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.

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 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
    add s to closed
    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 expanded state that has been 'waiting' for longest (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 evaluation function 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 heuristic 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.
  - *See more: Durative Actions in PDDL2.1, page 27*.
- **PDDL3** added preferences (soft goals) and trajectory constraints (e.g. always P, sometimes P, eventually P, etc.).
  - *See more: Planning with Preferences, page 21*.
- **PDDL+** allows for modelling of hybrid systems with discrete and continuous constraints.
  - *See more: Hybrid Planning & PDDL+, page 36*.

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, that 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.

Refining Plans & Creating Policies

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).
  - A similar idea is used for pattern database heuristics (*see more: Pattern Databases, page 12*).

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 or 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 (or on a route from the initial state *via* that state, depending on the algorithm).

The choice of $f(n)$ determines the search strategy (*see more: Heuristic Search Recap, page 6*). 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 6*). 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 it 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 $h(s) = 3$.

Pattern Databases

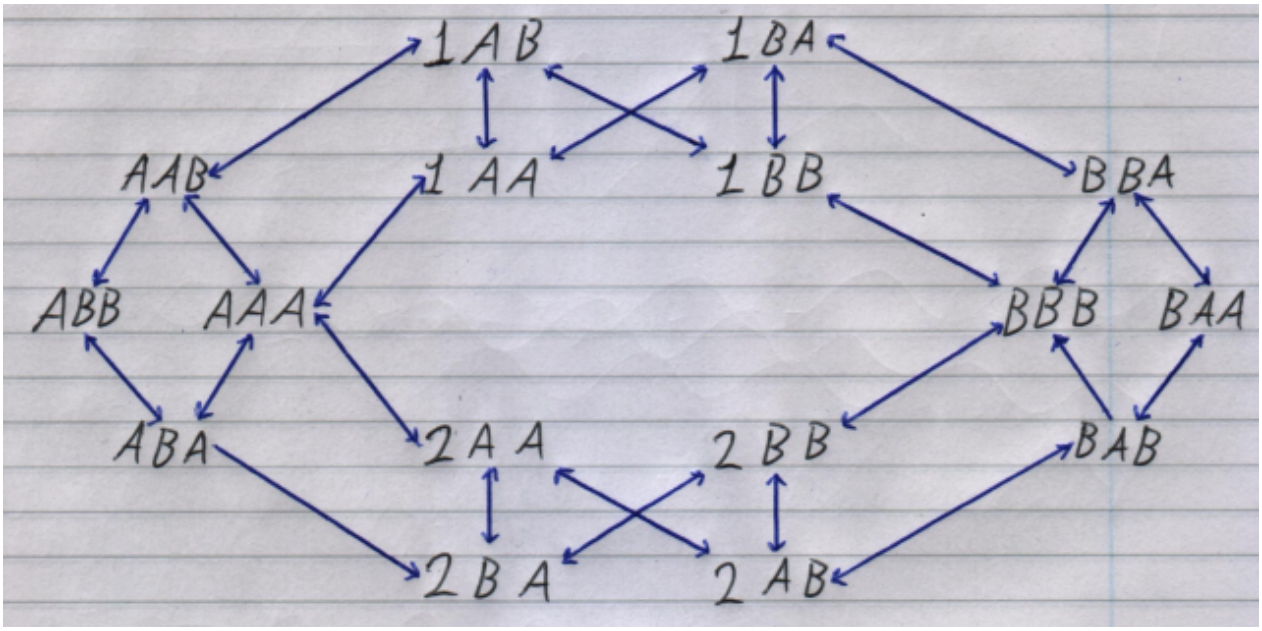
Pattern databases reduce the state search space by **combining states that match a given pattern**. The length of a solution for the reduced state space can be used as a heuristic for the original problem.

Example

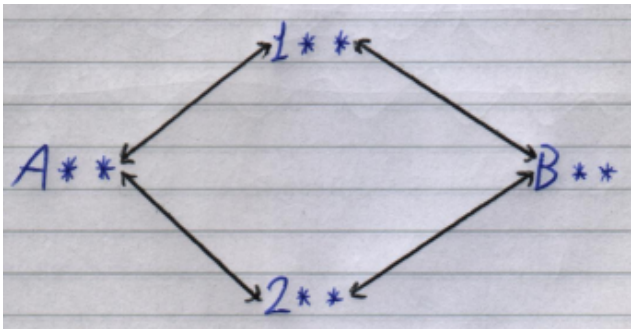
- Package: atA, atB, $inTruck1$, $inTruck2$
- Truck1: atA, atB
- Truck2: atA, atB
- *Actions*: drive, load, unload
- *Shorthand*: $AAB = packageAtA, truck1AtA, truck2AtB$

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

Entire state space:



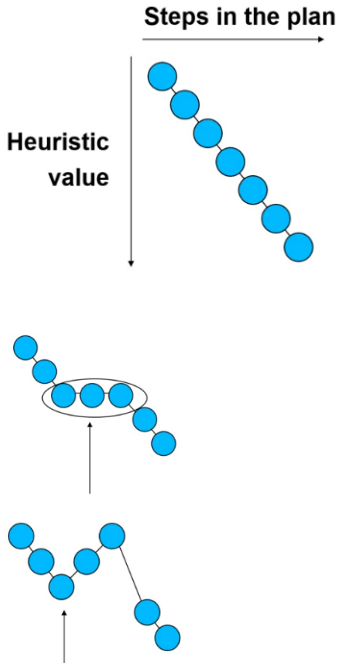
State space with the projection $\pi(package)$:



The Trouble with Heuristics

Heuristics are **not perfect**, because a perfect heuristic requires that the problem has already been solved. In reality, the right move might not always lower the heuristic, and the lowest heuristic might not always be a smart move.

The challenge is **finding a search strategy** that isn't defeated by an imperfect heuristic. **Local search in particular is vulnerable** to bad decisions, because there is no backtracking (*see more: Local Search, page 15*).



Ideal situation: the heuristic drops as the planner gets closer to a goal.

Plateaus: the heuristic values level out for a while.

Local minimums: the right move sends the heuristic value up before it goes down to the goal.

The last two situations are problematic, because in these situations **the heuristic is not informative**.

Combining Heuristics

We can combine two heuristics, $h_1(S)$ and $h_2(S)$, both estimating the distance and/or cost to the goal. We can then maintain **two open lists** and alternate between them. Successors are evaluated with **both heuristics** and placed into **both open lists**.

One heuristic can provide guidance while the other is stuck on plateaus, and provide **diversification** by giving two 'opinions'. The second heuristic can even be a random number generator, which performs surprisingly well!

Local Search

In a nutshell:

1. Start at a state S.
2. Expand S, generating successors.
3. S' = one of these successors.
4. Set $S = S'$.
5. Go back #1 and collect $200.

Heuristics (*see more: Heuristic Search, page 11*) are used to make the decision at the third step.

Enforced Hill Climbing

EHC is a **very fast** local 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.

EHC tackles plateaus and local minimums (*see more: The Trouble with Heuristics, page 14*) by carrying out breadth-first search until an improvement happens. This is why it's 'enforced': it only commits to a path once it believes it will lead towards the goal.

Fast-Forward (FF) Planner

FF is a forward-chaining heuristic search planner. The heuristic used is the **relaxed planning graph** (*see more: Relaxed Planning Graph, page 11*). FF uses a **fast local search** (EHC) followed by a **systematic search** (*see more: Best-First Search, page 7*). The second stage makes FF a **complete planner**.

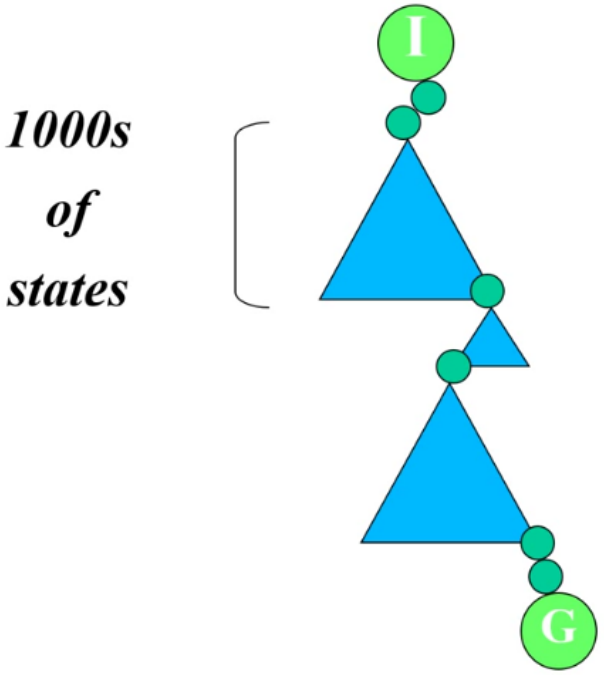
Helpful Actions

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.

Problems with EHC & FF

Imperfect heuristics may lead to **dead ends**, which EHC will get stuck in. In FF, for completeness, the planner then resorts to a systematic search from the initial state. This is **slow**.

