA$	$loadA$	$packAtA$
$packAtB$		$packAtB$	$driveAB$	$packAtB$	$unloadA$	$packAtB$
$packLoaded$		$packLoaded$	$driveBA$	$packLoaded$	$unloadB$	$packLoaded$
$truckAtA$		$truckAtA$		$truckAtA$	$driveAB$	$truckAtA$
$truckAtB$		$truckAtB$		$truckAtB$	$driveBA$	$truckAtB$

$g(0)$	$O(1)$	$g(1)$	$O(2)$	$g(2)$	$O(3)$	$g(3)$
$packAtA$	$driveBA$	$packAtA$	$loadA$	$packAtA$	$loadA$	$packAtA$
$packAtB$		$packAtB$	$driveAB$	$packAtB$	$unloadA$	$packAtB$
$packLoaded$		$packLoaded$	$driveBA$	$packLoaded$	$unloadB$	$packLoaded$
$truckAtA$		$truckAtA$		$truckAtA$	$driveAB$	$truckAtA$
$truckAtB$		$truckAtB$		$truckAtB$	$driveBA$	$truckAtB$

Key: Goal state, selected action, true fact/applied action, untrue fact/ignored action.

Following this relaxed plan gives $h(s) = 4$.

Enforced Hill Climbing

EHC is a **very fast** heuristic search strategy that always selects the node with a heuristic **strictly better** than the best it has seen so far, or applies breadth-first search until such a node is found, as follows:

1. Define $s = I$ and $best = h(I)$ (I is the initial state).
2. Expand s.
3. If there is a successor state s' with $h(s') < best$ the update $best$ and return to (2).
4. If no such state exists, do breadth-first search until one is found, then return to (2).

EHC is **not complete**, because it has **no backtracking**. It is an inexpensive effort to find a fast solution and should always be followed by a complete algorithm.

Fast-Forward (FF) Planner

FF is a forward-chaining heuristic seach planner. The heuristic used is the **relaxed planning graph**. FF uses a **fast local search** (EHC) followed by a **systematic search** (best-first search). The second stage makes FF a **complete planner**.

See more: Relaxed Planning Graph, page 11

See more: Enforced Hill Climbing, page 12

See more: Best-First Search, page 5

Helpful Actions

FF also uses a **helpful actions filter** when expanding a state during EHC to ignore actions that are not helpful towards achieving the goal. Helpful actions are **determined by the RPG**: actions are helpful if they could achieve a goal in $g(1)$. This risks reducing completeness, but EHC isn't complete anyway.

Pattern Databases

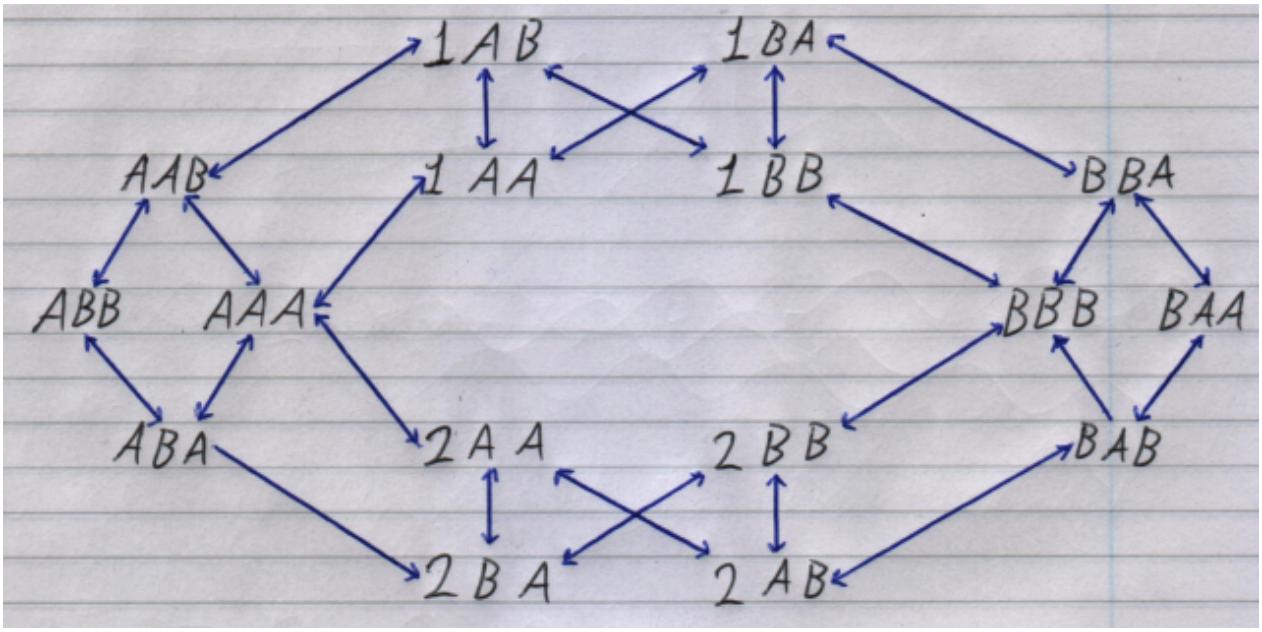
Pattern databases reduce the state search space by **combining states that match a given pattern**. The length of a solution for the reduced state space can be used as a heuristic for the original problem.

Example

- Package: atA, atB, $inTruck1$, $inTruck2$
- Truck1: atA, atB
- Truck2: atA, atB
- *Actions*: drive, load, unload
- *Shorthand*: $AAB = packageAtA, truck1AtA, truck2AtB$

Initial state: ABB; goal state: $B**$

Entire state space:



State space with the projection $\pi(package)$: