2020

Akshat Sood

[6CCS3AIP ARTIFICIAL INTELLIGENCE PLANNING]

Notes for 6CCS3AIP Artificial Intelligence Planning. Includes the following chapters: Planning Basics (Introduction to Classical Planning (Refresher), Introduction to Planning, Planning as Forward Search, Heuristic Search, PDDL Modelling Application Example - Scanalyzer), Classical Planning (Improving Heuristic Search, Relaxed Planning and RPG Heuristics, RPG Heuristic in the FF Planner, Landmarks for Planning, Searching with Landmarks, Dual Heuristics and Local Search Optimisation), Optimal Planning (SAS+ Planning, Pattern Databases, Cost Partitioning), Non-Forward Search (GraphPlan, Planning as SAT, Partial Order Planning (POP), Hierarchical Task Network (HTN) Planning), Planning with Uncertainty (Markov Decision Process (MDP), Partially Observable Markov Decision Process (POMDP)), Planning with Preferences (Numeric Planning, Planning with Preferences, LPRPG-P: Reasoning with Preferences), Temporal Planning (Introduction and Modelling in PDDL 2.1, Temporal Planning Challenges And Decision Epoch Planning, CRIKEY 3 And Simple Temporal Networks, The Temporal Relaxed Planning Graph Heuristic, The POPF (Partial Order Planning Forwards) Planner, Compression Safety, Planning with Deadlines) and Planning with Expressive World Models (Planning with Continuous Linear Change: COLIN, OPTIC: Combining Temporal Planning and Preferences, PDDL+: Processes, Events and Non-Linear Effects).

CONTENTS

INTRODUCTION TO CLASSICAL PLANNING (REFRESHER)

Planning is the process of taking an **initial state**, I, alongside a **goal state** G and a set of **actions** A, and finding the collection of actions (called a plan) that gets us from I to G as effectively as possible. Thus, a **plan** is a sequence of actions that bring the system from I to G.

The difference between planning and other search techniques is that in planning, the actions have additional constraints, i.e. the **preconditions** and **effects** of the actions.

- Preconditions: Facts that have to be true before the action can take place
- Effects: Encapsulate what happens by committing the action

If we consider more simple graph search techniques or even more advanced informed search such as A* search, it shares much of the same traits, in that we are looking to explain how a solution can be constructed by putting actions together that get us from an initial state to a goal. But as mentioned, these actions have preconditions and effects and this allows us to more accurately model when an action is feasible, but also what outcomes it will generate.

- **Problem Modelling**
  - Graph Search
  - Heuristic Search
  - Adversarial Search
- **Constraint Satisfaction Problems**
- **Automated Planning**
  - In general, planning is more powerful than CSP
  - You can see planning as multiple CSPs
  - Some planning problems can be modelled as a CSP (and vice-versa).

Constraint Satisfaction Problems: As an end-user we're not really interested in the path from the initial state to the answer, but it's important that we find the answer if it exists. Given these constraints of action preconditions and effects, it invokes some of the same principles and in-theory, we could model a planning problem as a CSP and indeed vice versa.

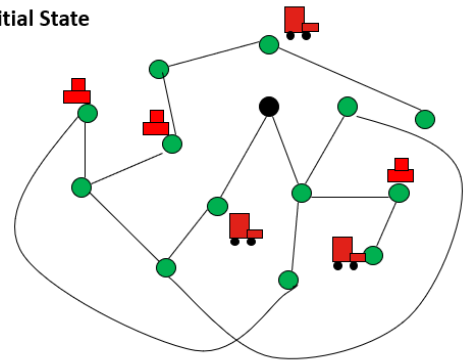
EXAMPLE – THE LOGISTICS DOMAIN

Consider the example of the Logistics Domain. Here we have a network of locations joined by roads, some cargo we which to deliver to specific locations and truck that are responsible for picking up, driving and dropping off said cargo.

Constraints:
Fuel
Truck capacity
Number of drivers
…
…

Cargos
Trucks
Destination point

Initial State



Goal: all cargos at a destination point

We need to consider the logic of how that would work. A truck needs to be in the location of cargo before it can collect the cargo. It needs to have that cargo in the truck in order to deliver it. So, for this to work as a planning problem, we need to encode all the logic needed for the search to do its work.

There are three critical pieces of information that we need to encode

- The **objects** involved
  - The Cargo $(c_1, \dots, c_n)$
  - The Trucks $(t_1, \dots t_2)$
  - The Locations $(l_1, l_2, \dots, l_n)$
- The **relationships** between these objects
  - Pairs of locations that can be connected: $connected(l_i, l_j)$
  - Each truck is at a specific location: $at(t_i, l_j)$

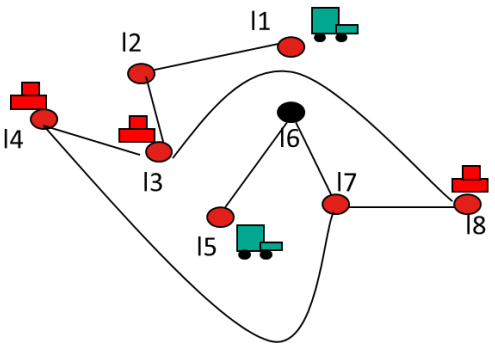
- Each cargo can be at a given location: $at(c_i, l_j)$
- Each cargo can be on a truck: $on(c_i, t_j)$

- Set of possible **actions** along with their preconditions and effects
  - Informally, we can state that there are 3 actions
    - A truck can move from one location to another ($move$)
    - A cargo can be loaded onto a truck ($load$)
    - A cargo can be unloaded from a truck ($unload$)
  - To model these unique actions, we need to consider the following things
    - Parameters: what objects are involved?
    - Preconditions: What needs to be true for the action to be applied?
    - Effects: How does the state change after execution?
  - Preconditions are described through a set of relations (that must be true in a state for the action to be applicable)
  - Effects are described by adding or deleting relations

Action	Parameters	Preconditions	Effects
Move	$truck\ t$ $location\ from$ $location\ to$	$at(t, from)$ $connected(from, to)$	$+\ at(t, to)$ $-\ at(t, from)$
Load	$truck\ t$ $cargo\ c$ $location\ l$	$at(t, l)$ $at(c, l)$	$+\ on(c, t)$ $-\ at(c, l)$
Unload	$truck\ t$ $cargo\ c$ $location\ l$	$at(t, l)$ $on(c, t)$	$+\ at(c, l)$ $-\ on(c, t)$

We must define the initial state and the goal state as well

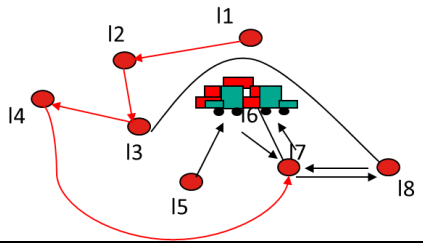
- **Initial State**: list of all the objects of the problem along with all the true relations among the objects. We can assume (**Closed-world Assumption**) that what is not listed is false.
  Consider the case where we have 2 trucks, 3 cargos and 8 locations
  - $at(c_1, l_3)$
  - $at(c_2, l_4)$
  - $at(c_3, l_8)$
  - $at(t_1, l_1)$
  - $at(t_2, l_5)$
  - $connected(l_1, l_2), connected(l_2, l_3), connected(l_3, l_4), connected(l_3, l_8),$
  - $connected(l_4, l_7), connected(l_5, l_6), connected(l_6, l_7), connected(l_7, l_8)$



- **Goal State**: List of all the relations that need to be true at the end of the plan. This only encapsulates critical information needed to solve the problem. Thus, information we express in the initial state, such as the topology of this road network, isn't relevant to the goal state.
  For the goal state in this case, we simply require all cargos to be at l_6.
  - $at(c_1, l_6)$
  - $at(c_2, l_6)$
  - $at(c_3, l_6)$

So, if we look at the initial state, we can create a **plan** by having both trucks work together.

- t_1 will move from l_1 through l_2 to l_3, load up the cargo c_1, drive to l_4, pick up cargo c_2 and then drive all the way around from l_4 to l_7 and then l_6 and unload c_1 and c_2 at the destination.



- t_2 drives from l_5 through l_6 and l_7 to l_8, loads up c_3, drives back to l_6 and unloads c_3 at the destination

This is one of several possible solutions to this problem, but if we count the total number of actions taken, its arguably going to be the most optimal one.

If we have a ridiculous number of locations, cargo, vehicles and maybe even additional vehicle types or constraints on driver availability and fuel costs, then we're going to need a **planner** to solve the problem.

PLANNERS

There are a very large number of planning systems out there, many of which devised by AI researchers to solve very specific issues within planning. In fact, this is the focus of the IPC: the International Planning Competition, which presents a series of challenging benchmarks for the planning research community to try and solve. Some of them are Zeno, FF, MetricFF, VHPOP, Marvin, Crikey, POPF, MIPS-XXL, LPGP, LPRPG, CoLin, Fast Downward, UPMurphi, Gamer and MBP.

In order to maintain some form of uniformity amongst all of them, there are a couple of common languages used within the planning community that are supported by these planners to allow you to take the same problems and apply them in different systems.

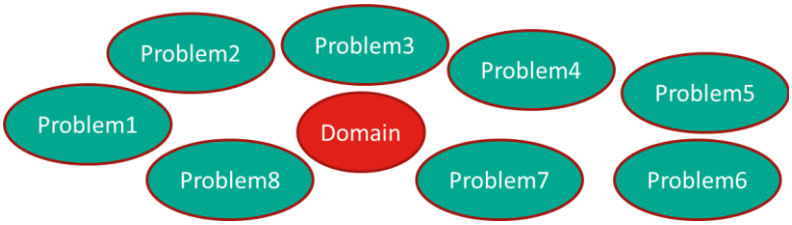
PDDL (PLANNING DOMAIN DEFINITION LANGUAGE) BASICS

It is the standard language spoken by the planners (i.e. if you write a correct model of your planning problem in PDDL then you can use existing planners to easily solve your problem). There are different versions (levels) of PDDL, each one able to handle different features. For example

- **PDDL**: Propositional Planning
- **PDDL2.1**: Numbers + Time
- **PDDL3**: Preferences
- **PDDL+**: Continuous Change, Processes, Events

A **PDDL Planning Model** is described through 2 files

- **Domain Files**
  - General description of the world
  - Which types of objects are involved
  - What actions can be applied etc.
- **Problem File(s)**
  - Specific situation you want to solve
  - List of objects, initial state, goal state

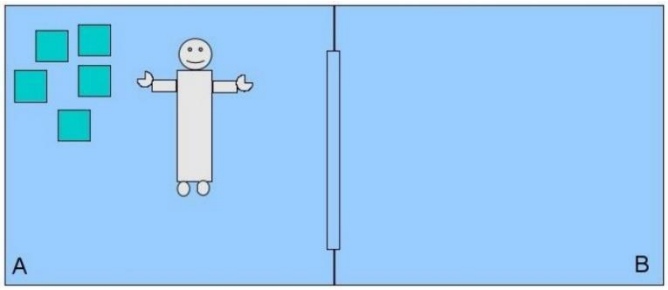


EXAMPLE: THE GRIPPER DOMAIN

Consider the following example. A robot can move between 2 rooms and pick up or drop boxes with either of his 2 arms.

- Initial State
  - Boxes $b_1, \dots, b_5$ in $room\ A$
  - $Robot$ in $room\ A$
- Goal State
  - Boxes $b_1, \dots, b_5$ in $room\ B$
  - $Robot$ in $room\ B$

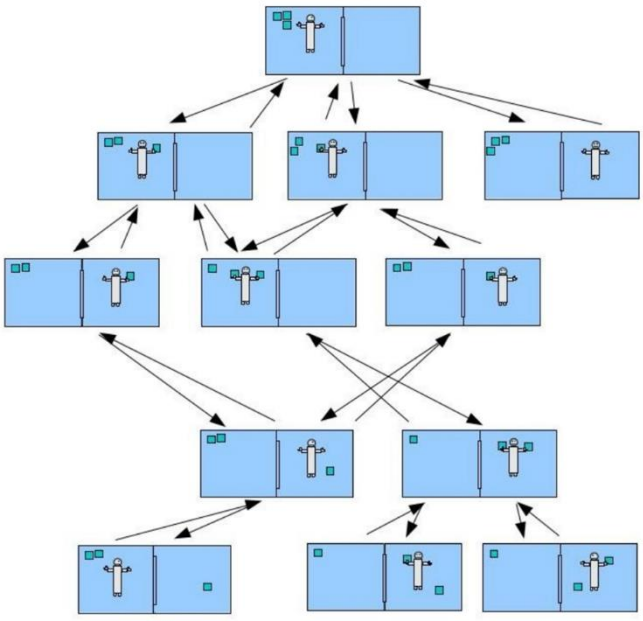
The robot can perform the following **actions**:



- Move from one room to the other
- Pick up a box in a gripper
- Drop a box, if it has a box in either of its arms

Action	Parameters	Preconditions	Effects
Move	$Room\ x$ $Room\ y$	$at-robot(x)$	$at-robot(y)$ $-at-robot(x)$
PickUp	$Room\ x$ $Ball\ b$ $Gripper\ g$	$at-robot(x)$ $at(b,x)$ $\boldsymbol{free(g)}$	$+\ carry(g,b)$ $-\ at(b,x)$ $-\ \boldsymbol{free(g)}$
Drop	$Room\ x$ $Ball\ b$ $Gripper\ g$	$at-robot(x)$ $carry(g,b)$	$+\ at(b,x)$ $+\ \boldsymbol{free(g)}$ $-\ carry(g,b)$

The description of each action is given in the table given alongside. It is quite similar to the logistics domain from the previous example, with one important distinction; the robot has limit to the number of boxes it can carry at a time. The robot can only carry one item in each gripper at a time. Hence, we have included a new relation, $free$, which expresses whether or not that gripper is free to pick up a particular object. This means that each gripper can only be used to carry one item at a time, unlike the truck in the logistics domain which only acknowledged whether a piece of cargo was on the truck or not.



So, taking this all into consideration, here's a graph showcasing a segment of the state space for a version of this problem with only 3 balls. Notice that the solution requires the robot to pick up one or two balls at once, move to the other room, drop them off and return to grab the remainder. This is where a more intelligent search algorithm can prove useful, given that in many of these states, progress is easily undone by an action being undone immediately after. We could pick up a ball, only to immediately drop it. Or move to the destination location only to move back to the original room before putting it down. So even a problem as small as this requires the planner to help optimise the solution.

THE GRIPPER DOMAIN IN PDDL

We start by defining the **objects** in the problem. In this example, we have 4 balls, 2 rooms (A and B) and we use $left$ and $right$ to denote the grippers on each arm of the robot.

These objects would be defined in the **problem file** as this is a specific problem permutation.

```
(:objects roomA roomB
    ball1 ball2 ball3 ball4
    left right)

(:predicates (ROOM ?x) (BALL ?x) (GRIPPER ?x)
    (at-robot ?x) (at ?x ?y)
    (free ?x) (carry ?x ?y))
```

Next, we can define the **Predicates**. Predicates in PDDL help us assign facts to objects or define the relations between objects in the domain. We would define these in the domain model.

Here we use three predicates `ROOM`, `BALL` and `GRIPPER` to define the three types of objects in the problem. Typing object like this is something later PDDL versions provide as a feature, but for now were going to use this to capture in the action preconditions that a variable is of a certain type.
There are 2 other predicates for capturing the location of objects. First the robot with `at-robot`, and then the ball with `at`. The reason why we have a specific predicate for the robot is because there is only 1 robot in the domain. If there would have been multiple robots in the domain, then the at predicate would have been used.

In addition to this, we have 2 other predicates `free` and `carry` which give information about whether the robot is carrying any boxes or whether the arm is free.

Next if we consider the problem from earlier, here is how we would encode that **initial state**, using the objects from the previous slide. This helps define the rooms, the balls and the two grippers. But also in the initial state we declare that all balls are in room A and that both grippers are free and not holding anything.

```
(:init (ROOM rooma) (ROOM roomb)
    (BALL ball1) (BALL ball2)
    (BALL ball3) (BALL ball4)
    (GRIPPER left) (GRIPPER right)

    (free left) (free right)
    (at-robot rooma)
    (at ball1 rooma) (at ball2 rooma)
    (at ball3 rooma) (at ball4 rooma))
```

```
(:goal (and (at ball1 rooma)
    (at ball2 rooma)
    (at ball3 rooma)
    (at ball4 rooma)))
```

Then the goal state is for all the balls to be in room B. Note the use of the `and` statement to act as a Boolean conjunction, stating that all these facts need to be true.

The move action reads straightforward; it carries two parameters `x` and `y`, both of which are rooms. The robot has to be at the first room, `x` as a precondition and the effect is that the robot is now at `Y`, but it is also no longer at `X`. This requires use of the `not` term in the language to state that this is not true anymore.

```
(:action move
 :parameters (?x ?y)
 :precondition (and (ROOM ?x) (ROOM ?y)
                    (at-robot ?x))
 :effect (and (at-robot ?y)
              (not (at-robot ?x))))
```

```
(:action pick-up
 :parameters (?x ?y ?z)
 :precondition (and (BALL ?x) (ROOM ?y) (GRIPPER ?z)
                    (at ?x ?y) (at-robot ?y) (free ?z))
 :effect (and (carry ?z ?y) (not (at ?x ?y))
              (not (free ?z))))
```

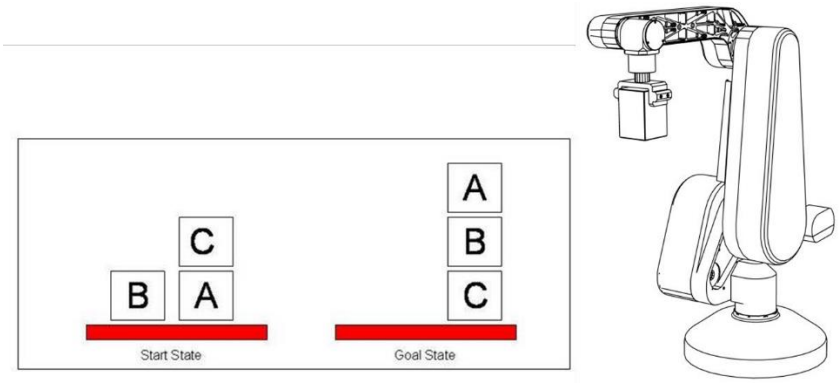
Our second action is pick-up. There are three parameters, `X`, `Y` and `Z`. `X` is a ball, `Y` is a room and `Z` is a gripper. In order for the `pick-up` to work correctly. The ball `X` has to be in room `Y`. The robot has to be in room `Y` and the Gripper has to be free. Note again another interesting situation that we simply assume the gripper is always where the robot is, given it is only coming in to play when the robot is in a given location. And the effects result in the gripper carrying the ball, the ball no longer being in the room (a weird distinction, but it's common) and that the gripper is no longer free to carry anything else.

There is also `drop`, where the robot is already carrying the ball and is in the room. Resulting in the ball being in the room, the gripper once again being free and not carrying anything.

```
(:action drop
 :parameters (?x ?y ?z)
 :precondition (and (BALL ?x) (ROOM ?y) (GRIPPER ?z)
                    (carry ?z ?x) (at-robot ?y))
 :effect (and (at ?x ?y) (free ?z)
              (not (carry ?z ?x))))
```

So when you put the predicates and actions together, this creates the complete gripper domain in PDDL. Allowing us to then create any permutation of balls and rooms for the robot to move around and solve.
By keeping it all in the domain file, we can refine the model here and the create any number of problems we want that stress the limits of the domain and the planners that will try and solve them.

EXAMPLE – BLOCKS WORLD DOMAIN



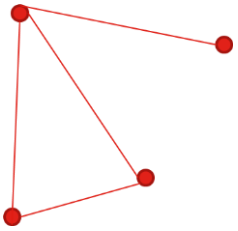

```
(define (domain blocks)
  (:predicates (on ?x ?y) (ontable ?x) (clear ?x) (handempty) (holding ?x))
  (:action pick-up
         :parameters (?x)
         :precondition (and (clear ?x) (ontable ?x) (handempty))
         :effect (and (not (ontable ?x)) (not (clear ?x))
                      (not (handempty)) (holding ?x)))
  (:action put-down
           :parameters (?x)
           :precondition (holding ?x)
           :effect (and (not (holding ?x)) (clear ?x) (handempty)(ontable ?x)))
  (:action stack
           :parameters (?x ?y)
           :precondition (and (holding ?x) (clear ?y))
           :effect (and (not (holding ?x)) (not (clear ?y)) (clear ?x)
                        (handempty) (on ?x ?y)))
  (:action unstack
           :parameters (?x ?y)
           :precondition (and (on ?x ?y) (clear ?x) (handempty))
           :effect (and (holding ?x) (clear ?y) (not (clear ?x))
                        (not (handempty)) (not (on ?x ?y)))))
```

PDDL2.1 FLUENTS (TEMPORAL PLANNING DOMAINS)

Original PDDL doesn't handle the idea of resource consumption or the passage of time, i.e. all actions take uniform time and have no costs.

PDDL 2.1 is an extension to PDDL for expressing temporal planning domains. It introduces the ability to model numeric values through **fluents**. We can identify a **resource** within the domain by having action effects increase or decrease the **value** of that resource. This more accurately helps maintain an understanding of the **costs** of actions.

Example – Metric Vehicle Domain



This domain is designed to allow for the driving of a vehicle along a road network, much like that seen in the logistics domain. But this time, we're capturing the fuel required to drive around.

```
(define (domain metricVehicle)
(:requirements :typing :fluents)
(:types vehicle location)
(:predicates (at ?v - vehicle ?p - location)
             (accessible ?v - vehicle ?p1 ?p2 - location))
(:functions (fuel-level ?v - vehicle) (fuel-used ?v - vehicle)
            (fuel-required ?p1 ?p2 - location) (total-fuel-used))
(:action drive
    :parameters (?v - vehicle ?from ?to - location)
    :precondition (and (at ?v ?from) (accessible ?v ?from ?to)
             (>= (fuel-level ?v) (fuel-required ?from ?to)))
    :effect (and (not (at ?v ?from)) (at ?v ?to)
            (decrease (fuel-level ?v) (fuel-required ?from ?to))
            (increase (total-fuel-used) (fuel-required ?from ?to))
                    (increase (fuel-used ?v) (fuel-required ?from ?to)))
)
)
```

In this example, the **`requirements`** tag is used to identify which features of the language we're going to use. In this case we are using **`typing`** and **`fluents`**.

- Typing Requirement: Able to identify specific variable types. Looking back at the Gripper domain example, we had to use predicates to identify whether a variable was a gripper, a room or a ball. Using typing, we can clearly denote that there are 2 types, vehicles and locations, and we can enforce this in the predicates themselves.
- Fluents Requirement: Allows us to define functions. These can be configured in the problem file or are a part of the preconditions and effects of specific actions. In this example we have 4 unique functions that revolve around the fuel consumption of a given vehicle
    - Current fuel level of the vehicle
    - How much fuel a given vehicle has used
    - Amount of fuel required to travel between locations
    - Total amount of fuel used (useful if there are multiple vehicles)

If we consider the **`drive`** action, it now not only considers whether it is possible to travel from A to B, but also how much fuel it costs to make the journey and whether the vehicle has sufficient fuel to make the trip. Once executed, it decreases the fuel level of the vehicle by the amount required of the journey, but also increases the fuel used by thus vehicle and the total amount of fuel used overall.

Now that we have defined the domain, let's consider the **problem file** (given below). In this problem file, we have 2 vehicles (a truck and a car) and four locations (Paris, Berlin, Rome and Madrid). We need to define the fuel for each vehicle, as well as the other fuel metrics that we wish to use.

We use fluents defined in the domain to enforce several elements of the problem, like the following

- The fuel levels of the truck and car are defined at a maximum level of 100

- The amount of fuel required to travel between each location is fully defined. For example, travelling from Paris to Berlin costs 40 units of fuel, while travelling from Berlin to Rome requires only 30 units
- The total fuel used is assigned to 0, as is the fuel used by the car and truck at the beginning of the problem

```
(define (problem metricVehicle-example)
    (:domain metricVehicle)
    (:objects truck car - vehicle
          Paris Berlin Rome Madrid - location)
    (:init
           (at truck Rome)
           (at car Paris)
           (= (fuel-level truck) 100)            (= (fuel-level car) 100)
           (accessible car Paris Berlin)         (accessible car Berlin Rome)
           (accessible car Rome Madrid)          (accessible truck Rome Paris)
           (accessible truck Rome Berlin)        (accessible truck Berlin Paris)
           (= (fuel-required Paris Berlin) 40)   (= (fuel-required Berlin Rome) 30)
           (= (fuel-required Rome Madrid) 50)    (= (fuel-required Rome Paris) 35)
           (= (fuel-required Rome Berlin) 40)    (= (fuel-required Berlin Paris) 40)
           (= (total-fuel-used) 0)   (= (fuel-used car) 0)    (= (fuel-used truck) 0))
     (:goal (and (at truck Paris)  (at car Rome)))
     (:metric minimize (total-fuel-used)))
```

In the goal, we also express that we want the truck and car to be in specified locations. We also specify a metric which enforces that we wish to minimize the total fuel used. Hence not only do we want to find a valid solution, we want one that is the most fuel efficient based on the values provided.

PDDL 2.1 – DURATIVE ACTIONS

Fluents allow us to handle resources and their ability to change throughout the execution of a plan. However, we still need to handle the passage of time, i.e. how long does an action take? This can influence conditions as well as effects of actions.

- Conditions might hold true at different times
- Effects can occur at different times

EXAMPLE – SLOW ELEVATOR ACTION

Consider the example given alongside. It is an action for moving an elevator slowly.

```
(:durative-action move-up-slow
    :parameters (?lift - slow-elevator ?f1 - floor ?f2 - floor)
    :duration (= ?duration (travel-slow ?f1 ?f2))
    :condition (and (at start (lift-at ?lift ?f1))
                    (at start (above ?f1 ?f2))
                    (at start (reachable-floor ?lift ?f2)) )
    :effect (and (at start (not (lift-at ?lift ?f1))))
                 (at end (lift-at ?lift ?f2)))
```

We can break down the action into the following components

- **Parameters**: We have taken 3 parameters; 1 `slow-elevator` and 2 floors (`f1` and `f2`)
- **Duration**: Since this is a `durative-action`, we can also specify the duration of the action. This duration can be described in the problem file (as shown alongside). This is predicated by the = sign within the brackets, and as we can see in the example given alongside, it takes 12 time units for us to travel from `f0` to `f1`.

Problem File

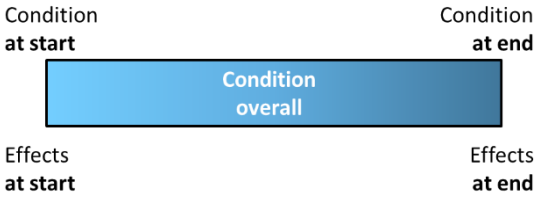
```
(above f0 f1)
(reachable-floor slow1 f1)
(= (travel-slow f0 f1) 12)
```

- **Condition**: This is the biggest change from the previous examples; we no longer have preconditions, instead we have conditions. This is because till now, preconditions were facts that needed to be true before the action can be selected, but now as we model the passage of time, we now have more granularity. We may have a predicate that is only true at the beginning (before the action executes), or perhaps it needs to be true throughout the execution. In this instance, we have 3 conditions that now use the `at-start` notation.
  - The lift has to be at floor f1
  - The destination floor f2 needs to be above f1
  - The destination is reachable using this lift (different lifts may only stop at specific floors of the building)
- **Effect**: Now we can specify effects which take place at the start of the execution of the action and effects which take place once the action has executed
  - Effects `at-start`: Once the action starts executing, we say that the lift is no more at floor `f1`.
  - Effects `at-end`: The lift arrives at the destination floor `f2`

Consider another action within the same domain; `board`. This action allows passengers to board onto the elevator.

```
(:durative-action board
    :parameters (?p - passenger ?lift - elevator ?f - floor)
    :duration (= ?duration 3)
    :condition (and  (over all (lift-at ?lift ?f))
                     (at start (passenger-at ?p ?f))
                     (at start (< (passengers ?lift) (capacity ?lift))) )
    :effect (and (at start (not (passenger-at ?p ?f)))
                 (at end (boarded ?p ?lift))
                 (at start (increase (passengers ?lift) 1)) ))
```

- **Duration**: Unlike the previous action, this action has a fixed duration of 3 time units.
- **Conditions**:
  - We have a new type of condition called `over all`. This means that this particular condition needs to stay true for the entirety of the execution of the action. In this instance the lift needs to stay at the floor from which the passenger is boarding. This makes a lot of sense given we want to ensure that the lift doesn't try to move, since that could injure or even kill someone if we're not careful.



  - There are 2 more conditions with the at-start notation: the passenger needs to be on the same floor as the elevator to board and the there should be space on the lift to board.
- **Effect**: The effects are only at the start and the end of the actions
  - At the start of the action, the passenger is no longer on the floor where the elevator is and the number of passengers in the elevator increases by 1
  - Once the action ends, the passenger has successfully boarded the lift

Problem File

```
(above f0 f1)
(reachable-floor slow1 f1)
(= (travel-slow f0 f1) 12)
(passenger-at p0 f1)
(= (passengers slow1) 0)
(= (capacity slow1) 4)
```

FORWARD CHAIN PLANNING

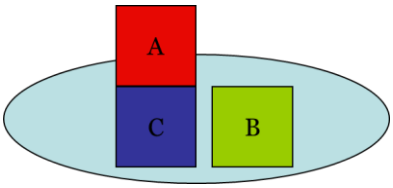
Forward chain planning involves the following steps

- Start at the initial state of the problem
  - Choose a state s and expand it. Beginning with the initial state s_{init}
- Consider all actions available and which ones are applicable given their preconditions
  - Consider the actions applicable in s. Examine the actions whose preconditions are satisfied
- Apply an action and observe the changes to the state. Given the effects of the action either add or remove facts from the state.
  - One successor state s' is generated for each applicable action a.
  - $s' = s - \{\text{Delete effects of } a\} + \{\text{Add effects of } a\}$
  - Create a plan for the state s' by chaining a to the end of the plan for s.
- Continue until we find a state that satisfies our goal conditions.
  - A goal state is a state containing all the goal conditions. If a state does not satisfy all the goal conditions, loop and expand more states.

The planner is going to generate the final plan, which is a sequence of actions that gets us from the initial state to the goal. But we still need to consider what actions are the best ones to take in a given situation, since we need to try and estimate the value of a given action in getting us to the goal.

Consider the `BlocksWorld` domain. We have the action `Put`, which is going to allow us to put object `X` onto `Y` from `Z`. In the diagram we have already put `A` on `C` from the table. To do this we, we need to ensure that `X` is clear, `X` is on `Z`, and `Y` is clear. This action has the following add effects

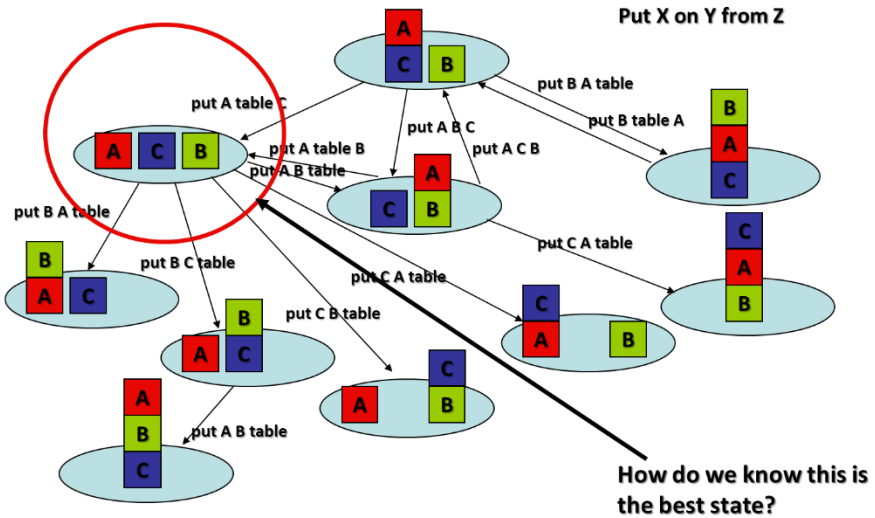
- `Z` is now clear
- `Z` is now on `Y`



And the following delete effects

- `X` is not on `Z` anymore
- `Y` is not clear anymore

If we consider the initial state, with `A` on `C`, `B` on the table, and we want them in order from top to bottom, we need to consider all of the possible actions. So if we expand all valid actions on the initial state, we can put `A` on the table, put `A` on `B` and `B` on `A`. Now putting `A` on the table is in fact the best possible action, given that as we roll out the actions for the other two successor states, we see this actually puts us in increasingly undesirable states. As we start to roll out from the state with all three on the table, we can see how quickly we still find ourselves generating undesirable states, but also some really good ones and ultimately find the goal.

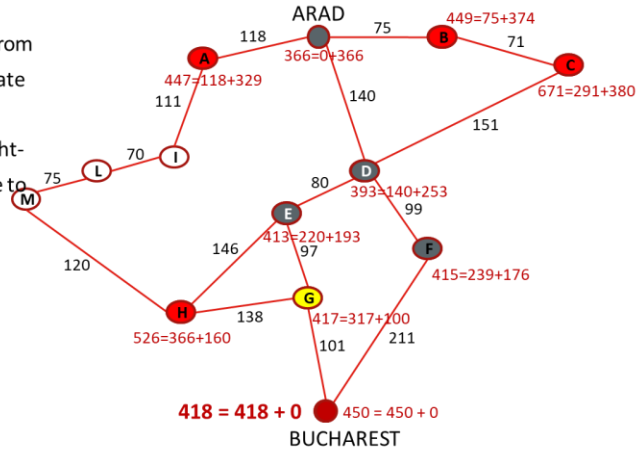


So from out casual observation and visualization of the states, we can see which one is the best one to prioritize, but how does a planner figure this out?

A* SEARCH

Typically, we would utilize a heuristic, used in the likes of A* search to help prioritize the best decision to take. A heuristic combined with the cost of an action – if we have that encoded in the problem – will help us more readily identify the best decision to take. In this example, the value of a state is calculated by both the cost to reach that state $g(n)$, plus the heuristic value $h(n)$. In this cases, we have pre-calculated the straight-line distances from each location to the goal of Bucharest. Now this presents a problem, the heuristic is domain dependent: we're creating a heuristic for this specific problem and it will only work in this problem and others in this road network domain.

- ***f(n) = g(n) + h(n)***
  - ***g(n)*** = cost from the initial state to ***n***
  - ***h(n)*** = straight-line distance to goal.



Straight-line distance from Bucharest

ARAD	366
A	329
B	374
C	380
D	253
E	193
F	176
G	100
H	160
I	244
L	241
M	242

This doesn't work for planning, given the idea of PDDL is as follows

- A planner should be able to receive any PDDL domain and begin to solve problems for it if they are correctly defined.
- We don't define a heuristic for the domain in PDDL. In fact, the person writing the domain might not even know what the heuristic should be for this problem.
-

DOMAIN INDEPENDENT PLANNING

So how can a planner make intelligent search decisions without being given a handcrafted heuristic? Is it possible to define heuristics that work for any PDDL domain? Yes – Domain Independent Heuristics; no knowledge about the specific problem is used.

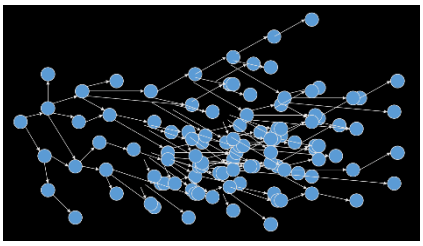
So with that in mind, let's consider the **Fast-Forward** or **FF** planner, developed in 2001 by Jorg Hoffmann and Bernhard Nebel. FF uses the relaxed planning graph or RPG heuristic, which is based on an existing planning system called GRAPHPLAN, which dates to 1995. RPG assesses how good a state is by calculating how many actions it would take to solve the problem from that state. Now if you do this on the original problem this is no different than just solving the problem. So the system works on what is known as the relaxed version of the problem. We take some of the complexity out of the domain and solve an invalid, incorrect version of the problem. But in doing so, as we'll see in the next chapter, we come up with a really good system for a heuristic of how close we actually are to the goal without factoring any domain specific information.

- **FF is a forward-chaining heuristic search-based planner**
- **FF uses the Relaxed Planning Graph (RPG) heuristic to guide search**
  - This involves finding a plan from the current state S which achieves the goals G but ignores the delete effects of each action
  - The length of this plan is used as a heuristic value for the state S

INTRODUCTION TO PLANNING AND PDDL

Planning in general is the process of deciding a sequence of steps to achieve an objective. It's a good thing to think about what it is that you're going to do, before you start doing it.

Artificial Intelligence Planning is the science of trying to design some software systems that can automate this process of planning, in a generic way, i.e. a domain independent way.



The challenge of planning is that we have a lot of choice in terms of steps that we can take. Depending on which choices we make, we are presented with a new set of choices, and so on. This leads to what we call a **Combinatorial Explosion**. Each time we make a decision, we are presented with another one. This leads to many possible paths that we can take.

PLANNING PROBLEMS

Planning problems are defined by a number of things, which are as follows. The example we will consider is of Amazon delivering a DVD to my house.

```
(at mydvd amazon)
(at truck amazon)
(at driver home)
(path home amazon)
(link amazon london)
(link london myhouse)
```

- An **Initial State** $\boldsymbol{I}$, which is the current state of the world. It is a collection of facts that are true in the world. The initial state for the example is a list of facts as shown alongside.
  In classical planning, any fact which is not listed as explicitly being true in the initial state, can be safely assumed to be false (*Closed World Assumption*)
- A **Goal** $\boldsymbol{G}$, this is what we desire to be the case in the future. The goal can be a conjunction of several facts, but in this example, the goal is just a single fact. The example given alongside is not a *Goal State* as we are not specifying facts regarding other items in the world. Thus for us, any state where the DVD is at my house can be considered to be a goal state.

```
(at mydvd myhouse)
```

- Some **Actions** $\boldsymbol{A}$, that we can apply to the world. These actions are defined according to the **domain**. They tell us what activities we can use in order to construct our plan.
  - We als need to know dictate what happens when we can perform these actions and what happens when we execute these actions respectively.
  - **Preconditions**: Conditions which must be true, in order for us to execute an action, i.e. we can only execute the action in that state, if these facts are true in that state.
  - **Effect**: The modifications made to the facts of the state on execution of the action.

```
(load mydvd truck depot)
(walk driver home amazon)
(board-truck driver truck amazon)
(drive-truck driver amazon london)
...
```

```
(:action load
      (:parameters (?d - item ?t - truck ?p - place)
      (:precondition (and (at ?t ?p) (at ?d ?p)))
      (:effect (and (not (at ?d ?p))
                    (in ?d ?t))
   )
)
```

  - Note that in the example given alongside, we say the the package is not at the location anymore, even though the truck is at the location and the package is in the truck. This is done to avoid confusion. Consider the scenario where another truck comes in (T_2) and tries picking up the package (which is already in the truck), it would be successful in doing so as the package is still at the location and the preconditions are met.

- This action is called a **Lifted Action** as it is parameterised.

Static Fact is a fact which is never added or deleted by any actions, and thus always remains true during the duration of the planning problem.

PLANNER

A planner is a piece of software, which takes initial state I, goal G and actions A to give us a **plan**. A plan can be defined as follows:

- An ordered sequence of actions from A, that will turn I into G.
- It is a set of instructions which tells us *what* to do, and *when* to do it, in order to achieve our goal.

It does so by performing a **search** to consider the different possible plans available, until a plan from I to G is found.

PLANNING MORE FORMALLY

A classical (or STRIPS) planning problem are problems where all states are collections of facts, or preconditions are conjunctions of facts and the effect is either add or to delete one or more facts. A classical planning problem is a tuple $\langle P, A, I, G \rangle$ where:

- P is a finite set of propositions, which consists of all the facts that we might consider in the planning problem, i.e. facts in the initial state, preconditions, goal or effects of actions.
- A is a set of actions each of which has the following
  - A Precondition, a set of propositions that must be true in order for A to be executed
  - Add Effects: a set of propositions added that become true upon execution of A
  - Delete Effects: A set of propositions that are deleted upon execution of A
- I a finite set of propositions representing the initial state
- G the goal, a set of propositions that must be true for a state to be considered the goal state.

The logistics example we considered earlier, seems simple since it is a familiar plan, but to a computer, it is just a set of facts. Consider the problem given alongside.

The solution does not seem obvious to us, but the computer will still look at it like a list of facts.

- Initial: (nostranger tolove), (knows rules)
- Goal: (so do I)

Action	Preconditions	Effects
Never	(full commitment) (knows rules)	(not (full commitment)) (so do I)
Gonna	(knows rules)	(not (knows rules)) (any other guy)
Give	(any other guy)	(not (any other guy)) (knows rules)
YouUp	(nostranger tolove) (any other guy)	(not (nostranger tolove)) (full commitment)

PLANNING AS FORWARD SEARCH

This is one of the most popular and intuitive planning algorithms and involves forward searching from the initial state. The idea is to search over a space of states. Each state is a collection of facts. To create new states, we consider the actions whose preconditions have been satisfied by the current state, and apply those actions to the state, which leads to more states (with delete effects removing some facts and add effects adding some facts).

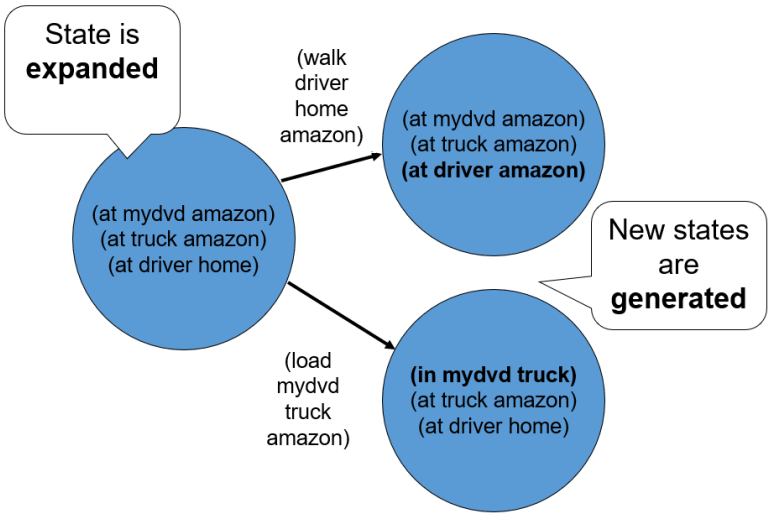
In the example given alongside we have the following facts in the first state

- The DVD is at Amazon
- The Truck is at Amazon
- The Driver is at Home

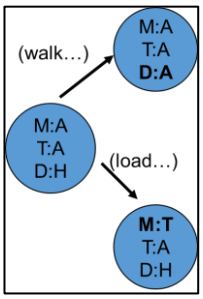
This allows us to perform the following 2 actions as their preconditions have been met by the current state

- Walk Driver Home Amazon (since the Driver is at Home)
- Load DVD Truck Amazon (since the DVD and Truck are both at Amazon)

This leads to 2 newly generated states, with the effects of the actions applied to each state individually.



In the example given alongside, the names have been abbreviated. *M* stands for MyDVD, *A* stands for Amazon, *T* stands for Truck, *D* stands for Driver and *H* stands for Home. So the example given above can be abbreviated to the one shown alongside.