In EHC, a **lot of time** is also spent at the breadth-first stages when no better heuristic exists. On the following diagram, green nodes show the quick jumps, blue nodes show the slow, systematic stages:



Identidem - An Alternative to EHC

On a plateau or local minimum, the heuristic is at its **least informative**, so systematic search is inefficient. Why not use **local search with restarts**?

- As in EHC, we record the best state seen so far.
- When expanding a state S, choose a successor S' **at random**, even if it's not the best so far.
- If planning from S' gets more than d steps away from the best without getting out of the plateau, jump back to the best state seen so far and pick another random successor.

  - d is the **probe depth bound**.
  - d starts small (2 or 3), and is then gradually increased if things don't improve.
- When a better state is found, Identidem commits to it and continues on from there.

Diversification and Intensification

Diversification: try lots of different things, in an attempt to brute force past an uninformative heuristic. The ideal solution in the probing stage is to slightly listen to the heuristic, but not much (*see more: Neighbourhoods in Identidem, page 17*).

Intensification: try a lot of options around a given state, hopefully near where the goal might be.

Local search is a balancing act: with Identidem, long probes are good for diversification because they will spread out into the search space and short probes are good for intensification because they will stay fairly close to the starting state.

Different Neighbourhoods

When expanding a state S in local search, a **neighbourhood function** is used to find successors. The simplest neighbourhood is all states reachable from S by applying a single action. Not all actions are helpful, so often not all successors will be interesting to consider.

Neighbourhoods in FF

In FF, during the EHC stage, a **helpful actions filter** is used to choose states reachable by applying the actions related to the first layer of the relaxed planning graph. *See more: Fast-Forward (FF) Planner, page 15*.

Neighbourhoods in Identidem

Identidem only chooses one successor, but may evaluate many states, which is expensive. It gets around this cost by using a **random subset neighbourhood** and evaluating only those - this ties in nicely with restarts.

Picking **3 or 4 random states** seems to work well: more would increase the evaluation cost, and less would reduce the odds of finding a useful successor.

When the random set is chosen, the successor is chosen via **roulette selection**, which is randomised but biased by the heuristic. Imagine a roulette wheel, where lower-heuristic states have a wider wedge of the wheel. This can be achieved by giving a state with the heuristic $h(S) = x$ a 'wedge' proportional to $1/x^b$, where b is a 'dial' of bias ($b = 0$ gives all states an even 'wedge').

Systematic Search vs. Local Search

- Systematic search is **good for puzzle problems**, where there are lots of ways to take a bad move. As it has no backtracking, local search is very likely to eventually go down a bad path.
  - It's also good for proving optimality and it's *probably* complete - if there is a solution, it will find them or die trying.
- Local search is **generally quicker** for finding *some* solution, but it may not be optimal.
  - Repeated local search often gets a near-optimal solution, but without proof.
  - Local search is **incomplete**, but planners can fall back to a systematic search.

Solution Costs

For any given state, we have:

- The **distance-to-go**, which is a heuristic estimate of the **number** of steps to go.
- The **cost-to-go**, which is a heuristic estimate of the **cost** of steps to go.
- E.g. commuting everywhere by helicopter is probably fewer steps but much more expensive.

Some examples of approaches using these metrics (*see more: Heuristic Search Recap, page 6*):

- Best-first search with $f(S) = h(S) = distance\_to\_go$.
  - This is sometimes called greedy search.
  - It usually gives a small search tree.
  - It looks for the shortest plan whilst ignoring cost.
- A* search with $f(S) = g(S) + h(S)$, where $g(S) = cost\_so\_far$ and $h(S) = cost\_to\_go$.
  - $f(S)$ is an estimate of a complete plan via S.
  - It ignores distance, so expands a lot of nodes.
  - If some nodes have zero cost, a plateau is encountered.

Both of these are strict, but what if we don't need 'optimal' and just want '**good enough**'? What if we only care whether or not the plan fits inside some **budget** C?

Bounded-Cost Search

In each state, we know the cost $g(s)$ to reach it, and we have an estimate of future cost $h(s)$ to the goal. We care about the states where:

$$f(s) = g(s) + h(s) \leq C$$

The states matching this condition make up the **focal list**. We can then sort the focal list by the **distance** to go. This is a compromise that does a good job of finding plans quickly (using the distance) and staying within the budget (using the cost).

Problem: what if $h(s)$ *under*estimates? The more it underestimates, we the closer we get to non-bounded-cost search.

Solution: We use a **possibly-non-admissible** heuristic, $\hat{h}(S)$. This shrinks the focal list, at the potential cost of excluding in-budget plans.

Problem: what if $h(s)$ *over*estimates a lot?

Solution: BEES.

Bounded-Cost Explicit Estimation Search (BEES)

- If the focal list is not empty, pick the state with the smallest distance-to-go estimate.
- Otherwise, use A* search.

With this approach, the value chosen for the budget C is important:

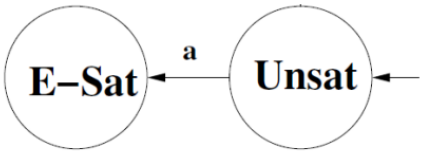
- If C is too tight, we end up with an always-empty focal list and just do A* search.
- If C is too loose, we end up with a huge focal list, and we're just doing greedy search.
- Ideal: somewhere in the middle.

Planning with Preferences

`sometime` Constraints

Rather than requiring that a fact is true at the end of a plan (i.e. in the goal state, as usual), we can specify that it has to happen **at some point** during the plan with the syntax `(sometime get_coffee)`.

Achieving these constraints can be recorded with an automaton:



`(sometime a)`

This shows that we start unsatisfied, then once the action a is applied, we are eternally satisfied.

Preferences

If our `sometime` constraints **aren't absolutely necessary** but we would *like* to achieve them, they can be recorded as preferences. These allow them to be skipped, but impose a **violation cost** penalty for doing so. A **higher cost of skipping** an action encodes a **stronger preference** for doing it.

```
(preference p0 (sometime (at mark university)))
(preference p1 (sometime (at mark starbucks)))

cost(p0) = 100  # it's more important for me to go to
cost(p1) = 10   # university than it is for me to get coffee
```

Preferences can be **combined with 'hard goals'**:

```
(:goal (and
    (at mark home)  # i need to go home at the end of the day
))

(:constraints (and
    (preference p0 (sometime (at mark university)))
    (preference p1 (sometime (at mark starbucks)))
))
```

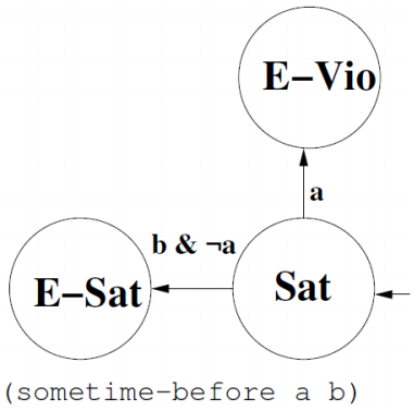
`sometime-before` Constraints

The `sometime` constraints can be extended by specifying that they are **only satisfied if achieved before achieving some other fact**. Continuing the example above, lets say that I only need coffee **if** I'm going to university, and I need to have it **before** I go. This can be implemented with `sometime-before`:

```
(sometime-before (at mark university) (at mark starbucks))
```

This means that **if I don't go to university, then I don't need to get coffee**. This allows a non-university plan to be created without requiring either a useless trip to get coffee or the violation cost for not doing so.

Again these can be expressed with an automaton:



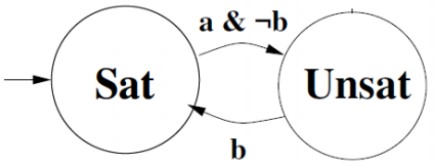
This time we start satisfied, and can move in two directions. If we achieve a before achieving b, we move (up) to an eternally violated state because we can no longer achieve b before a. If we achieve b and we haven't yet achieved a yet, we move (left) to an eternally satisfied state because we can never break this constraint (whether we achieve a or not, we will have achieved b first).

`sometime-after` Constraints

The `sometime` constraints can be extended by specifying that they are **only satisfied if achieved after achieving some other fact**:

```
(sometime-after (at mark starbucks) (at mark toilet))
```

This means that **if I go to Starbucks, then some time after that I need to go to the toilet**. Once again, an automaton can be used to represent this:



(sometime-after a b)

Once again we start satisfied, and if nothing happens we would remain there (if we don't achieve a, then we don't need to achieve b after it). If we achieve a and not b then we will move to an unsatisfied state, where we remain until we achieve b (thereby making the statement 'we achieved b after a' true). Note that neither state is eternal.

Important note: `(sometime-after a b) != (sometime-before b a)`

Goal Preferences

A goal preference is a fact we would **like to achieve** by the end of the plan, with an associated **violation cost** for not doing so.

```
(:goal (and
    (at mark home)                      # i need to go home at the end
    (preference p0 (todos complete))    # i would like to complete my todos
))
```

Compiling Goal Preferences

Goal preferences can be handled by the types of planners that we've seen so far:

- Add a fact $normal-mode$ as a precondition to every action.
- Add an action end that deletes $normal-mode$ and adds $end-mode$.
- Every preference p_i has two actions:
  - $collect(p_i)$ has $end-mode$ and p_i as it's preconditions and adds p'_i.
  - $forgo(p_i)$ has $end-mode$ and $not(p_i)$ as it's preconditions, adds p'_i **and increases the cost by** $cost(p_i)$.
- Set the goal to $(and(p'_0)(p'_1)(p'_2)...)$.

