AIP: Artificial Intelligence and Planning

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 12

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:

```
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 5*). 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 5*). 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 12

See more: Enforced Hill Climbing, page 13

See more: Best-First Search, page 6

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

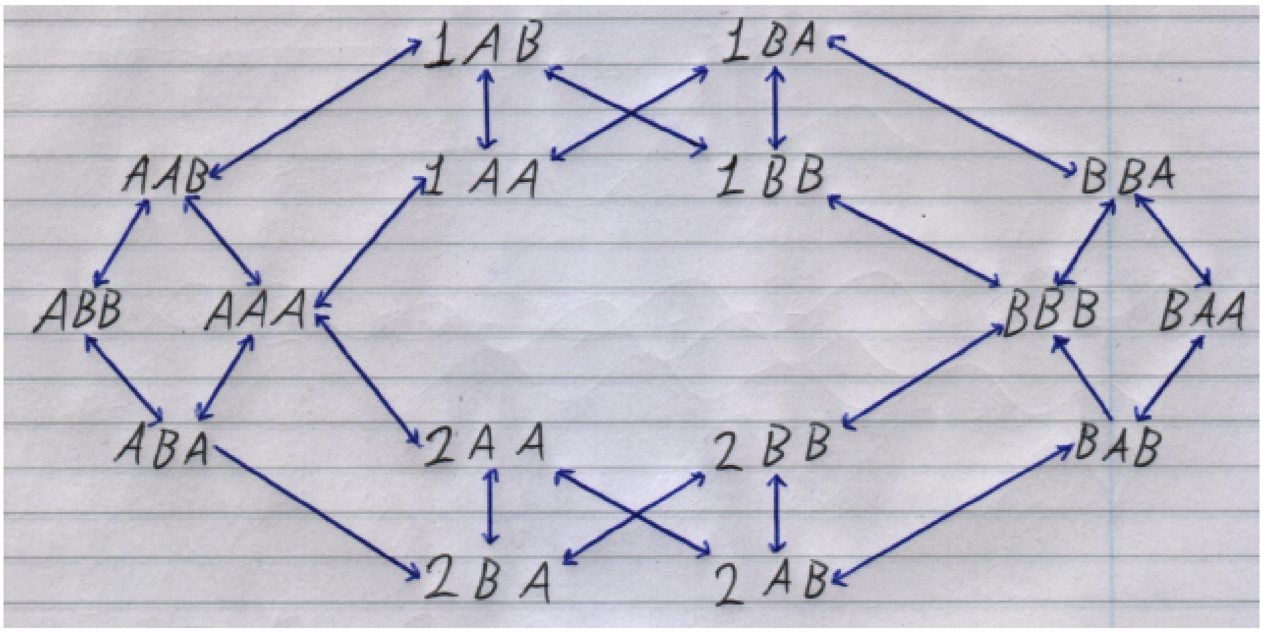
Pattern databases reduce the state search space by **combining states that match a given pattern**. The length of a solution for the reduced state space can be used as a heuristic for the original problem.

Example

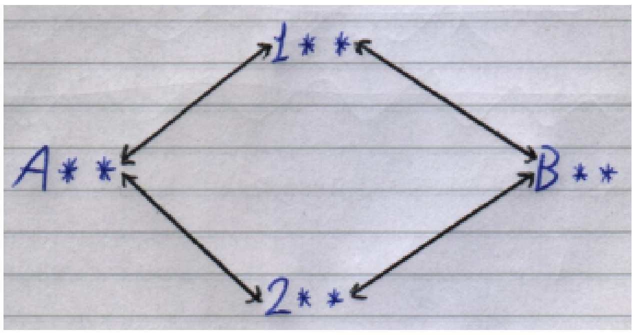
- Package: atA, atB, $inTruck1$, $inTruck2$
- Truck1: atA, atB
- Truck2: atA, atB
- *Actions*: drive, load, unload
- *Shorthand*: $AAB = packageAtA, truck1AtA, truck2AtB$

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

Entire state space:



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



Temporal Planning

We've looked at instant actions only so far, but doing things usually takes time - we need **temporal planning** with **durative actions** (actions that take time).

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 for the **start** of the action.
- Preconditions and effects that for 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:

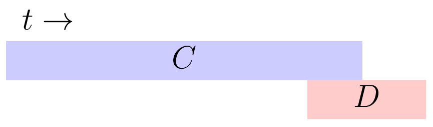
- Start a durative action, by...
  - ...applying a snap action $A_\vdash$ at time t
  - ...inserting $A_\dashv$ into Q at time $t + duration(A)$
  - ...advancing time, $S'.t = S.t + \epsilon$
- End a durative action, by...
  - ...removing and applying the first end snap action in Q
  - ...setting $S'.t$ 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 do forward-chaining search, but replace the priority queue with **separately managed temporal constraits**. 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 **sequence 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 $t = 0$.

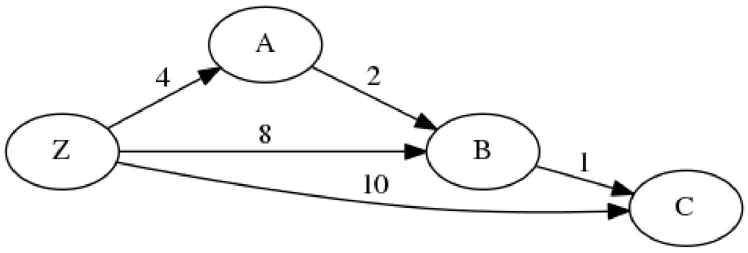
Latest Possible Times

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**.

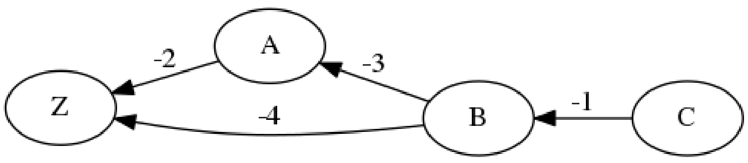
Earliest Possible Times

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**:



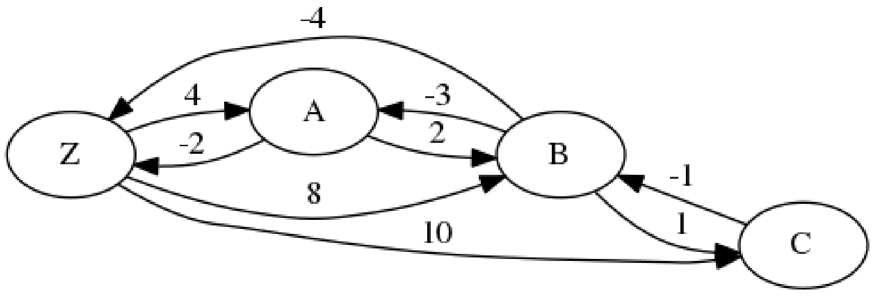
To find the **earliest** possible time: look for the **longest path**. Unfortunately graph algorithms usually find the 'shortest path', so we negate every constraint (multiply everything by -1, which also flips the sign) and then look for the shortest path (equivalent to the longest path of the original 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 $y \to x$ 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 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-Forg 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 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 17*.
- T - temporal constraints of the steps in P.

Note: Total Ordering 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 $B_{\vdash}$.
- $B_{\dashv}$ must precede $A_{\dashv}$ (i.e. A achieves some precondition of $B_{\dashv}$).

An ordering that places $A_{\vdash}$ before $B_{\dashv}$ 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 planned 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 the 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.