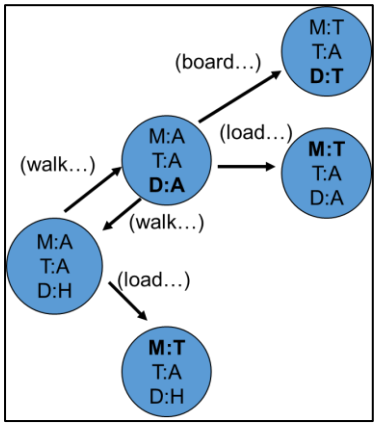


Thus, there are 2 possible actions,

- Have the driver walk from home to Amazon
- Have the DVD loaded onto the truck

Expanding on the first option (the driver walking from home to Amazon), we can apply the following 3 actions (shown alongside)

- The Driver can walk back Home
- The Driver can board the Truck
- The DVD can be loaded onto the Truck

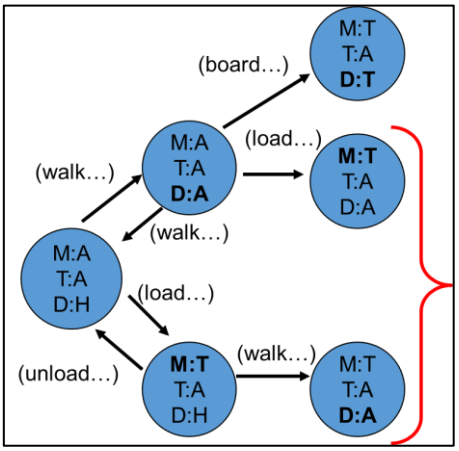


Even though it seems counterintuitive to have the Driver back Home as it undoes the action we have just performed, but for the planner, each state is just a set of facts, and it tries to apply all possible actions to it.

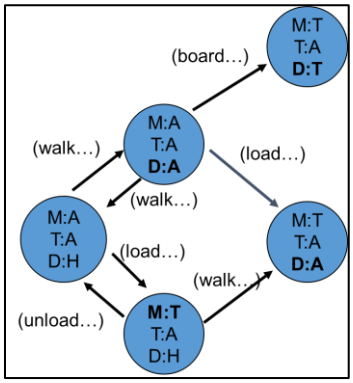
Considering the other branch of states which was created from the initial sate, we can apply the following actions to the new state.

- We can unload the DVD from the Truck and go back to the initial state
- We can have the Driver walk from Home to Amazon

It is important to note that the 2 states marked with red in the diagram shown alongside are the same states, as they contain the same set of facts. This means that performing the *walk* action followed by the *load* action, is the same as performing the *load* action followed by the *walk* action.



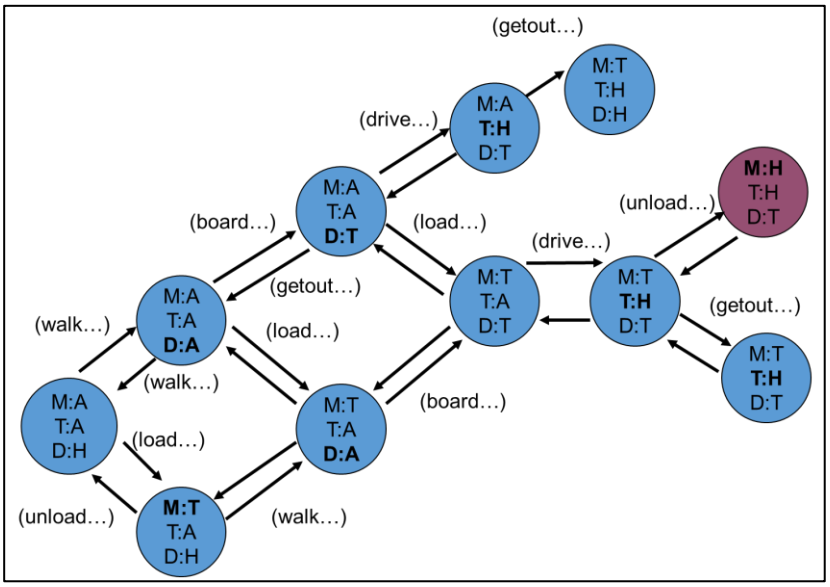
The diagram is no more a tree but is now a directed graph (as shown alongside), as we've come to the same state via 2 different paths.

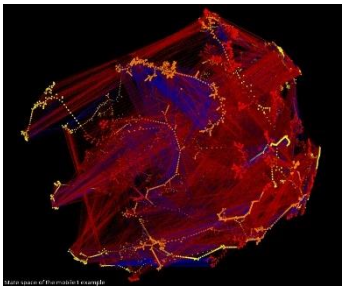


On expanding the graph further, we get a bigger directed graph until we get the goal state (highlighted in purple), where the DVD is at my house.

Once we get the goal state, we can backtrack and find the path we took to get to the solution. In this case the path would be as follows:

- Driver Walks from Home to Amazon
- Driver Boards the Truck at Amazon
- The DVD is Loaded onto the truck (these 3 steps can be done in any order, and will all lead to the same state)
- The Driver Drives the Truck from Amazon to My Home
- The DVD is Unloaded from the Truck and is now at My Home





This was a pretty simple example with only a few states to consider. In practice, these graphs can become quite complex, which is what makes planning difficult. Given alongside is a 3d depiction of one of these complex planning problems.

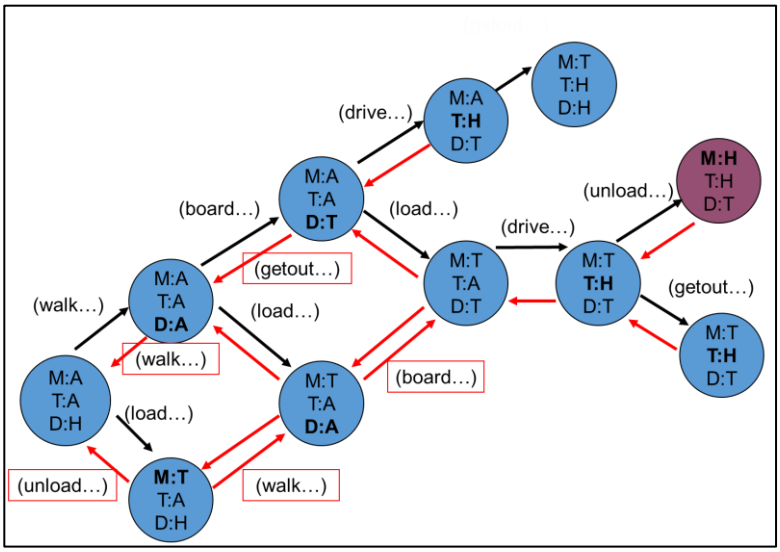
NOTES ON FORWARD SEARCH

Don't Go Around in Circles!

We don't want to be considering states that we have already been in. So when we expand on a state, and see a successor we've already seen before, we can ignore it.

- Keep a closed list which contains all the states that have been seen before
- When expanding a state, ignore the successors on the closed list

Consider the previous example, by maintain the closed list, we can eliminate all the actions highlighted in red in the diagram given alongside, as their successor states have already been observed before.



Forward Search, Graphs and Trees

By eliminating the states with the closed list, we've now converted the directed graph intro a **tree**.

- Only one path into each node is kept
- No backwards edges

FORWARD SEARCH ALGORITHMICALLY

Forward search is exactly what we've been doing till now, while maintaining a **Closed List** (with all the states that have already been dealt with) and an **Open List** (with states that we have yet to deal with). Thus the initially the lists would look like as follows

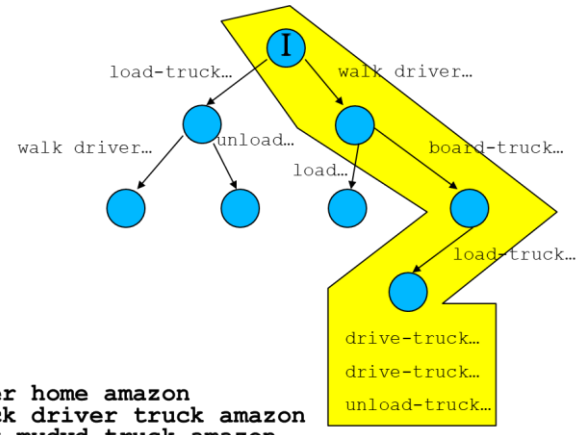
- Closed List = [ ] (empty list since we haven't dealt with any state just yet)
- Open List = [*I*] (contains only the initial state as that is the starting point of our problem)

The algorithm to now solve the forward search problem is shown alongside.

```
while open not empty:
   S = remove the next state from open
   If S is not in closed
    expand S and
    add S to closed;
    add successors to open
```

PLANNING AS FORWARD SEARCH

Applying the above algorithm to our example, we come across the tree given alongside, where we keep expanding until we reach the goal state and get the desired plan.




```
walk driver home amazon
board-truck driver truck amazon
load-truck mydvd truck amazon
drive-truck driver amazon london
drive-truck driver london myhouse
unload-truck mydvd myhouse
```

OPEN LIST OPTIONS

This algorithm is pretty general. We can modify it to make it more specific.

BREADTH FIRST SEARCH

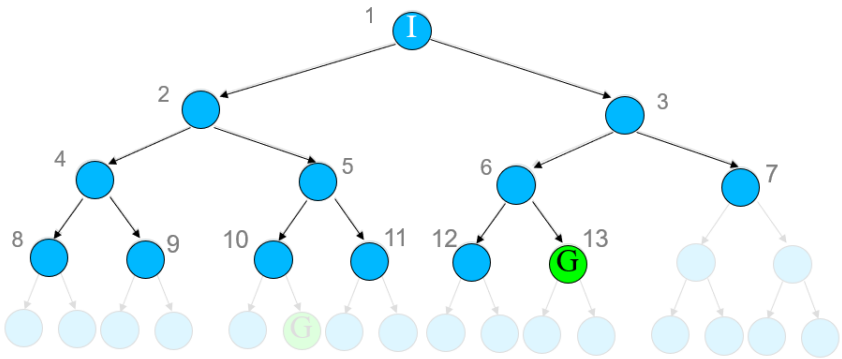
What if the Open List was a **Queue**?

- Can **Push** states onto the back
- Can **Pop** states from the front

This gives us **BREADTH FIRST SEARCH**.

- Generate all the children of the current node
- Expand nodes in the order that they were generated in.

By performing breadth first search, out planning would look like the diagram shown alongside. The numbers corresponding to each node, is the order in which they were considered. The planner keeps looking for different states, until it reaches the goal state (number 13).



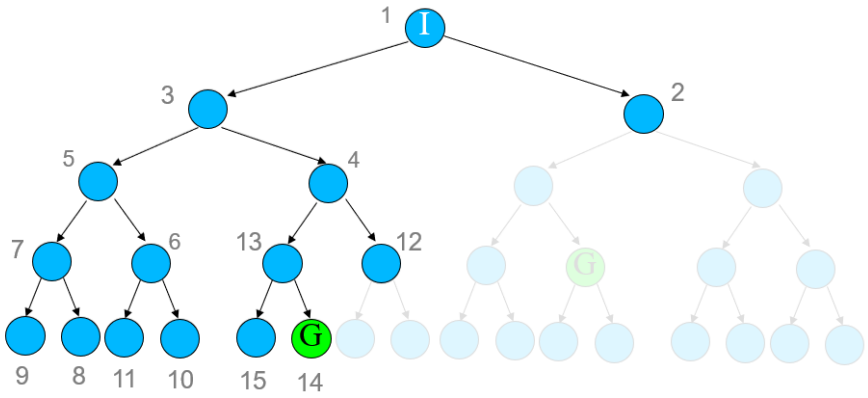
DEAPTH FIRST SEARCH

What if the Open List was a **Stack**?

- Can push/pop states from the front

This give us **DEPTH FIRST SEARCH**.

- Generate all children of the current node
- If it has children, select and expand its first child
- If not, backtrack to the most recent node with unexpanded children



In the diagram given alongside, the numbers corresponding to each node is the order in which they were added to the stack.

NOTE

DFS is usually not used in planning. As can be seen in the examples shown above, the number of actions required to reach the goal state using BFS was 3, whereas with DFS its 4. Thus if each path has the same cost, then BFS gives us the optimal solution.

HOW LONG WILL THIS TAKE?

Consider the following scenario

- There are 20 actions to choose from at each stage
- Solution plan is 20 steps long

The worst-case scenario for this plan would be to search through all 20^{20} possibilities

- This would mean searching through more than 10^{26} possibilities
- The estimated age of the universe in seconds is 4.35×10^{17}.
- Even if we consider each state in a microsecond, it will still take 10^{23} seconds, which is still greater than the estimated age of the universe.

Planning is **PSPACE Hard**. Thus, we need to find a way to search through only the most promising looking plans.

HEURISTIC SEARCH

A **heuristic** is a function that takes a state and returns an estimate of its distance from the goal. A heuristic is a guideline which gives us an estimate of how far a given stat is from the goal.

- Smaller = Better, as nearer to the goal state

Heuristics are (usually) not perfect, otherwise there would be no need for a search.

For now let's assume we have some function $h(S)$ that gives us a 'distance to go' from the state S to the goal

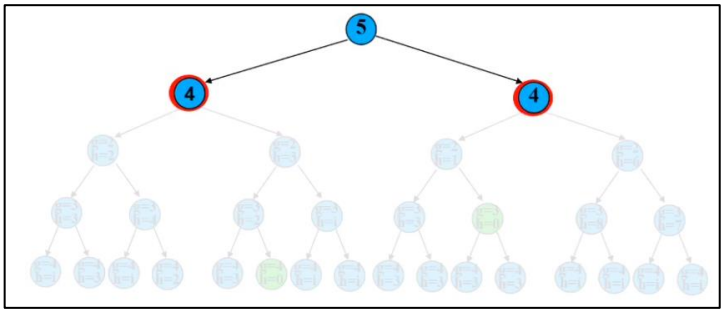
- For goal states, $h(S) = 0$
- For other states, $h(S) > 0$

BEST FIRST SEARCH

Unlike Breadth first search where we used a queue and depth first search where we used a stack, in **Best First Search**, we use a **Priority Queue**, i.e. each state is associated with a priority value when entering the queue. It is important to note that when we see 2 states with the same heuristic value, we would want to expand on the state which we observed

first (*stable insertion*). So the aim is to **always expand the first node we found that has the lowest heuristic value**. So if there are 2 children with the same value, we will consider the one on the left first, and then the one on the right.

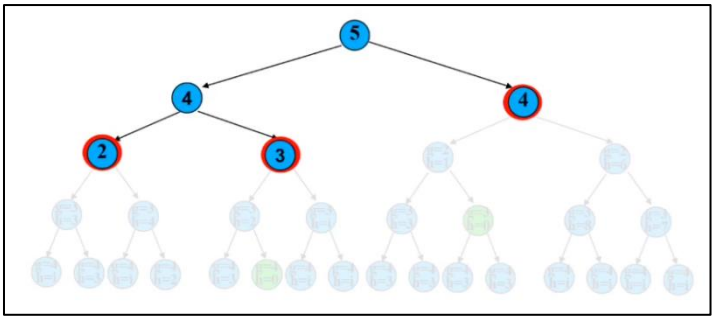
Consider the example given alongside. The values given in each node is the heuristic value of that node.



In the initial state, we only have one state in the open list which has a heuristic value of 5.
When we expand on that node, we get 2 more states, each with a heuristic of 4.