In this system, when the plan enters $end-mode$ every preference has either been achieved (p_i) or not achieved ($not(p_i)$), and the appropriate $collect$ or $forgo$ actions can be applied to determine the final cost.

RPG Heuristics

See more: Relaxed Planning Graph, page 11

Working from an end goal where all goal preferences are achieved, an RPG heuristic **may not always be useful**. For a plan with 3 goal preferences, the RPG from any state (where hard goals have been achieved) will always be along the lines of end, $forgo(p_0)$, $forgo(p_1)$, $forgo(p_2)$ and the heuristic value will always be $h(S) = distance\_to\_go = 4$.

This is because RPG **only considers distance and ignores cost**, causing it to use this very expensive but very short plan.

Distance to Go vs. Cost to Go

For a problem where the goal is made entirely of preferences, **distance to go** (i.e. the minimum number actions needed to call something a goal state) is **zero**. For the same problem, the **cost to go** can range from **zero** when all preferences are satisfied to **the sum of all preference violation costs**. In short:

$dist(S) = 0$ by giving up immediately and applying no actions.

$cost(S) = 0$ by applying actions to meet all goal preferences.

We have two heuristic values and **they're both useless**. However, these are **not the same** relaxed plan.

What if we combined them into **distance-cost pairs**?

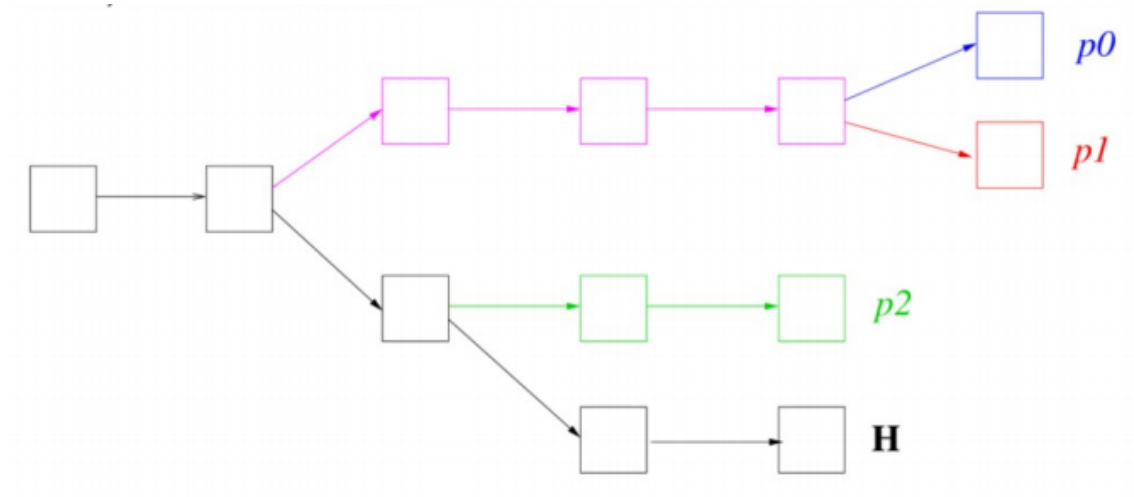
- 4 actions will get down to cost 20
- 8 actions will get down to cost 10
- 10 actions will get down to cost 7
- etc.

Distance-Cost Pairs

These are **compromises between the two values** seen earlier, and they may be a lot more useful.

Distance-cost pairs are computed by **pretending that preference goals are hard goals** and **constructing the RPG**. Whilst finding the relaxed plan, **extra information is recorded** to note whether an action was applied to achieve a real hard goal or a preference.

Once this information is collected, a **tree** from the current state can be constructed where **inner nodes are actions** and **hard or preference goals are achieved at leaves**. In the following example there was one hard goal and three preferences:



Violation costs of the preferences are as follows:

```
cost(p0) = 20
cost(p1) = 10
cost(p2) = 3
```

There are now various possible pairs of distances and costs, depending on what is achieved:

- H: 5 actions with cost 33
- H, p_2: 7 actions with cost 30
- H, p_0, p_2: 11 actions with cost 10
- H, p_1, p_2: 11 actions with cost 20
- H, p_0, p_1, p_2: 12 actions with cost 0
- etc.

The trouble is that **trying all combinations would create too many pairs** ($O(2^n)$ for n preferences). Some pairs may also be useless, because all pairs with a given cost are dominated by the one with the shortest distance ('10 actions, cost 20' is useless if we already have '8 actions, cost 20'), and vice-versa for all pairs with a given distance being dominated by the cheapest.

A compromise can be reached by using a **greedy algorithm**. This approach takes the relaxed plan, selects a preference and adds it to the plan, records the new distance and cost, then repeats until all preferences have been collected. This produces $O(n)$ plans for n preferences, instead of 2^n. The order in which preferences are selected is often determined by whichever adds the fewest actions.

Turning Distance-Cost Pairs into a Heuristic

Remember BEES? *See more: Bounded-Cost Explicit Estimation Search (BEES), page 20*.

To find a plan with $cost < C$, we can define this 'distance to go' heuristic:

$h_{<C}(S) = min(d|(d, c) \in distance\_cost\_pairs\_for\_S, c < C)$

This states that the heuristic from the state S to find a plan with $cost < C$ is the smallest distance d for all the distance-cost pairs (d, c) for S where $c < C$.

Temporal Planning

Temporal Actions

Temporality introduces another dimension to planning: the **time spent** in a state 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.

Another option is to use **temporal planning** with **durative actions** (actions that take time).

Note: hybrid systems are covered under PDDL+; *see more: Hybrid Planning & PDDL+, page 36*.

Durative Actions in PDDL2.1

Durative actions have:

- A duration (can be a fixed value, a $>, \geq, \leq, <$ bound, or dependent on numeric values from the action's instantiated terms).
- Preconditions and effects that must be true at the **start** of the action.
- Preconditions and effects that must be true at the **end** of the action.
- Preconditions that remain true **throughout** the action (called **invariants** or **over all conditions**).

Example:

```
(:durative-action LOAD-TRUCK
    :parameters (?obj - obj ?truck - truck ?loc - location)
    :duration (= ?duration 2)
    :condition (and
        (at start (at ?obj ?loc))
        (over all (at ?truck ?loc))
    )
    :effect (and
        (at start (not (at ?obj ?loc)))
        (at end (loaded ?obj ?truck))
    )
)
```

Durative Actions in LPGP

(LPGP is another planner, but we don't need to know about it; we're only interested in how it handles durative actions.)

Durative actions can be split into **three separate instant actions**:

- $A_{\vdash}$ to denote A's start.
- $A_{\leftrightarrow}$ to denote A's invariant.
- $A_{\dashv}$ to denote A's end.

The start and end actions have preconditions and actions; the **invariant action has preconditions only**.

Snap Actions

These three actions can be **reduced to just start and end actions**, called snap actions. To make sure actions must be started before they can be ended, and to make sure starts and ends are correctly matched, the following conditions and effects are applied:

- $A_{\vdash}$ adds the fact $A_{started}$ in its effect.
- $A_{\dashv}$ requires $A_{started}$ in its precondition and removes it in its effect.

Problems with Snap Actions

Consider this plan, which reaches the goal: $A_{\vdash} \quad B_{\vdash} \quad A_{\dashv}$

Clearly it cannot be valid, because the durative action B has not ended.

Problem 1: What if $B_{\dashv}$ interferes with the goal?

Solution 1: Insist that **no durative actions can be executing in a goal state**. This can be achieved by adding $\neg A_{started}$, $\neg B_{started}$, etc. to the goal definition, or by making it implicit in the planner. The latter is preferred, because most planners do not support negated goals.

Problem 2: What about invariant conditions that must hold throughout a durative action? (i.e. if A is executing, $A_{invariant}$ must hold)

Solution 2: Insist that in every state where $A_{started}$ is true, $A_{invariant}$ must also be true. This can be achieved by **maintaining a list of active invariants**, updated when actions are started and ended to match the invariants required for the actions that are currently executing. When selecting actions for state expansion, this list is used to ignore actions that would un-set an invariant.

Problem 3: Where did the duration constraints go? $A_{\vdash} \quad B_{\vdash} \quad B_{\dashv} \quad A_{\dashv}$ may be planned even if B takes longer than A. In short, **logically sound $\neq$ temporally sound**.

Solution 3: Apply **decision epoch planning**, **CRIKEY!3** or **POPF**.

Decision Epoch Planning

This approach uses forward-chaining search with **timestamped states** and a **priority queue of pending 'end' snap actions**.

Every state has a timestamp t which defines the time at which it exists, relative to the initial state at $t = 0$.

In any state S at time t with a queue Q, the next state S' is achieved by doing one of two things:

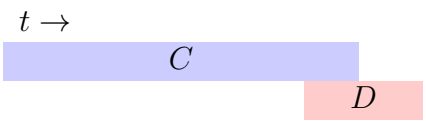
- Start a durative action, by...
  - ...applying a snap action $A_\vdash$ at time t
  - ...inserting $A_\dashv$ into Q at time $t + duration(A)$
  - ...advancing time, $S'.t = S.t + \epsilon$
- End a durative action, by...
  - ...removing and applying the first end snap action in Q
  - ...setting $S'.t$ to the scheduled time of the end snap action, plus ϵ

Epsilon ϵ is a **tiny amount of time**, representing the amount of time that the plan must **wait between achieving a fact and using it as a precondition**.

Semantics state that the **first action can be applied at time** $t = 0$, without adding ϵ first. Some planners allow actions to be applied simultaneously if they do not interfere with each other.

There is a problem with this: the **start and end timestamps must be fixed** when the action is started, because the values are used in the priority queue. This is not always possible, for example:

- $duration(C) = 10$
- $duration(D) = 1$
- Expected plan: $C_\vdash \rightarrow D_\vdash$ (achieves q) $\rightarrow C_\dashv$ (requires q) $\rightarrow D_\dashv$ (removes q)



This plan is not possible, because after starting C the only options are applying D (which would end too early) or advancing by ϵ (which wouldn't move forwards enough). It is possible to apply the action 'advance by ϵ' many times, but that would severely impact planning time.

CRIKEY!3

Still does forward-chaining search, but replaces the priority queue with **separately managed temporal constraints**. All constraints take one of these forms:

- $\epsilon \leq t(i+1) - t(i)$
  - The time at which action $i+1$ is applied must be at least ϵ after the time at which action i is applied.
  - This is a **sequence constraint**.
- $duration_{min}(A) \leq t(A_{\dashv}) - t(A_{\vdash}) \leq duration_{max}(A)$
  - The difference between the times for $A_{\vdash}$ and $A_{\dashv}$ must be between the minimum and maximum durations for A.
  - This is a **duration constraint**.
- General form: $lb \leq t(j) - t(i) \leq ub$
  - The time between some action i and some action j must be between some upper and lower bounds, ub and lb.

This style of problem is called a **Simple Temporal Problem (STP)**.

- The good news: polynomial algorithms exist to solve STPs (i.e. produce a schedule).
- The bad news: planners need to solve these problems for every state.

Simple Temporal Problems/Networks (STPs/STNs)

Note: when planning with STPs/STNs, a **'zero state'** Z usually exists where $t = 0$.

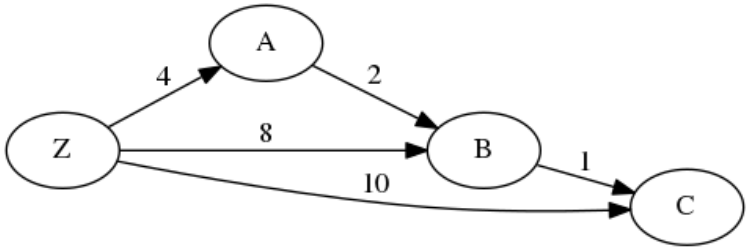
Latest Possible Times

In this example, there are 4 states Z, A, B and C with the following five constraints:

- $t(A) - t(Z) \leq 4$
- $t(B) - t(Z) \leq 8$
- $t(C) - t(Z) \leq 10$
- $t(B) - t(A) \leq 2$
- $t(C) - t(B) \leq 1$

A constraint in the form $t(A) - t(Z) \leq 4$ gives an **upper bound** of 4 on the time from Z to A. This is more intuitive if you rearrange the constraint to $t(A) \leq t(Z) + 4$, which more clearly denotes that A is at most 4 time units after Z.

These constraints produce the following graph, where edges show **maximum separation**:



To find the **latest** possible time: look for the **shortest path**.

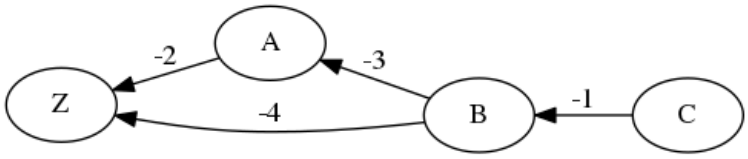
Earliest Possible Times

We now add the following constraints:

- $2 \leq t(A) - t(Z)$
- $4 \leq t(B) - t(Z)$
- $3 \leq t(B) - t(A)$
- $1 \leq t(C) - t(B)$

A constraint in the form $2 \leq t(A) - t(Z)$ gives a **lower bound** of 2 on the time from Z to A. This is more intuitive if you rearrange the constraint to $t(Z) + 2 \leq t(A)$, which more clearly denotes that A is at least 2 time units after Z.

These constraints give the following graph, where edges show **minimum separation**:



To find the **earliest** possible time: look for the **longest path**. Unfortunately graph algorithms usually find the shortest path, so we negate every constraint (multiply everything by -1, which also flips the sign) and then look for the shortest path (equivalent to the longest path of the original solution). This 'hack' is why the edges above are negative.

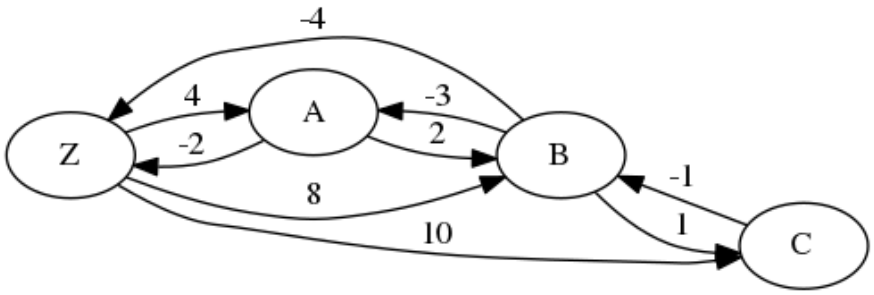
General Case

As shown above, STPs can map to digraphs. The general case is as follows:

- One vertex per time-point (and one for $t = 0$).
- For constraints in the form $lb \leq t(y) - t(x) \leq ub$...
  - An edge $x \to y$ with the weight ub.

- An edge $y \rightarrow x$ with the weight $-lb$.

Applying all of these rules to the previous example gives the following graph:



Minimum/Maximum Timestamps

Once the full STN digraph has been created, the paths between 0 (Z) and actions inform the earliest and latest possible times that things can happen:

- $minDistance(0, j) = x$ means x is the **maximum timestamp** at which j can happen.
- $minDistance(j, 0) = y$ means $-y$ is the **minimum timestamp** at which j can happen. (Negative because of the shortest/longest path hack from above.)

Suggested algorithm for shortest paths: **Bellman-Ford**, because Dijkstra doesn't work with negative edge weights.

When searching for shortest paths, a **negative cycle** means that the STN constraints are inconsistent and there is no solution for those constraints. The Bellman-Ford algorithm will detect negative cycles.

Different graphs are possible for the same set of actions, based the ordering of actions in the plan that is found. Some may be valid, some may be invalid. It is okay to **prune temporally-invalid plans**, but it is not okay to memoise a closed list based on facts alone (as with classical planning), because temporal planning is **order-sensitive**.

POPF - Partial Order Planning Forwards

The combination of forward search and STNs (i.e. CRIKEY!3) creates a **total ordering** by ordering every action after some other action, but that isn't always necessary.

POPF creates a **partial ordering by only enforcing ordering constraints where required**, rather than every time an action is selected.

Formal State Definition

A state S is the 4-tuple $\langle F, V, P, T \rangle$, consisting of:

- F - propositional facts.
- V - values of numeric task variables (not concerned with these now).
- P - the plan so far to achieve S (using snap actions).
  - *See more: Snap Actions, page 28*.
- T - temporal constraints of the steps in P.

The Problem with Total Orderings of Start/End Snap Actions

Consider two actions, A and B, such that:

- B takes longer than A.
- There is no interaction between $A_\vdash$ and $B_\vdash$.
- $A_\dashv$ must come before $B_\dashv$ (i.e. A achieves some precondition of $B_\dashv$).

An ordering that places $A_\vdash$ before $B_\vdash$ will produce a **negative cycle** and, hence, an **invalid STN** (which causes the planner to backtrack). We didn't need to choose which of $A_\vdash$ and $B_\vdash$ came first (because they do not interact), so imposing an unnecessary ordering created this wasted effort.

Reducing Commitment

This is the idea of not imposing orderings unless they are actually required. It requires the planner to **store extra information at each state**, concerning which actions achieve, delete and depend on each fact. This information is used to commit to fewer ordering constraints.

Threats are still resolved with the intuition of forward chaining expansion: **new actions cannot threaten preconditions of previous facts**. This means that if A deletes a precondition of B and B is already in the plan, then the planner will always order A after B. This is okay, because a plan where A comes before B will appear somewhere else in the search space.

Extending States: Propositional

To capture ordering information, we store this **addition information with every state**:

- $F^+(p)$ and $F^-(p)$ store the index of the step that **most recently added or deleted the proposition** p.
- $FP(p)$ is the set of pairs $\langle j, d \rangle$, where:
  - $\langle j, \epsilon \rangle$ denotes that the step j has a start or end precondition on p at that time. Start/end preconditions have to hold at the start or end of an action, so **any action that deletes p must go at least ϵ after j**.