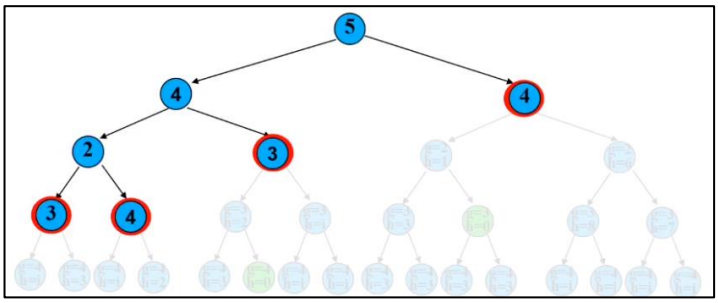
So now the initial state has been removed from the open list, and its 2 child node have been added to the Open List (highlighted in red).

Since we are considering the one on the left before the one on the right, we will expand on the one on the left.



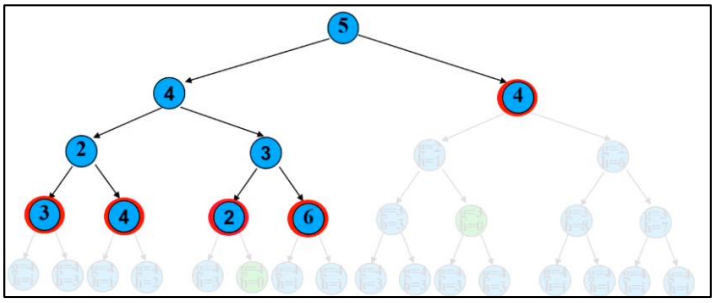
So the states with the heuristic value of 2 and 3 are added to the Open List in addition to heuristic value 4 (highlighted in red)

Since the list needs to be sorted by its heuristic value, the first node on the list become the one with heuristic value 2, and that's the one which we expand.



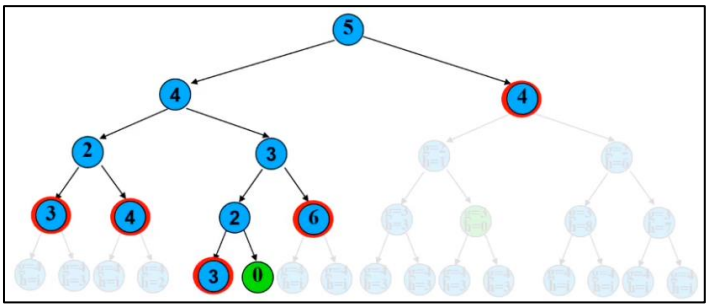
Now the Open List has 4 nodes ($[3,3,4,4]$). Since we want to expand on the node with the lowest heuristic value which was observed first, we will expand the node with heuristic value 3, on the 3rd tier of the tree (instead of the one on the 4th tier).

The Open List will now have 5 values in it (the ones highlighted in red) ($[2,3,4,4,6]$). Since the node with the heuristic value 2 is the first one on the list, that is the node we will expand.



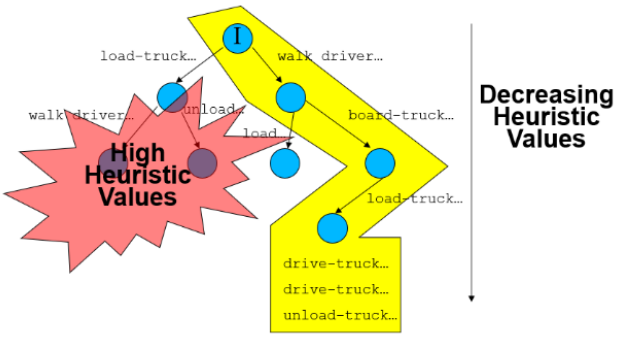
We get 2 more states with heuristic values 3 and 0. The state with the heuristic value 0 is in fact a goal.

Thus we were able to use heuristic to guide us to the goal state without having to expand all the nodes in the search space.



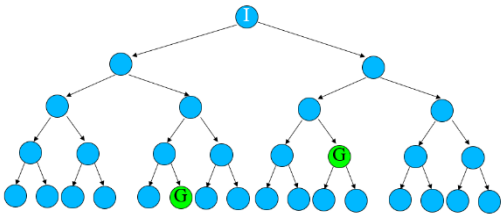
HEURISTIC VALUES IN PLANNING

Ideally, we want all the values that take us to the goal to have decreasing heuristic values, and all the nodes which do not take us to the goal to have high heuristic values. If the heuristics had this property, they would allow us to get to the goal state more quickly, and not expand those states which do not take us to the goal.



But What About Length of the Plan?

With Best First Search, we reached the goal on the left (in the diagram), which has 4 plan steps, as opposed to the goal on the right which has only 3 plan steps.



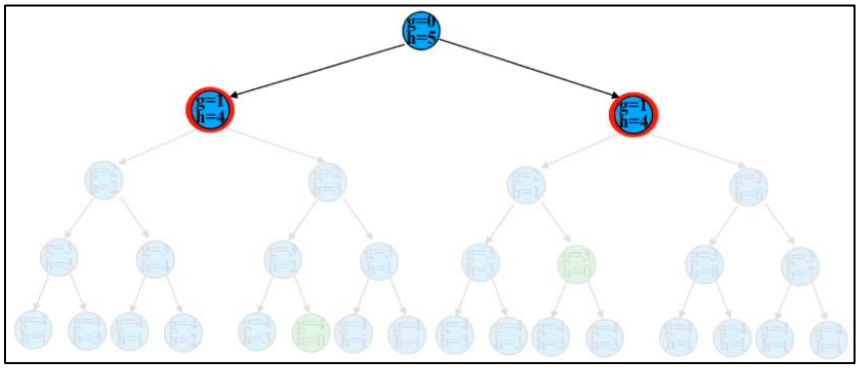
A* SEARCH

Thus we need to redefine the "best" in our best first search. We can do so by defining $f(S)$, where $f(S)$ is our priority and is calculated using the following formula

$$\boldsymbol{f(S) = g(S) + h(S)}$$

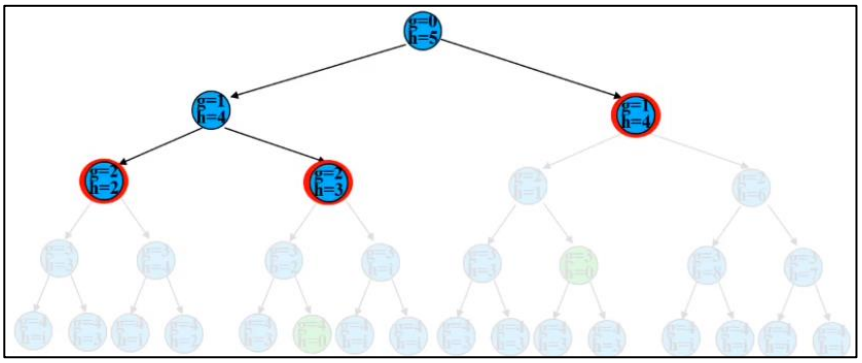
Where $g(S)$ is the distance from the root of the search tree to the current known, which is a known value since we know how many states we have expanded. Thus $f(S)$ is the sum of how far we have come to reach the goal ($g(S)$) and how far we think we have to go to get to the goal ($h(S)$).

Considering the previous example again, we have the tree given alongside. But in addition the heuristic value in each node (h), we also have the distance from the root of the tree (g). For the root node, $h = 5$ and $g = 0$, thus $f = 5 + 0 = 5$. Since that is the only node on our Open List, we will expand it and get its 2 child nodes.

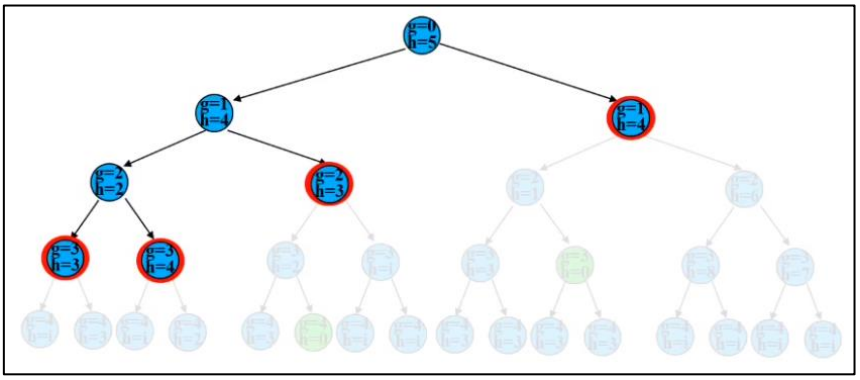


Both child nodes have a priority of 5 ($f = 4 + 1 = 5$), thus we consider the one which we observed first, i.e. the one on the left, and leave the other one on the Open List.

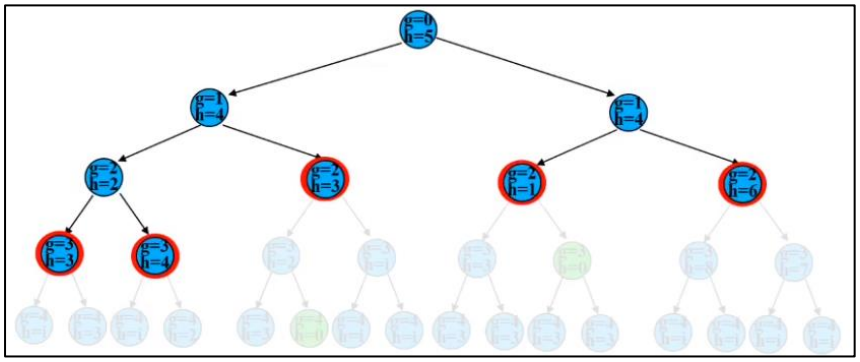
Now we get 2 more child nodes, one with $f = 4$ and one with $f = 5$. The lowest priority is 4, thus now we expand on the left most node.



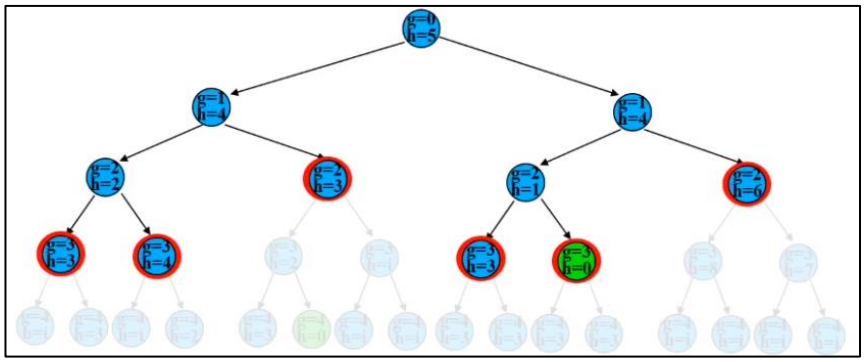
On expanding that node, we get 2 more nodes. Now there are 4 nodes in the Open List with priorities of 5,5,6 and 7. Thus we want to consider a node with the priority of 5, but we also want to break the tie by considering the one we observed first. Thus now we expand on the right child of the root node.



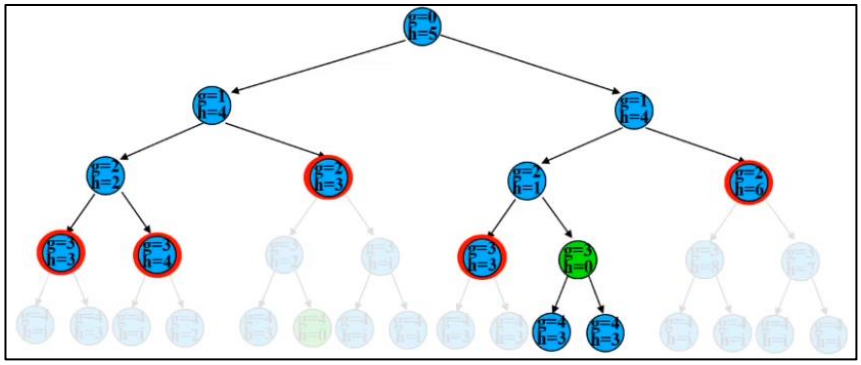
On expanding that node, we get 5 states in the Open List, with priorities 3,5,6,7 and 8. The node with the lowest priority is the one with priority 3, thus we expand on that node.



On expanding that node, we get 2 more nodes, one of which has a heuristic of 0, i.e. is the goal state. Now we have 6 nodes in the Open List, with the priorities of 3,5,6,6,7 and 8. Thus we expand the node with priority of 3, which happens to be the goal state.



On doing so we get the nodes as shown alongside, and we can terminate our A* search. Note that we did not terminate the planning until we expanded the goal state.



Question: What if $h(S)$ is admissible and consistent?

- **Admissible**: never overestimates the distance to a goal state. So whatever the heuristic tells us about the distance to the goal state, the distance is always less than or equal to that value.
- **Consistent** (monotonic): $h(S) \leq h(S') + d(S, S')$. For any state, the heuristic value of that state must be less than or equal to the distance from that state to another state ($d(S, S')$) plus the heuristic value of the other state ($h(S')$).

Answer: If the heuristic satisfies these properties, then at any point, the smallest $f(S)$ on the Open List underestimates ($\leq$) the shortest possible plan that can solve the problem.

- Proof: $g(S)$ is perfect, so doesn't overestimate distance from I to (S); $h(S)$ doesn't overestimate; so adding them together will never produce an overestimate.

Question: What if we find a goal state, but put it on the open list, and wait until its expanded?
Answer: We have found the shortest possible plan.

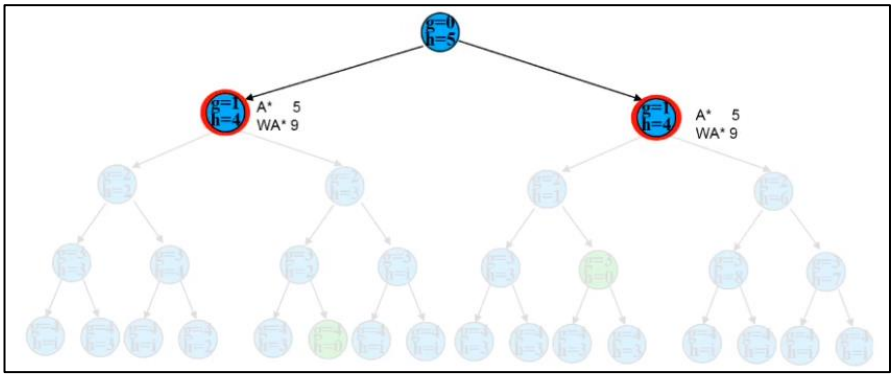
- Proof by Contradiction: Search expands a state with smallest $f(S)$; if there was a shorter path to a goal state S', $f(S')$ would be less that $f(S)$ and it would have been expanded first.
- Consistency guarantees that there is no node that has not yet been put on the Open List that might have a shorter path than the one that is there.
- What if the best node just hasn't been added to the open list yet? Every node has an ancestor that is (or has been) on the open list (e.g. Initial State) so if the cost of getting from any ancestor to the goal is guaranteed to be greater than the cost of getting from the ancestor to S + the cost of getting from S to the goal, no node below one that has ever been on the Open List can be better. This is the definition of consistent heuristics.