- $\langle j, 0 \rangle$ denotes that the step j marks the end of an action with an over all condition (invariant) on p. Invariants have to hold over the closed period of an action (from *just after* the start until *just before* the end), so **any action that deletes p can happen at the same as j.**
- These rules allow an action to achieve its own invariant at the start, hold it throughout, then delete it at the end.

Using the Extra Information

During planning, the planner uses the extra information and the following rules to decide whether or not ordering constraints need to be applied.

1. Actions with preconditions must start *after* the preconditions are added.

For each `at start` condition p on the action applied at i:

$$t(F^+(p)) + \epsilon \leq t(i)$$

This states that the action at i must be scheduled at least ϵ after the last action that achieved its precondition p.

2. Actions with over all conditions may start *when* the over all conditions are added.

For each `over all` condition p on the action applied at i:

$$\text{If} \quad F^+(p) \neq i \quad \text{then} \quad t(F^+(p)) \leq t(i)$$

This states that the action at i may be scheduled at the same time as the last action that achieved its invariant p. The $F^+(p) \neq i$ clause states that if the action at i was the last thing to achieve p (i.e. it achieved its own invariant) then no ordering constraint is required.

3. An action's delete effects must come ϵ or 0 after actions with preconditions on what is being deleted (depending on the type of precondition).

For every `at start` effect in the action applied at i that deletes p:

$$\forall \langle j, d \rangle \in FP(p), t(j) + d \leq t(i)$$

This states that for every step that had a **start/end precondition** on p (i.e. step j and the pair $\langle j, \epsilon \rangle$), step i must be ordered at least $d = \epsilon$ afterwards.

For every step that had an **invariant precondition** on p (i.e. step j and the pair $\langle j, 0 \rangle$), step i must be scheduled at least $d = 0$ afterwards (i.e. it can happen at the same time).

The action at i must also come at least ϵ after the last action that added p, to keep the plan consistent and safe:

$$t(F^+(p)) + \epsilon \leq t(i)$$

Finally, $F^-(p) = i$ and $FP(p)$ is cleared, because i has deleted the fact p.

4. An action's add effects must come after the last action that deleted what is being added.

For every `at start` effect in the action applied at i that adds p:

$$\text{If} \quad F^-(p) \neq i \quad \text{then} \quad t(F^-(p)) + \epsilon \leq t(i)$$

This states that if the action at i will add the fact p, then it needs to start at least ϵ after the last action that deleted p.

The $F^-(p) \neq i$ clause states that if the action at i was the last thing to delete p then no ordering constraint is required, because an action is allowed to delete and add the same proposition (the delete goes first; this is used for mutexes).

Finally, $F^+(p) = i$.

Why is Partial Ordering Good?

A single partially ordered STN can represent many, many totally ordered STNs, because the totally ordered planner does not know which ordering constraints have a meaning, and which are there purely because it (pointlessly) put them there.

Hybrid Planning & PDDL+

Some processes represent **flows** or **continuous changes**, rather than discrete, instant actions; for example, 'charge battery' is a flow rather than a discrete action. Actions can then be used to initiate and stop flows, and further events can be triggered at certain points during flows.

PDDL+ is an extension of PDDL used to model **hybrid systems**, which include both discrete actions and continuous changes. The following terminology is used:

- **Actions** are performed by agents to change the world in some way.
- **Processes** execute continuously under the direction of the world (i.e. the world dictates whether the process is active or not).
- **Events** are 'performed' by the world when conditions are met.
- Actions vs. events: an action is executed by an agent (such as 'turn on header'); an event is executed by the world itself in response to conditions (such as 'catch fire when $temp \geq 451$').

Hybrid planning is about deciding which actions should be taken by **executive agents** by anticipating the effects of actions and events before they occur. This requires a way to predict what will happen when actions are executed, and when they are not.

Building plans requires making several types of decision:

- **Action choices** decide which discrete actions should be applied, leading to mode/world changes.
- **Timing choices** decide when to apply certain actions.
  - These can be discrete or continuous - it depends on the temporal model.
- **Parameter choices** decide things like flow rates, power levels, etc. They are generally derived from continuous parameter spaces and their choice is governed by mathematical functions.
  - *These will be covered in the next lecture*.

Actions/Events vs. Processes

When actions or events are performed they cause **instantaneous change** in the world. These are discrete changes to the world state. An action or event has either happened or not happened, and when an action or event has happened it is over - it is never *currently happening*.

Processes represent **continuous change** in the world: once they start they generate continuous updates to the world state. They will run over time, changing the world at every instant.

Modelling Change

Before, in temporal planning (*see more: Temporal Planning, page 27*), we have packaged continuous change into discrete chunks (such as 'go to university'). However, PDDL2.1 adds durative actions (*see more: Durative Actions in PDDL2.1, page 27*) that can also include **continuous change effects**:

```
(:durative-action drop-ball
 :parameters (?b - ball)
 :duration (> ?duration 0)
 :condition (and
    (at start (holding ?b))
    (at start (= (velocity ?b) 0))
    (over all (>= (height ?b) 0))
 )
 :effect (and
    (at start (not (holding ?b)))
    (decrease (height ?b) (* #t (velocity ?b)))
    (increase (velocity ?b) (* #t (gravity)))
 )
)
```

Note: `#t` represents the rate of change of time.

The action above makes sense and works, but a better approach would be to **separate** the modelling of the 'drop ball' action from the fate of the ball after this happens. That way, the falling process can be terminated by various possible actions (e.g. catching, hitting, etc.) or events (e.g. bouncing).

```
(:action release
 :parameters (?b - ball)
 :precondition (and
    (holding ?b)
    (= (velocity ?b) 0)
 )
 :effect (and (not (holding ?b)))
)
```

This action make the conditions of the following process true, which **immediately initiates** it.

```
(:process fall
 :parameters (?b - ball)
 :precondition (and
    (not (holding ?b))
    (>= (height ?b) 0)
 )
 :effect (and
    (increase (velocity ?b) (* #t (gravity)))
    (decrease (height ?b) (* #t (velocity ?b))
 )
)
```

This process changes properties of the world (namely the ball's height and velocity), which will eventually **trigger** the following event:

```
(:event bounce
 :parameters (?b - ball)
 :precondition (and
    (>= (velocity ?b) 0)
    (<= (height ?b) 0)
 )
 :effect (and
    (assign (height ?b) (* -1 (height ?b)))
    (assign (velocity ?b) (* -1 (velocity ?b)))
 )
)
```

Finally, we have an **action** to drop the ball, a **process** to make it fall, and an **event** to make it bounce once it hits the floor. Let's add an action to catch it:

```
(:action catch
 :parameters (?b - ball)
 :precondition (and
    (not (holding ?b))
    (>= (height ?b) 4.99)  // lets say that our catching
    (<= (height ?b) 5.01)  // hand is at height = 5
 )
 :effect (and
    (holding ?b)
    (assign (velocity ?b) 0)
 )
)
```

To make the domain more realistic, we should modify it slightly to reduce the velocity on each bounce:

```
(:event bounce
 :parameters (?b - ball)
 :precondition (and
    (>= (velocity ?b) 0)
    (<= (height ?b) 0.0001)
 )
 :effect (and
    (assign (height ?b) (* -1 (height ?b)))
    (assign (velocity ?b) (* (coeffRest ?b) (velocity ?b)))
 )
)
```

`coeffRest` should be negative to make it easier to use. We also slightly increase the height required for the ball to bounce, which gets rid of problems with **Zeno behaviour**.

Zeno Behaviour & Cascading Events

These are both problems within hybrid planning and must be guarded against:

- **Zeno Behaviour** is named after Zeno's Paradox (roughly, 'if you get from A to B by going half way, then going half of the remaining distance, etc., you will always be getting closer but will never arrive'). It describes the behaviour when infinitely many events happen in a finite time.
- **Cascading events** are pairs or loops of actions and/or events that always trigger each other (e.g. a light-sensitive switch that turns the light off when it's bright and on when it's dark).

'Discretise & Validate' Solving Approach

The general discretise and validate approach is as follows:

1. **Convert** the continuous model into a discrete one.
2. **Solve** the model with a discrete timeline.
3. **Validate** the solution using the continuous model and the validator.
    - If valid, hooray!
    - If invalid, refine the discretisation and go back to #1.

Discretisation takes two forms:

- **Variable discretisation**.
- **Time discretisation**.

  - The timeline is broken down into discrete **clock ticks**. Actions/events can only happen at the **start of every clock tick**. For example, if the timeline is broken down into 0.1s ticks, an event can't happen at 0.55s.