WEIGHTED A* SEARCH (WA*)

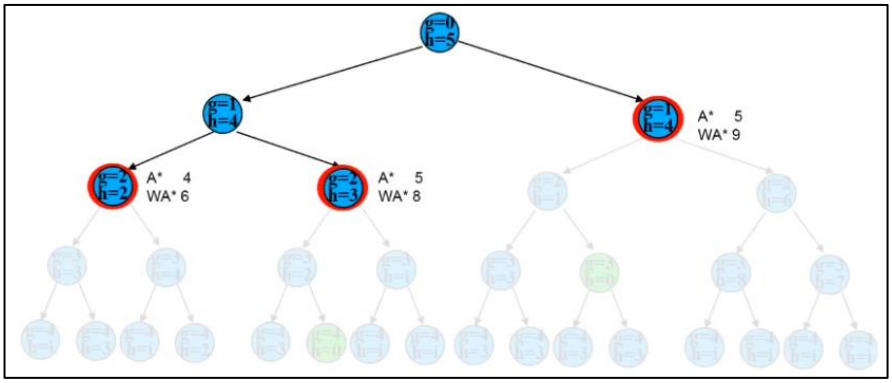
Question: What if we want a plant that is 'good' quality but not necessarily optimal?
Answer: We can use **Weighted A***

- $\boldsymbol{f(S) = g(S) + W * h(S)}$, where $W > 1$ (if $W = 1$, then it becomes regular A* Search)
  - W is a constant that gives an optimality bound
  - Now the solution we find will be within a factor W of optimal, i.e. if $W = 2$, then the solution we find is no more than twice the length of the optimal solution.
  - Find a solution faster by biasing search towards nodes closer to the goal
  - Advantages of this method are:
    - Don't have to explore as much of the search space to find a solution
    - Still have some guarantee of the solution being a reasonable one

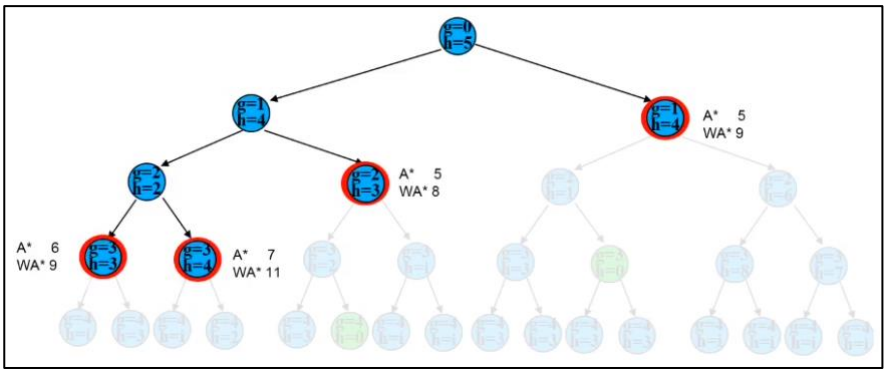
Consider the previous example. We can try finding the goal state using WA* Search where $W = 2$.
As before we start with the initial node and expand to its 2 child nodes. Now in WA* each has a f value of 9 (instead of 5 in A*). To break the tie, we expand on the one on the left.



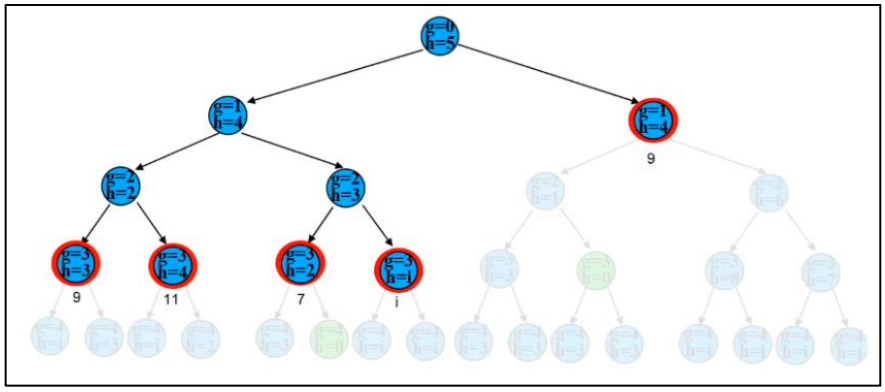
On expanding that node, we get 2 other nodes in the Open List, with f values of 6 and 8. So we expand on the node with the value 6.



On expanding that node, we have a total of 4 nodes in the open list. The lowest f value is 8, so we can expand that node.

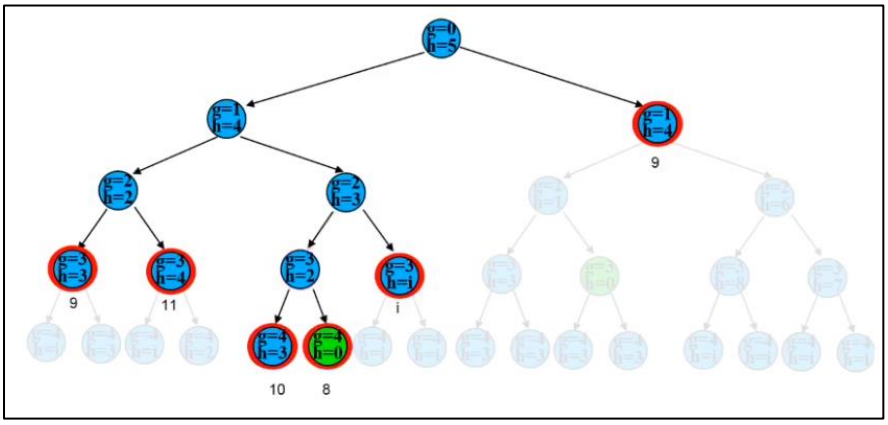


Now we've got 5 nodes on the list with f values of 7, 9, 9, 11, and ∞. So we can expand on the node with the f value of 7.



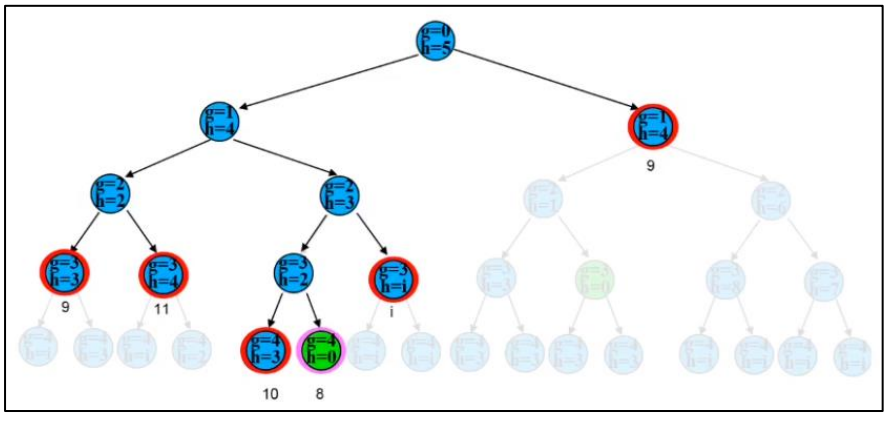
By expanding on that state, we have reached the goal state.

In A* we have to continue planning until we have expanded the goal, to maintain optimality. In WA* we have to continue searching until we have expanded the goal state in order to guarantee that our solution is within a factor W of optimality.



So, now that we have reached a goal state, we have to pick another node to expand upon, which in this case happens to be the goal node itself as its f value is 8 which is the least amongst the others in the Open List. This is not always the case, and usually other nodes might have a lower f value than the goal state.

So we remove the goal node from our Open List (cant expand it further since it's a leaf node) and we can terminate our search.



Notice here in contrast to A* we have found the non-optimal solution. The optimal solution has a length of 3, but the non-optimal solution has a length of 4, which is less than twice the length of the optimal solution ($4 < W * 3 = 2 * 3 = 6$)

WHAT IS A GOOD PLAN?

Suppose we want to go from the Strand Campus to Barbican. Let's say we have 2 options

- Fly a helicopter to Barbican
- Walk to Strand Temple station and take the tube to Barbican

(fly-helicopter strand barbican)

or…

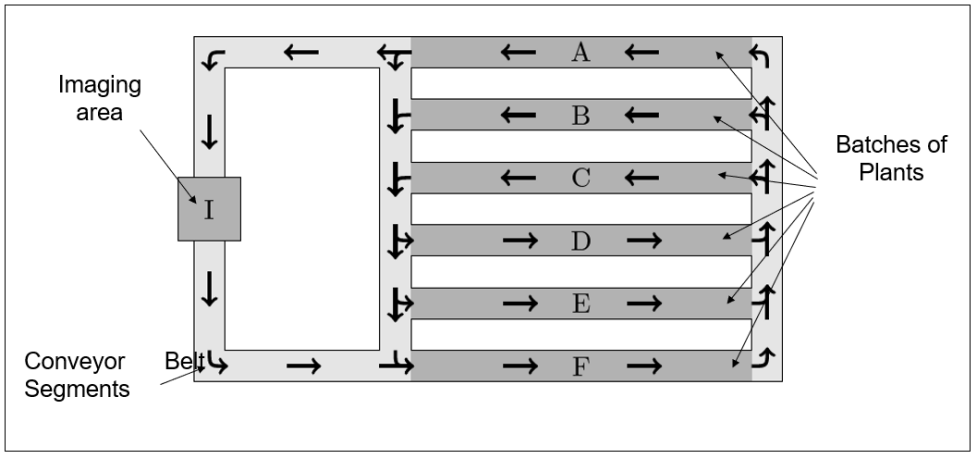
(walk strand temple)
(tube temple barbican)

It might seem nice to fly to Barbican, but the cost of flying to Barbican is much higher than the cost of the other plan. Thus, basic STRIPS planning cannot cover this scenario. But an extension of STRIPS planning is STRIPS planning with **Action Costs**. The idea is that we can assign a cost to each action, that tells us how expensive it is for us to do those actions. A* and WA* can be used for this. It will affect the g values for the states and the costs needs to be considered when calculating the h values for each state.

PDDL MODELLING APPLICATION EXAMPLE – SCANALYZER

It is an application of planning to greenhouse logistics. Modern genetic research in greenhouses relies in plants being exposed to all of the same conditions in order to make sure that each experiment is well controlled. When you have a plant in a fixed place in a greenhouse, there are a number of things that will affect how well it grows. For example, how sunny that area is, or how exposed to the sprinkler system that area is.

The Scanalyzer solution is to have a series of conveyer belts in the greenhouse, so that each plant is moving around in the greenhouse, which means that the conditions to which is plant is exposed are much more uniform. This allows scientists to perform reliable experiments on different genetically modified crops, and to ascertain that it the genetic modification which is improving growth and not the condition.



The plants cycle through an imaging area where the scientist can take measurements of the plant to track its growth and other properties.

SCANALYZER TYPES, PREDICATES

Types:

- Type of the Conveyor Belts: **Segments**
- Type of the plants themselves **batches**
  - Operational constraint – plants are moved one batch at a time, not individually.

Predicates:

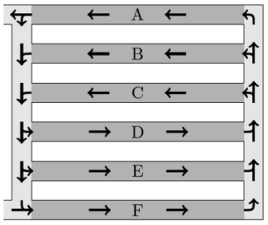
- A batch of plant is **on** a given segment
- A batch of plants has been **analysed**
- Segments are linked together in **cycles**, i.e. the movements the batches can do while going around on the conveyer belts.

SCANALYZER: INITIAL STATE

Cycles are defined in pairs of belts moving in opposite directions. There is a special cycle called CYCLE-2-WITH-ANALYSIS, that represents going through the analysis chamber to do the imaging and other analysis.

```
(:init (= (total-cost) 0)
(CYCLE-2 A D) (CYCLE-2 A E) (CYCLE-2 A F)
(CYCLE-2 B D) (CYCLE-2 B E) (CYCLE-2 B F)
(CYCLE-2 C D) (CYCLE-2 C E) (CYCLE-2 C F)
(CYCLE-2-WITH-ANALYSIS A F)

(on b1 A) (on b2 B) (on b3 C)
(on b4 D) (on b5 E) (on b6 F)
)
```

Static Facts



Static facts are those facts which do not change during the duration of the planning problem.

At the top of the state we have state that the total cost of the plan is 0, as we are using STRIPS with action costs.

SCANALYZER ACTION 1

This action is a **Parameterized Action**. This means that we are going to define it once for all pairs of batches and pairs of segments using **Typed Parameters**. We are not going to define the actions individually for every pair of batches and every pair of segments.

```
(:action rotate-2
:parameters (?s1 ?s2 - segment
             ?b1 ?b2 - batch)
:precondition (and (CYCLE-2 ?s1 ?s2)
             (on ?b1 ?s1) (on ?b2 ?s2))
:effect (and
   (not (on ?b1 ?s1)) (on ?b1 ?s2)
   (not (on ?b2 ?s2)) (on ?b2 ?s1)
   (increase (total-cost) 1)))
```

Typed Parameters

Restricts pairs of segments that can be swapped

Swap two batches

Action has cost 1

The precondition restricts the pairs of segments which can be swapped to the ones which have a defined cycle between them. The other 2 preconditions make sure that when swapping batches from the conveyers we ensure that the correct ones got swapped.

The effect of this action simply swaps the batches between the segments. It also increases the cost of by 1.

SCANALYZER ACTION 2

This action takes the batch one the first segment and takes it through the analysis chamber.

Analysis can only be performed on plants which are on Conveyor A and are going to Conveyor F.

The effect of this action adds the fact that the batch which was on conveyor A initially has now been analysed and increases the total cost by 3.

```
(:action rotate-and-analyze-2
:parameters (?s1 ?s2 - segment
             ?b1 ?b2 - batch)
:precondition (and
          (CYCLE-2-WITH-ANALYSIS ?s1 ?s2)
          (on ?b1 ?s1) (on ?b2 ?s2))
:effect (and (not (on ?b1 ?s1)) (on ?b1 ?s2)
             (not (on ?b2 ?s2)) (on ?b2 ?s1)
             (analyzed ?b1)
(increase (total-cost) 3)))
```

In example: only the A—F conveyor goes through the imaging area

Mark batch as being analyzed

SCANALYZER GOAL

At the end of the plan, we want to make sure that all the batches have been analysed and that every batch has been returned to its original positions.

```
(:goal (and
(analyzed b1) (analyzed b2) (analyzed b3)
(analyzed b4) (analyzed b5) (analyzed b6)
(on b1 A) (on b2 B) (on b3 C)
(on b4 D) (on b5 E) (on b6 F)
))
```

Analyse Batches

Return to Initial Positions

In addition to this, since we have defined actions costs, we can define a **metric**. In this case our metric is to minimize the total cost.

```
(:metric minimize (total-cost))
```

Prefer plans with lower costs:
`rotate-2` costs 1, `rotate-and-analyze` costs 3.