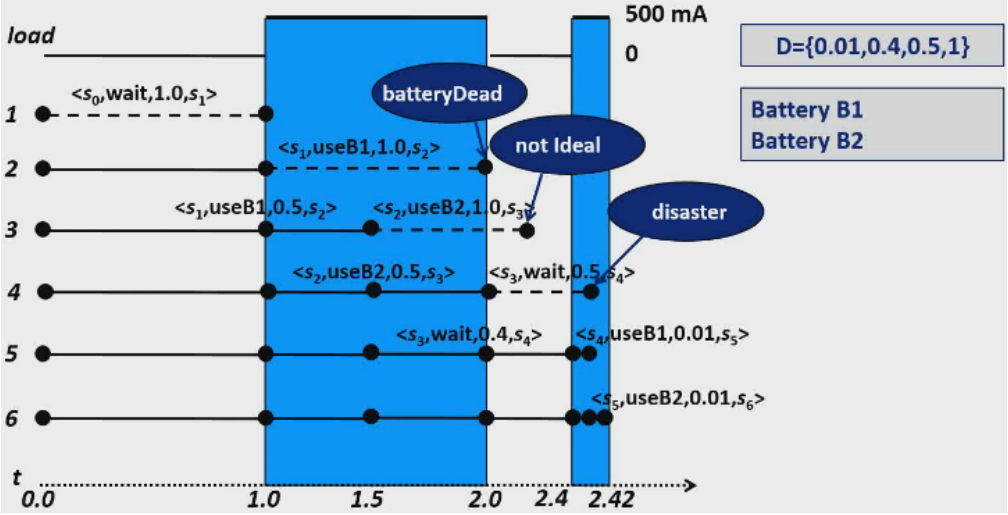
Finding a **suitable discretisation** is not easy:

- Too coarse, and details may be missed.
- Too fine, and the state space explodes.

Non-Uniform Discretisation

A **fine** discretisation will capture details, but will also lead to state space explosion. However, they may be parts of the plan where not very much is happening, so a **coarse** discretisation would be suitable. The solution? Non-uniform discretisation.

This can be achieved by defining a set of durations to use, then using something akin to a **greedy** heuristic that always tries to apply the longest-duration actions first until a suitable plan is found. The following example illustrates this:



- The domain has two batteries ($B1$ and $B2$) and periods of demand (blue blocks).
- The allowable durations are defined as $D = \{0.01, 0.4, 0.5, 1.0\}$.
- Only six states are explored, instead of the hundreds that would be needed with a uniform 0.01 discretisation.

Selection of **appropriate discretisation values** can be found with the discretise and validate approach.

Planning with Expressive World Models

Planning started with simple propositional statements and actions, but **in the real world we have additional complexities** including deadlines, windows of opportunity, discrete numeric values, continuous numeric change, uncertain environments, optimisation goals, etc.

Example Domain for this Section

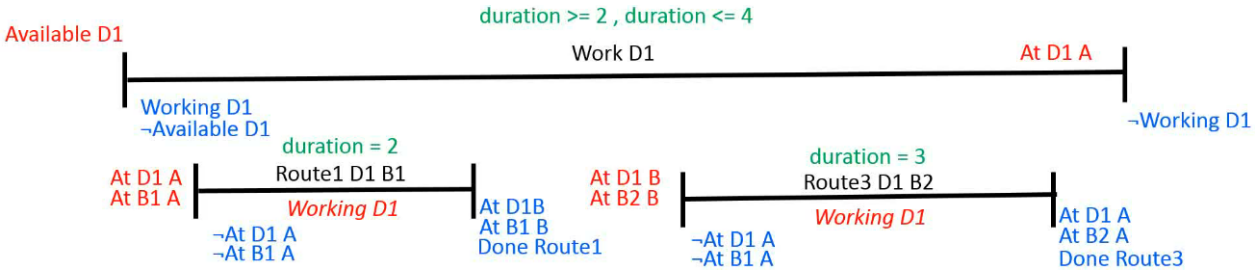
- Domain: bus drivers and routes.
- Drivers have working hours.
- Routes have fixed durations and start/end locations.
- Routes have timetables that they must follow.
- The goals require that each bus route is done.

We have some durative actions:

- $Work, Dx$ means that driver x is working.
- $Routex, Dy, Bz$ means that driver y is using bus z to operate route x.

Actions have:

- Conditions and effects at the start and end.
- Invariant/overall conditions.
- Duration constraints.

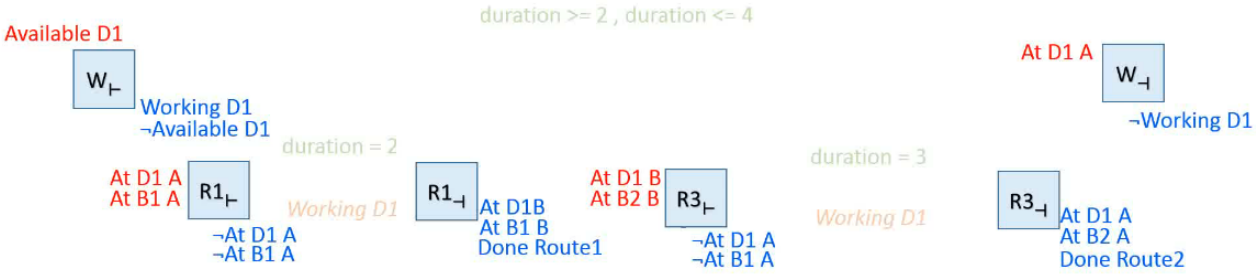


The precondition $At, D1, A$ at the end of the $Work$ action enforces that the driver ends his/her shift at location A.

Recap: Planning with Snap Actions

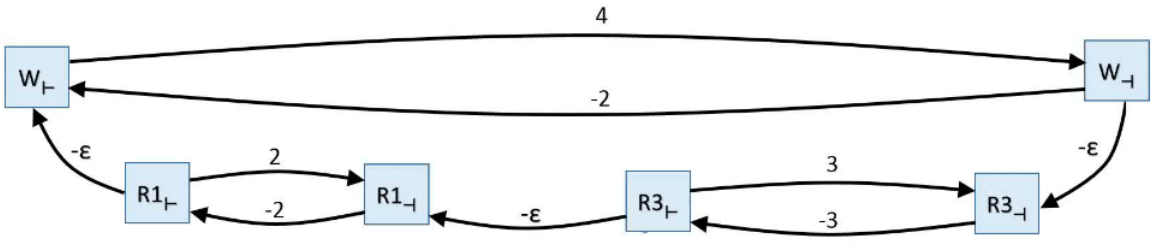
See more: Snap Actions, page 28.

Using snap actions in a planner like CRIKEY! or CRIKEY!3, these actions would be broken down into start and end actions, as so:



There are three main challenges with this approach:

- Make sure end actions cannot be applied unless their start actions have been applied.
  - This is just an extra check on action applicability.
- Overall conditions.
  - For propositional actions we just make sure the conditions are true when the start action is applied and forbid actions that would violate them until the end action is applied.
- Duration constraints.
  - Would be represented as on the diagram below.
  - *See more: Simple Temporal Problems/Networks (STPs/STNs), page 30*.



Constraints:

$W_{\dashv} - W_{\vdash} >= 2$
$W_{\dashv} - W_{\vdash} <= 4$
$R1_{\vdash} >= W_{\vdash} + \varepsilon$
$R1_{\dashv} - R1_{\vdash} = 2$
$R3_{\vdash} >= R1_{\vdash} + \varepsilon$
$R3_{\dashv} - R3_{\vdash} = 3$

The diagram shows duration constraints and ordering constraints (although one is missing). It also has a negative cycle, showing that 5 hours of driving cannot fit into 4 hours of work.

Timed Initial Literals (TILs)

These allow us to model **facts that become true or false at a specific time**. These can be used to model deadlines and time windows (albeit slightly indirectly).

Modelling Deadlines

To do this, we make sure that any actions that achieve the deadline-limited fact have a **pre-condition** on a fact that is only true before the deadline. That fact is **initialised as true**, and then **made false by a TIL** once the deadline expires.

Adding the precondition as an over-all precondition generally makes planners perform better, because they can take advantage of compression-safe actions, but this may not always be possible.

Modelling Time Windows

This follows the same logic as modelling deadlines, but uses **multiple TILs** to 'open' and 'close' the window as required. Again, adding the precondition as an over-all precondition is preferable.