SCANALYZER DEMO

A demo for the Scanalyzer plan has been given below.

```
0.000: (analyze-2 seg-in-1 seg-out-1 car-in-1 car-out-1)  [0.001]
0.001: (analyze-2 seg-in-1 seg-out-1 car-out-1 car-in-1)  [0.001]
0.002: (analyze-2 seg-in-3 seg-out-1 car-in-3 car-out-1)  [0.001]
0.002: (rotate-2 seg-in-1 seg-out-2 car-in-1 car-out-2)  [0.001]
0.003: (rotate-2 seg-in-3 seg-out-1 car-out-1 car-in-3)  [0.001]
0.004: (analyze-2 seg-in-2 seg-out-1 car-in-2 car-out-1)  [0.001]
0.005: (rotate-2 seg-in-2 seg-out-1 car-out-1 car-in-2)  [0.001]
0.006: (analyze-2 seg-in-1 seg-out-1 car-out-2 car-out-1)  [0.001]
0.007: (rotate-2 seg-in-1 seg-out-1 car-out-1 car-out-2)  [0.001]
0.008: (rotate-2 seg-in-1 seg-out-2 car-out-2 car-in-1)  [0.001]
0.009: (rotate-2 seg-in-1 seg-out-3 car-in-1 car-out-3)  [0.001]
0.010: (analyze-2 seg-in-1 seg-out-1 car-out-3 car-out-1)  [0.001]
0.011: (rotate-2 seg-in-1 seg-out-1 car-out-1 car-out-3)  [0.001]
0.012: (rotate-2 seg-in-1 seg-out-3 car-out-3 car-in-1)  [0.001]
```

PLANNER PERFORMANCE

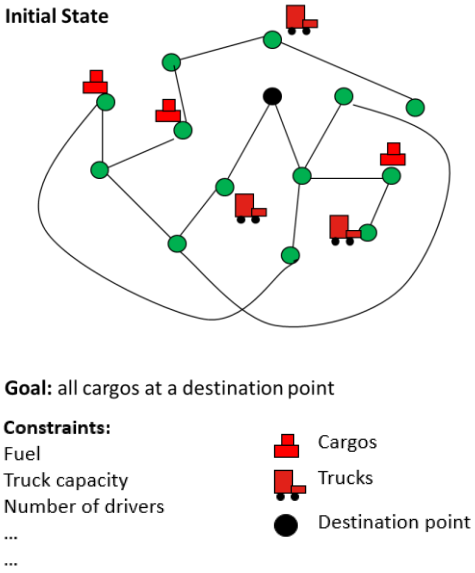
DSS: A* search + heuristic **just for Scanalyzer**
IPC: **domain independent** planning competition planners

Layout 1			Layout 2			Layout 3			Layout 4		
Size	DSS	IPC	Size	DSS	IPC	Size	DSS	IPC	Size	DSS	IPC
6	18*	18*	6	22*	22*	6	26*	26*	6	22*	22*
8	24*	24*	8	30*	30*	8	36*	β36*	8	32	30*
10	30*	30*	10	38*	44	10	46*	46*	10	40	44
12	36*	36*	12	46*	54	12	56*	60	12	—	56
14	42*	42*	14	54*	64	14	66*	72	14	—	66
16	48*	48*	16	62*	74	16	—	86	16	—	84
18	54*	54*	18	—	84	18	—	94	18	—	94
20	60*	60*	20	—	94	20	—	108	20	—	106
22	66*	66*	22	—	104	22	—	114	22	—	116
24	72*	72*	24	—	114	24	—	134	24	—	128

As can be seen in the table given alongside, the DSS was not able to complete all the planning tasks for layouts 2-4, even though the IPC domain independent planner was able to do so. This might have been because when creating a solver, people often reach a point where they have a working model, and don't bother optimising it further. Whereas when there is a domain independent planner, there are usually some clever tricks programmed into it, which allow you to complete more complex plans easily. So even though you cannot use domain dependent stuff in domain independent planners, they are still extremely useful. It is usually simpler to write a PDDL problem for a domain independent solver rather than implementing a complex AI system solver.

IMPROVING HEURISTIC SEARCH

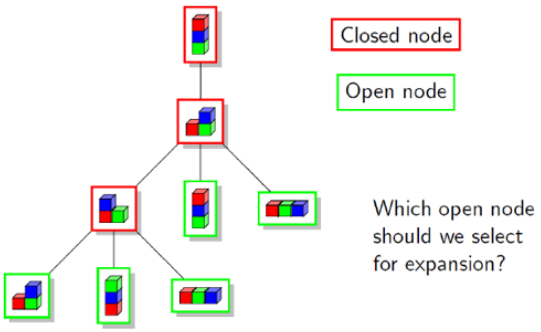
FUNDAMENTALS (RECAP)



If we return to the opening of this module, we discussed the idea of how to formulate a planning problem. Such as the logistics domain of driving packages between locations using the available trucks. We need to make sure that we successfully encapsulate all the knowledge required to express any possible permutation of the problem – this being the domain – alongside the individual problems themselves. We then encapsulate this within the PDDL language as means to express the planning problem in a way that numerous planning systems can look towards searching for a solution.

After this, we can then begin to search for a solution to the problem. We may perhaps use a simple solution forward search algorithm such as Breadth or Depth First search which allows us to explore the search space in a simple, methodical fashion. But this of course quite suboptimal, we want something more effective.

We want to find a more effective means of calculating which of the open nodes should be explored next. Hence, we have the '**open list**' data structure which maintains the collection of 'open nodes' to visit. In Breadth First Search, this is just a queue. In depth first search, it's a stack. But what we really want, is to prioritise them within the open list based on some criteria.



Hence, we begin to consider the **heuristics** of the problem: a simple rule of thumb metric where we can utilise that knowledge in a manner that will allow us to search the state space faster.

This is in effect, a **goal distance estimation**: how far away do we think roughly that we are from the goal state of the problem.

HEURISTIC SEARCH PLANNING (RECAP)

So, to use heuristic search planning, we need to have established our heuristic of the problem space, which tells us information about how close we are to the goal, denoted as h.

As well as how far we have travelled in the problem space to date, which is expressed as g. This is the cost of everything we have done to reach this point.

Depending on the algorithm, we can exploit g and or h to help us prioritise how to explore the problem space.

Hence for example, uniform cost search expands open nodes with the lowest g value, which best-first search uses the h value.

- **Requires**
  - **h**: estimate of $s_{current}$ to goal
  - **g**: cost of path from s_{init} to $s_{current}$
- **Method**
  - Prioritise open nodes with heuristic
  - Different algorithms exploit heuristic and costs.
- **Examples**
  - **Uniform Cost Search:** Expand node with minimum **g**
  - **Greedy Best-First Search:** Expand node with minimum **h.**
  - **A*:** Expand node with minimum **(g+h)**

It's then we combine them in A* to achieve a much more effective technique. A search process that is both prioritising states that are closer to the goal and detracting from those reached using expensive paths.

BUILDING HEURISTICS (RECAP)

But building a heuristic is a difficult challenge, especially on a problem to problem basis and in situations in planning where we need to create something that is domain independent.

There are three key features of a heuristic that we hope to achieve in order to achieve the best search process…

- **Admissible**, whereby the heuristic never overestimates the cost of reaching the goal from a given destination. It either underestimates or – in the worst case – calculates exactly how far away we are from the goal. This might seem weird to have it underestimate, and ideally we don't want it to underestimate too much, but if it overestimates it's more likely for it to detract us from exploring a given state as a valid path from the goal.

- **Admissible**
  - If **h(s)** is a lower bound of the cost of all goal-reaching plans starting at **s**.
  - i.e. Heuristic value is less than or equal to the actual cost.
- **Consistent**
  - **h(s)** is always less than or equal to estimated value from any neighbouring vertex, plus the cost of reaching it.
  - i.e. if we travel from **s** → **s'** using action **a**
    - $h(s') - h(s) + cost(a) = 0$
- **Additive**
  - $h(s) = h_1(s) + h_2(s)$ where $\forall s \epsilon S$ admissible.

- **Consistent**, meaning that for any child states, the heuristic value will be equivalent to the parent's heuristic value minus the cost it took us to reach it.
- **Additive**, where two heuristics h_1 and h_2 are additive, if they and a third heuristic, which is the summation of the two are all admissible.

OUTSTANDING PROBLEMS (RECAP)

However, while we have established these base principles, there are a lot of issues we still need to address, which are the following,

- **Building Domain-Independent Heuristics**
- **Improving Search**

- How do we build a heuristic for a given domain or problem?
  - Especially if using a planning system that relies on PDDL, where we're not going to be giving it this information. We just give the domain and problem notation.
- How can we search more effectively in the state space?
  - All the heuristic search systems we've seen to-date work fine, but we're going to be dealing with large and complex problem spaces. So, we need to find a way to further optimise the search process.
  - How do we build more effective heuristics that help us better search the state space?
  - Can we build heuristics that help not just prioritise states on the path to the goal, but also help us search the space more efficiently?

IMPROVING HEURISTIC SEARCH

Now there are multiple approaches to doing all of this and like many facets of planning, these are ongoing research endeavours. The following are the 3 approaches that are useful for building on basic heuristic search.

- **Problem Relaxation and Relaxed Planning (RPG Heuristic)**
- **Planning Landmarks**
- **Landmark Counting (and LAMA)**

- **Problem Relaxation and Relaxed Planning (RPG Heuristics)**: whereby we create a simpler – relaxed – version of the problem that removes some of the constraints. We then solve the relaxed version of the problem from this state and this gives us an estimate of goal distance. This creates a domain-independent heuristic for any given problem.

- **Planning Landmarks**: these are facts that must be true at some point in every valid solution to a problem. For example, if we were to consider the driver logistics domain again, then all trucks need to visit the packages and in turn those packages need to be on a truck at some point in order for them to reach their destination. This is actually a useful means not just to estimate distance to goal but can help decompose planning problems into smaller tasks that need to be solved.
- **Landmark Counting (and LAMA)**: once we have landmarks for a given problem, we can estimate the goal distance by counting how many landmarks we have yet to solve for a that problem.

RELAXED PLANNING AND RPG HEURISTICS

DOMAIN INDEPENDENT HEURISTICS

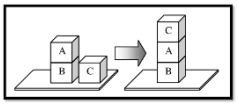
Heuristics for problems in planning are important, but we might not know what a good heuristic to a problem at the time is we're establishing what the problem is. Typically, when encoding planning domains in the likes of STRIPS and PDDL, we don't provide the heuristic as part of that problem. So, we need to establish a means to create the heuristic in a way that is independent of the actual domain itself, but instead arises from an analysis of the problem itself. We can do this by using relaxed planning.

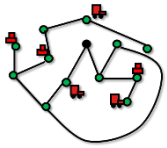
- Building heuristics that can adapt to different domains/problems
- Not reliant on specific information about the problem
- We analyse aspects of the search and planning process to find potential heuristics.

If we consider early planning applications, even the likes of STRIPS was reliant on domain independent information, given much like PDDL it is reliant on a notation that users adopt to input the planning problems they are trying to solve.
In fact, STRIPS relies on what is known as the Goal-Counting heuristic. It looks at the current state and counts how many of the goals listed in the goal state are currently satisfied.

- **STRIPS planning (Fikes & Nilsson, 1971) uses number of goals satisfied.**
  - Number of variables from s_{goal} satisfied in $s_{current}$
- **Goal-Counting Heuristic: More satisfied goals, then closer to the goal?**
- **Problems**
  - Uninformative about the problem.
  - Ignores a lot of the problem structure.

So, if we look at say the examples given alongside, then we would count how many of the packages are currently in the correct location in driver log, how many cubes are positioned correctly atop one another in blocks world. But the problem is, that's actually not a very good heuristic. It doesn't tell us an awful lot about how far away we are from the current state to the goal, which is what we need from a good heuristic. If we consider that driver log problem, the number of states where the goal-counting heuristic would be zero is significant, given there is a lot of legwork that needs to be completed in order to get even one package in the correct destination, never mind all three. Hence goal-counting is ignoring a huge amount of the problem structure and not really telling us anything interesting or useful.






RELAXED PLANNING

Ideally, we want a much more complex heuristic that gives us richer information about the problem. This is hard as planning is expensive to compute. Most planning problems exist within PSPACE-complete of computational complexity while trying to solve a problem is typically NP-Hard. So we need to find a heuristic that is informative, but also can be calculated in polynomial time so as to minimise overhead.

One great way to achieve this is using **relaxed planning**: whereby we attempt to solve a simpler or relaxed version of the problem, and then use the solution to help construct a heuristic value.

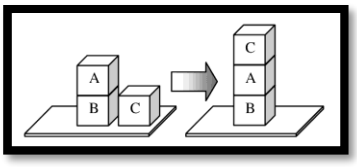
Relaxed planning Heuristics reliant on solving the relaxed problem.

- Estimate the cost using simpler version of the existing problem
- Solving the relaxed problem provides a lower bound on optimal plan length.

DELETE RELAXATION

Relaxed planning is based on the principle of delete relaxation. Actions in planning problems can be of the following 2 types:

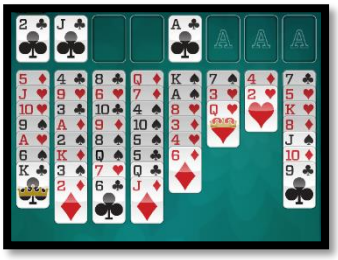
- **Add Effects**: where we add new information to the state
- **Delete Effects**: where we remove information from the state



If we consider the action that is happening in BlocksWorld example (shown alongside), as C move onto A, we add the effect that C is on A. But the delete effect removes the fact that C is on the Table and A is clear.

Deletion relaxation is a principle where during the state transition system that we could use for a typical problem space, we ignore the delete effects when the state transition function γ (gamma) is applied. That is, that the state space becomes broader and more actions are applicable in each state, because information that shouldn't be true is now also true during planning.

- Delete Relaxation heuristics ignore delete effects when the state-transition model is applied.
- Estimate cost of the goal by removing negative effects of actions, i.e. any PDDL effect that removes a fact is no longer considered.



So if we consider for example a game of FreeCell solitaire, where cards can either moving into free cells on the board or into tableau positions – meaning the decks of decreasing numbers and alternating red/black colours – if we remove the delete effects, then the fact that a cell is free still holds, even after we put a card on top of it. Similarly, because cards are still free when we put cards on top of them, it means other cards can still be put atop them as well.

This sounds anarchic, but what it does is it removes so many of the hard constraints of the puzzle and makes it a lot easier to solve.

RELAXED PLANNING

The process of finding a solution to a problem that uses **Delete Relaxation** is known as **Relaxed Planning**, which in turn generates **Relaxed Plans**.