For example:

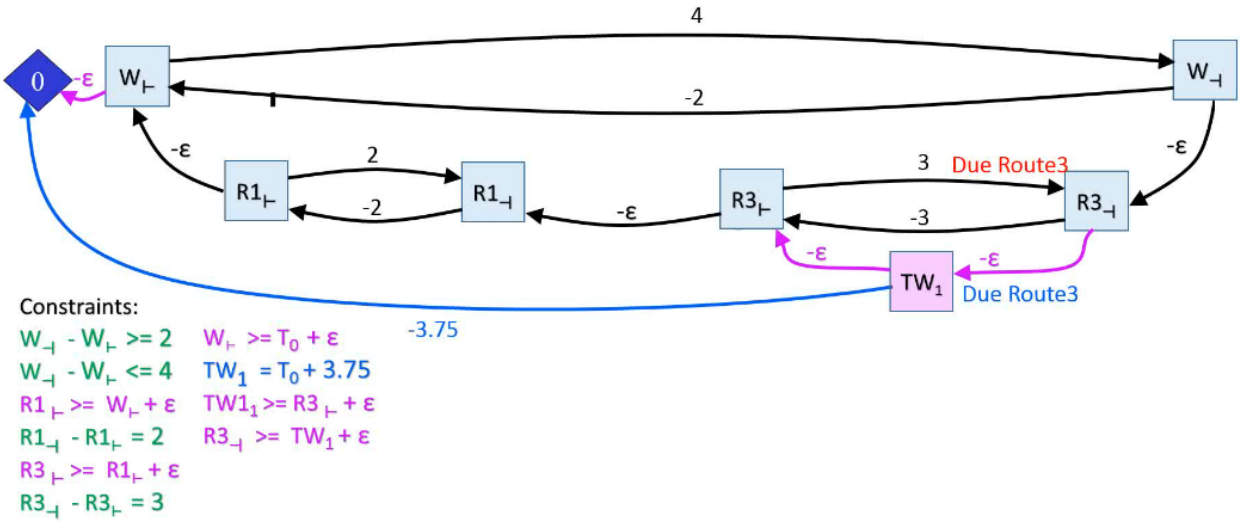
```
(:durative-action bus-route
 :parameters (?d - driver ?r - route ?b - bus ?from ?to - loc)
 :duration (= (?duration (route-duration ?r)))
 :condition (and
    (at start (route ?r ?from ?to))
    (at start (at ?d ?from))
    (at start (at ?b ?from))
    (over all (working ?d))
    (at end (due ?r)) # the bus must arrive during it's due window
 )
 :effect (and
    (at start (not (at ?d ?from)))
    (at start (not (at ?b ?from)))
    (at end (at ?d ?to))
    (at end (at ?b ?to))
    (at end (done ?r))
 )
)

# route 2 is due between 3:45 and 4:00
(at 3.75 (due route2))
(at 4.00 (not (due route2)))
```

Reasoning with TILs in Forward Search

- Order TILs chronologically, because at every point during search the only TIL that be applied is the soonest one that is scheduled.
- At each state, we have a choice:
  - Apply an applicable action.
  - Apply the next available TIL.
- This allows us to let the planner choose whether an action should happen before or after a TIL.
- POPF has some advantages here, as it only enforces orderings where they are strictly necessary.

The example from the previous section, following this method, adds a time-window action to the STN, as follows:



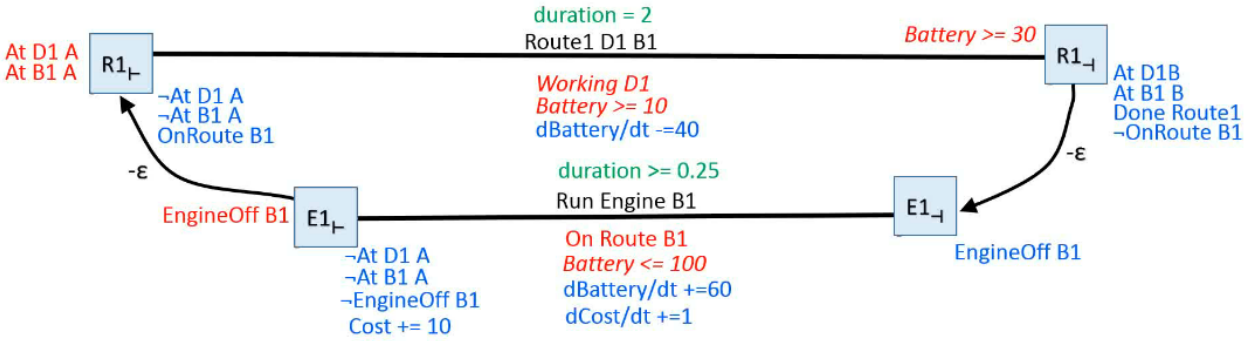
Continuous Linear Change

So far we've only encountered numeric quantities that change instantly, but what about continuous change? It can be implemented by discretising time (*see more: 'Discretise & Validate' Solving Approach, page 39*), but that requires reasoning at every time step (which gets harder and harder with more granular discretisations).

Another approach handles continuous change with **linear programming and LP solvers**. Note that this approach currently **works for linear change only**.

Note: expressions will be noted as $v' = vw + c$, meaning that the new value of v, v', is the current value multiplied by w, plus some constant c. v and w are vectors, the former containing variable values, the latter containing the coefficients of the expression.

Adding to The Routes Domain



The following conditions have been added to a battery introduced to the routes domain:

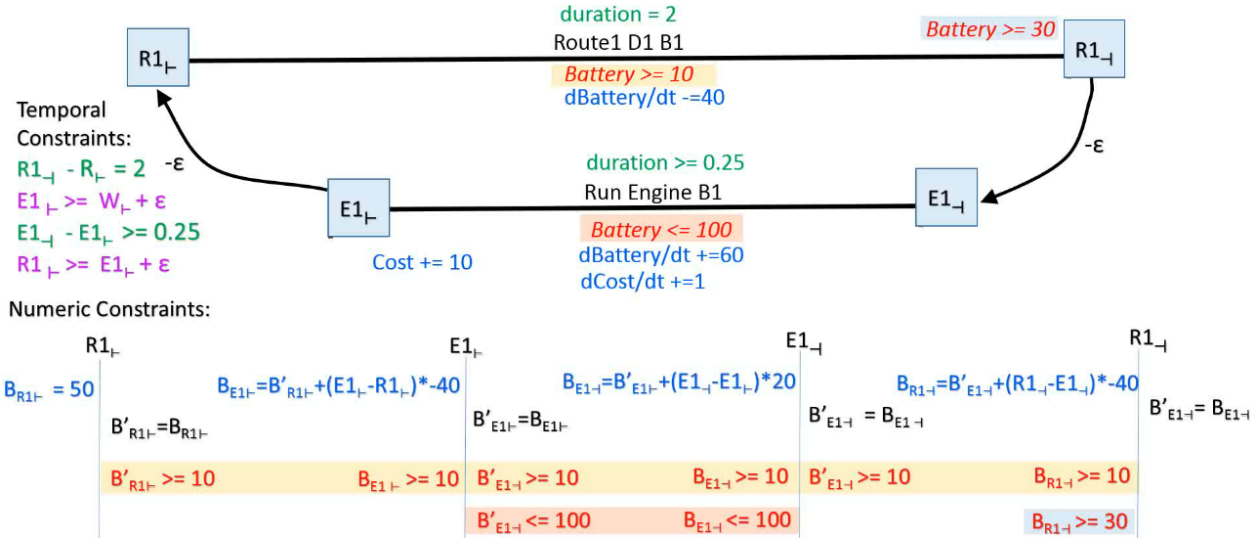
- Driving a route decreases the charge by 40 units/hour.
- Running the engine increases the charge by 40 units/hour.
- The bus must be returned with at least 30 charge units.
- The engine must be run for at least 15 minutes.
- Turning the engine on costs 10 immediately.
- Running the engine costs 1 per hour.
- The battery cannot be charged over 100 units.
- The battery cannot drop below a charge of 10 units.

Modelling as a Linear Program

We now need to assign timestamps to actions to **make sure that all of the numeric conditions are satisfied**, as well as the previous deadline and scheduling conditions. This is achieved using **LPs**:

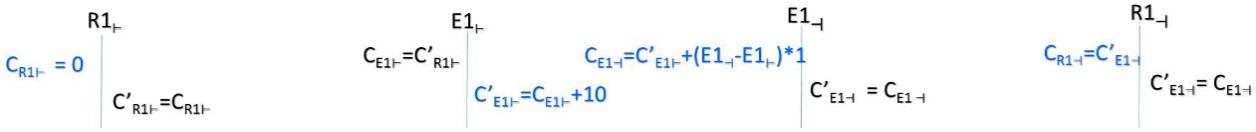
- The linear equations of the temporal constraints are written as an LP problem.
- Additional numeric constraints are added to the LP problem, by creating a variable for the value of every problem variable at every step in the plan.
  - Constraints are added to represent the changes to variables before and after each snap action, and to enforce overall constraints.
- A solution for the LP problem gives timestamp assignments to all of the snap actions.

Given the plan $R1_{\vdash}$ - $E1_{\vdash}$ - $E1_{\dashv}$ - $R1_{\dashv}$, the following variables and assignments represent the changes to the battery level caused by the actions, and the constraints that must be satisfied throughout.



- Equations before lines show the value immediately before the snap action.
- Equations after show the value immediately afterwards (used for instant effects, which are not used here).
- Black equations show unchanging, boring equations.
- Red values show the constraints that must be respected.
- $B'_{R1\vdash} \geq 10$ does not need to be checked before $R1\vdash$ because the action may achieve its own invariant.
- $B_{R1\dashv} \geq 30$ is checked before $R1\dashv$ because it is a precondition.

The same process is applied to model the cost of running the engine. This variable does use an instant effect, shown after $E1\vdash$.



Generalising

For each snap action A_i in the partial plan, create the following LP variables for each numeric variable v in the problem:

- v_i - the value of v immediately before A_i is executed.
- v'_i - the value of v immediately after A_i is executed.
- δv_i - the rate of change active on v after A_i is executed (this is a constant).

Create a single LP variable t_i to represent the time at which A_i will be executed.

Then, we add **constraints**:

- Initial values:
  - v_0 = the initial state value of v.
- Temporal constraints:
  - $t_i \geq t_{i-1} + \epsilon$
  - $t_j - t_i \leq max\_dur(A)$ where t_j is the end of an action starting at t_i.
  - $t_j - t_i \geq min\_dur(A)$ where t_j is the end of an action starting at t_i.
- Continuous change:
  - $v_{i+1} = v'_i + \delta v_i(t_{i-1} - t_i)$
- Discrete change:
  - $v'_i = v_i + wv_i$
- Preconditions:
  - Checked before their corresponding action.
  - $wv_i\{\geq, =, \leq\}c$
- Invariants:
  - Invariants of A must be checked before and after every step between the start (i) and end (j) of A.
  - $wv'_i\{\geq, =, \leq\}c$
  - $wv_{i+1}\{\geq, =, \leq\}c$
  - $wv'_{i+1}\{\geq, =, \leq\}c$
  - ...
  - $wv_j\{\geq, =, \leq\}c$

Linearity Assumption

This approach only works with linear change, and hence δv_i **is a constant**. If it was a function instead, it could mean multiplying linear variable by other linear variables, which is non-linear.

This is important, because invariants are only checked immediately before and after actions are applied - they are not checked at points between actions. This is okay, because is a precondition is satisfied at two points, its **linearity guarantees that it is satisfied at all points** between them. A non-linear function could go out of the precondition bounds and back in between the two points.

Objective Function

LPs have an objective function, which can be used to tune the plan:

- Want to minimise plan span?
  - Make a variable t_{now} and order it after all other steps in the plan (i.e. $t_{now} - t_i \geq \epsilon$ for all steps i).

  - Set the LP objective to minimise t_{now}.
- Want to minimise some cost function?
  - This could be a function of the final variable values.
  - Write an objective as a function of t_{now} (see above) for the final action in the plan.
  - E.g. minimise $3v_{now} + 2u_{now} - 0.5x_{now}$.

Note that this will optimise the plan that has been found, but **may not be optimal for the entire problem**.

Action Applicability

In general in discrete numeric planning we know the values of the variables at all states, because they are set in the initial state and changed by discrete amounts by every action.

With continuous change, the value of a variable depends on how much time we allow to elapse, even if we don't execute any actions. This means that we **cannot know a fixed value** for a given variable in a given state. However we **can know its range**, because we know its value when the state was entered, we know the maximum duration of the action, and we know how the value changes over time.

This is useful, because we may need to **wait in a state** for a precondition to be true.

Using LP to Find Value Bounds

In general it's not easy to know the bounds for a variable, even when all active change is known. We can use the LP to calculate bounds for us, with a small modifications:

- Add a variable t_{now} representing the time of the next action being applied.
- Add a variable v_{now} for each variable, representing its value at t_{now}.

For each variable, we set the objective function to:

- Maximise v_{now} to give the **upper-bound** on v.
- Minimise v_{now} to give the **lower-bound** on v.

We can now use the upper/lower bound to check satisfaction of $\geq/\leq$ conditions.

The action is **not guaranteed** to be applicable, because it may not be possible to simultaneously wait the necessary time to achieve the appropriate values for complex conditions like $2v + 3u \leq 6$. This method is still useful though, because it can be used to **prune states** arising from actions that are definitely not applicable.

Aren't LPs Expensive?

Yes, but they tend to be quite small and simple LPs that typically take **a lot less time than heuristic computation**.

Soft Constraints/Preferences

We can encode preference constraints in the same way as regular constraints. However, if they are not attainable then the **LP would become unsolvable**, even though the problem can be solved by violating the preferences.

To allow solving of problems with preferences, we need to use **mixed integer programming (MIP)**, using **big M** constraints.

For example, given the preferences:

- Preference 1: A happens before B.
  - $T_B - T_A \geq 0.01$
  - Violation cost: 5
- Preference 2: C happens by time 42.
  - $T_C \leq 42$
  - Violation cost: 2

To encode this with big M constraints:

- We need:
  - 0/1 integer variables per preference, p_1 and p_2.
  - A very large constant, M, large enough to dominate other values in the problem.
- The preferences become:
  - $T_B - T_A + Mp_1 \geq 0.01$
  - $T_C - Mp_2 \leq 42$
- We include the costs in the objective function:
  - Minimise $\ldots + 5p_1 + 2p_2$.
  - Maximise $\ldots - 5p_1 - 2p_2$.

p_1 and p_2 become boolean switches for satisfying or violating the preferences: if they are satisfied ($p_i = 0$) then M has no impact on the constraint and no cost is added; if they are violated ($p_i = 1$) then M will 'brute force' the constraints into being satisfied and the cost will be added to the objective function.

Aren't MIPs Expensive?

Yes, but they also tend to be quite small and simple MIPs that can be solved quickly.

Relationships Between Planners

- CRIKEY! = FF + STNs
- COLIN = CRIKEY!, swapping STNs for LPs
- POPF = COLIN + fewer ordering constraints
- OPTIC = POPF + preferences

UPMurphi

UPMurphi is a leading planner for hybrid problems that uses the discretise and validate approach. It creates the discretised problem, solves it with an explicit forwards search, and validates solutions with Val.

Timeline Discretisation

Timelines are discretised with the following UPMurphi rule:

```
rule "time_passing"
    (true) ==>
    begin
        apply_continuous_change();
        t := t + T;
    end;
end;
```

`t` is the current time, and `T` is the time step.

Action Discretisation

Action rules always check for events after they finish executing. The following PDDL...

```
(:action heat
 :parameters (?s — satellite)
 :precondition (and
    (> (energy ?s) 0)
    (heaterOff ?s)
 )
 :effect (and
    (heaterOn ?s)
    (not (heaterOff ?s))
 )
)
```

...becomes this UPMurphi rule:

```
ruleset s:satellite do
    rule "heat"
        (energy[s] > 0) && (heaterOff[s]) ==>
        begin
            set_heaterOn(s, true);
            set_heaterOff(s, false);
            event_check();
        end;
    end;
end;
```

Event Discretisation

At each clock tick, any event can be **fired at most once**. The event must delete its preconditions.

Process Discretisation

The continuous change within processes is handled by discretisation. The original continuous dynamics are discretised with **step functions** for resource values.

The following PDDL...

```
(:process warming
 :parameters (?s — satellite)
 :precondition (and
    (> (energy ?s) 0)
    (heaterOn ?s)
 )
 :effect (and
    (increase (temp ?s) (* #t (heatRate ?s)))
    (decrease (energy ?s) (* #t (heaterPower ?s)))
 )
)
```

...becomes this UPMurphi procedure:

```
procedure process_warming(s: satellite)
begin
    if (heaterOn[s] && energy[s] > 0) then
        temp[s] := temp_update_warming(temp[s], heaterRate[s])
        energy[s] := energy_update_warming(energy[s], heaterPower[s])
    end
end;
```

Process/Event Interaction

The `apply_continuous_change()` function mentioned earlier can be broken down as follows:

1. Call the procedures for all processors that are currently enabled.
2. Check if new events have been triggered and fire them if necessary.
3. If some event has fired, then:
   (a) Check the preconditions of all processes to see if some process has been enabled or disabled.
   (b) If so, repeat the whole sequence 1-3.
   (c) If not, end.

DiNo

DiNo is a **heuristic planner for non-linear PDDL+**. It is built on top of UPMurphi and based on the **discretise and validate** approach (*see more: 'Discretise & Validate' Solving Approach, page 39*). It's main extension from UPMurphi is the use of heuristic-guided search (specifically a modified EHC search). The heuristic used is a **staged relaxed planning graph (SRPG+)**.

Staged Relaxed Planning Graphs (SRPG+)

See more: Relaxed Planning Graph, page 11.

SRPG+ follows the same ideas as RPG:

- Fact/action layer structure.
- Delete effects are ignored.
- Helpful actions are derived.

SRPG+ explicitly represents **every fact layer with the corresponding time clock**, always separated by Δt.

Processes & Events

SRPG+ can handle process effects and events, which standard RPGs cannot. Process effects and events are relaxed in SRPG+ and handled in the same way as action effects:

- Continious process effects are handled similarly to durative action effects.
- Event effects are handled similarly to instantaneous action effects.

Relaxations

Standard RPG considers only predicates, so how can we relax other entities?

- Predicate actions are relaxed by ignoring their delete effects.
- Value actions are relaxed by operating on the bounds of a value, rather than reassigning it.
  - For example, if $v = 7$ in fact layer f_0, then action layer a_0 applies two actions with the effects of decreasing v by 2 and increasing v by 4, the value of v in fact layer f_1 can be anywhere from 5 to 11.
  - Value bounds only ever grow or stay the same; they never shrink.
- Processes are relaxed in the same fashion, by only operating on the bounds of their values. The discrete time step, Δt, is used between fact layers for computation.
- Events are relaxed by ignoring their delete effects.

Time Passing (TP)

This is an additional action added to the discretised model. It does three things:

- Advances time by Δt.
- Applies the (discretised) continuous change due to active processes.
- Applies changes due to triggered events.

SRPG+ can recognise **time passing** as a helpful action, identifying points where jumps forwards in time in the search can be made.

DiNo employs a time-passing pruning technique to drastically reduce the search space.