How does this work as a heuristic?

- For current state s_{curr}, calculate a relaxed plan
- $h(s_{curr}) = cost\ (no\ of\ actions)\ of\ the\ relaxed\ plan$
- Provided we generate the optimal relaxed plan, then this will provide us with an admissible heuristic.

RELAXED PLANNING GRAPH (RPG) HEURISTICS

This is the approach utilised in the *FF planner* from 2001 that is reliant on the principles of *GraphPlan*, a system whereby we generate a flat layered approach to search. The RPG heursitc generates relaxed plans that are sets of actions that can

be executed in parallel at a given time step. We then count how many sets of these actions exist and this gives us our heuristic value.

- Relaxed Planning Graph (RPG) Heuristic creates a simple layered approach to exploring the state space.
- Inspired by the GraphPlan system
- The relaxed solution plan $P' = \langle O_0, O_1, \dots, O_{m-1} \rangle$
  - Where each O_i is the set of actions selected in parallel at time step i_i and m is the number of the first fact layer containing all the goals.
  - $h(S) \coloneqq \sum |O_i|$

BUILDING A RELAXED PLANNING GRAPH

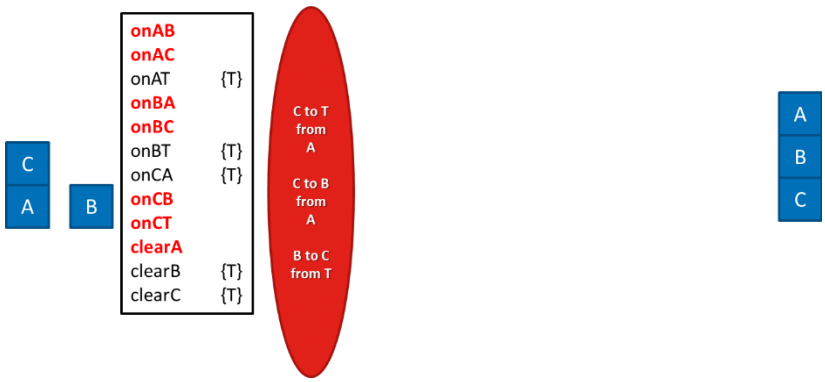
So, let's start with this situation, where we're going to calculate the RPG value for this BlocksWorld example. I have C on A, A and B on the table and the goal state is to have A on B on C.

For the relaxed planning graph to be constructed, it builds what is known as a fact layer: this is going to include all the information that is true at this point in time with respect to the problem.

So our first **fact layer**, F_0, only has the facts that are true for the current state (S_0). However, for the sake of this example, the facts that are not true are also mentioned (highlighted in red). The facts that are true are in black followed by a "{T}" in front of them. Normally, the facts which are not true are not mentioned in the initial fact layer, due to *closed world assumption*, but have been mentioned here for the example.

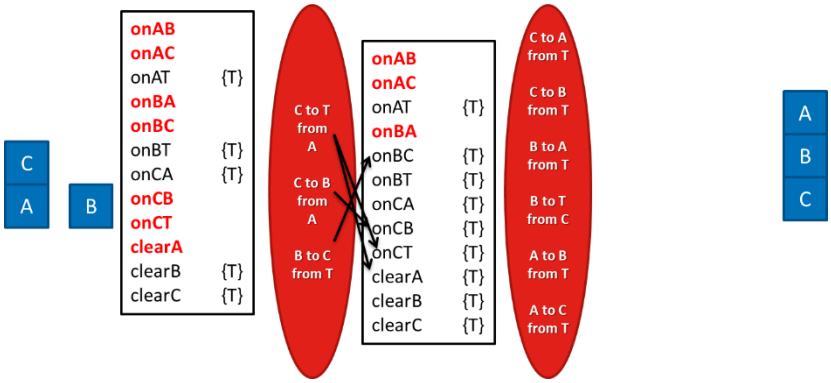
Next, we also have the first **action layer**, A_0. This is a list of all the valid actions that can exist at this point in time. We can use these to generate the first fact layer (F_1), by performing the following actions.

- Moving C on to the table T
- Moving C onto B
- Moving B onto C



Once we have generated the new fact layer (F_1), we can generate the new action layer (A_1) and generate a new fact layer, keeping in mind that we only apply the add effects and not the delete effects.

This new fact layer has made several facts now true such as B now being atop C, C being on top of A and C being atop B. Given that the delete

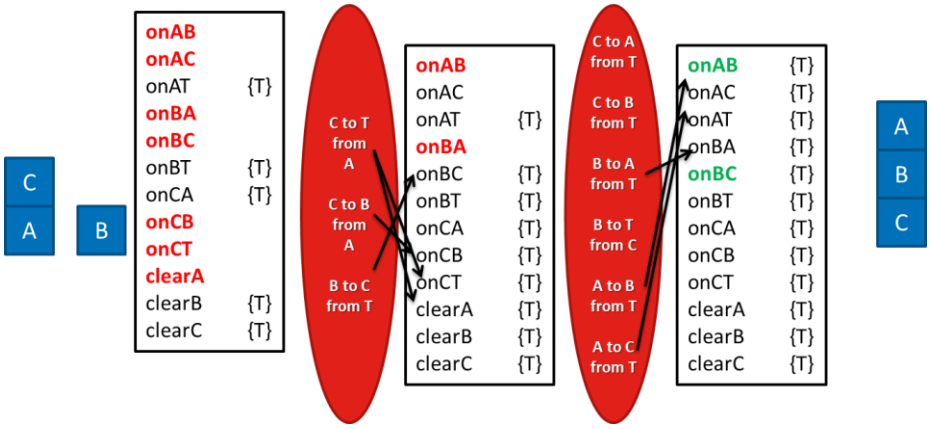


effects are not applied, all these facts are true in the fact layer. Even though it's impossible for all three of them to be true at the same time.

On generating the new fact layer (F_2) we get out final fact layer, as all three facts that we expected to see in the goal state, i.e.

- A on B
- B on C
- C on T

Are now all true. Hence, we've reached a fact layer where all the facts pertaining to the goal are true and from this, we can now work backwards to create a solution.



EXTRACTING A SOLUTION FROM A RELAXED PLANNING GRAPH

Given all the facts are true, the process of extracting the relaxed plan solution is to work backwards from the goal layer g, creating new goal layers back to the initial layer, figuring out the number of sets of actions required to achieve it.
We look at the preceding layer and whether a fact at layer n existed in layer $n-1$.
If it did, we add it to $g_{(n-1)}$. If it doesn't, we choose an action from the intermediate fact layer and add its preconditions to the layer $g_{(n-1)}$.

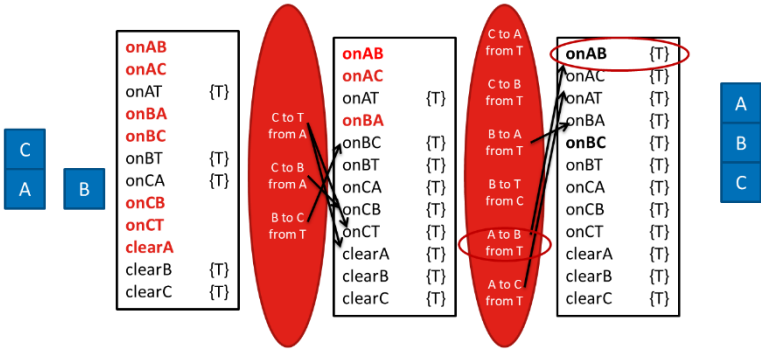
- To get a solution work backwards through the RPG
  - At each fact layer (f_n) we have goals to achieve, g_n
- We start with g_n containing the problem goals
  - For each fact in g_n:
    - If it was in f_{n-1}, add it to g_{n-1}
    - Otherwise, choose an action from a_n and add its preconditions to g_{n-1}
- Stop when at g_0

Considering the previous example, if we look at the goal layer, the 2 facts we need are

- A on B
- B on C

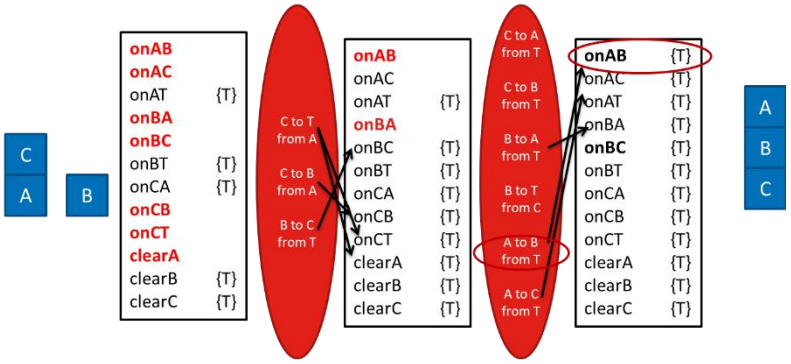
If we then look at the preceding fact layer, B on C is satisfied, but A on B is not. So, we need to find the action that achieves it, in this case, move A from the table onto B.

- **G(n) = onAB, onBC**
- **G(n-1) = onBC**



Next, we have to factor not just the last meaning goal fact (B on C), but also the precondition to move A onto B action, which is that B is clear.

- **G(n-1) = onBC, clearA**
- **G(n-2) = ???**

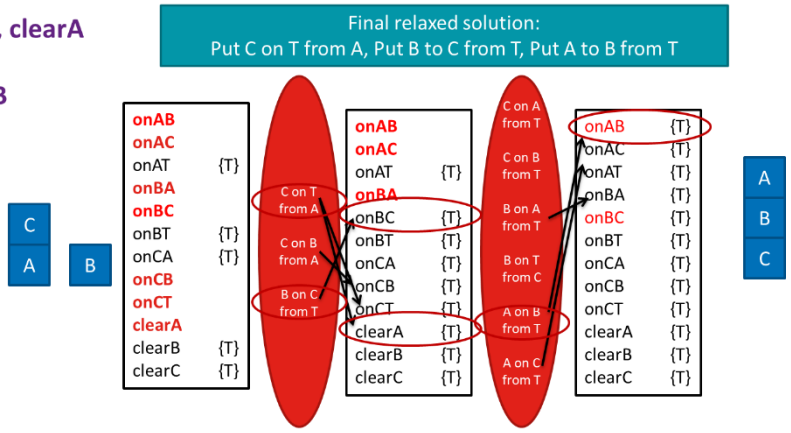


We identify the 2 actions that achieve onBC, which are

- Moving B onto C
- Moving C onto the Table

This satisfies both facts in g_{n-1}, i.e. onBC and clearA. The preconditions for moving C onto the Table (clearC) are already satisfied in fact layer g_{n-1}. The only outstanding fact is whether B is clear, which is satisfied in g_{n-2}.

- **G(n-1) = onBC, clearA**
- **G(n-2) = clearB**



This means we have found a relaxed solution, with two actions set, putting C on T and B on C, and the second set comprised only A on B.

Thus, we have a PRG heuristic of length 3 since there are 3 actions in the relaxed plan.

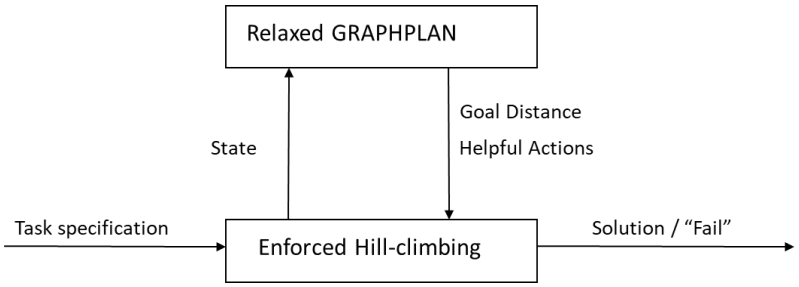
SUMMARY

- A good way to come up with heuristics: solve the simplified version of the problem
- Delete Relaxation: discarding all delete effects – provides simplification that can be explored using greedy search in polynomial time
- Delete relaxation always creates simplification: problems always easier to solve than the original
- RPG layers provide admissible heuristics to relaxed problems

RPG HEURISTIC IN THE FF PLANNER

The **Fast-Forward** or **FF Planning System** from 2001 utilises the RPG heuristic as means to help guide the search.

- FF is a **forward-chaining heuristic search-based planner**
- FF uses the Relaxed Planning Graph (RPG) heuristic to guide search.
  - This involves finding a plan from the current state S which achieves the goals G but ignores the delete effects of each action



- The length of this plan is used as a heuristic value for the state S.

- FF uses both **local and systemic search**
  - **Enforced Hill Climbing (EHC)**: aggressively pursues the next best state it finds, with the qualification of best being influenced by the RPG heuristic. But this aggressive nature causes problems, so it also uses best-first search if RPG cannot help it identify a good state to visit.
    - Upon termination of EHC either outputs a solution plan or reports that it has failed
    - If EHC fails, best first search is invoked.

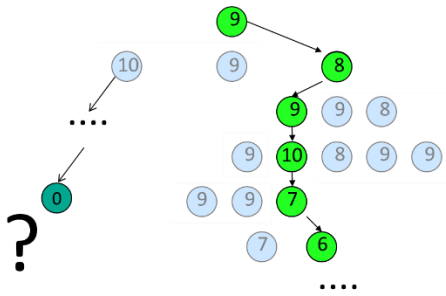
ENFORCED HILL CLIMBING

EHC runs on the idea that it will always seek the best possible improvement in the current state heuristic value. It expands the current state, evaluates all of the successor states and grabs the first state it finds with the better h value and assigns that as the next stage. If no state can be found, it will run breadth first search until it can find one, which is slower and more inefficient.

- Basic idea: lower heuristic = better
- Always try and find the states with the better heuristic value seen so far. Start with $best = h(S_{init})$, aim to get to $best = 0$
- Steps:
  1) Expand a state S
  2) If we have a successor state S' with $h(S') < best$
     - $S = S'$
     - Return to Step 1
  3) If no such state exists
     - Perform breadth first search until one is found
     - Return to Step 1

FF – EHC IN PRACTICE

If we start with this first state with a value of 9, it will then find the next child state with a better value, hence this one here with eight. But then it has a problem, it can't find one with a better value. This is what is known as a **plateau**, given no states in proximity appear to be better that where we currently are in the search space. Thus, it runs breadth-first search until it finds this state with a value of 7 a few layers down. Then resumes as normal.



But there is an issue here, in that it is possible that this path doesn't yield the desired outcome and that actually the goal state exists in the other subtree that we omitted right at the beginning.

PROBLEMS WITH EHC

Given its running purely on the heuristic, it is possible for FF using enforced hill climbing to run down dead ends. Hence therefore the best first search is employed as a backup from the initial state. However, this is expensive, given it is now having to effectively resume search all over again. Similarly, searching using breadth first search on the plateau is also expensive, given that algorithm is in no way

- **EHC can sometimes fail to find a solution**
  - The RPG heuristic can lead it down dead ends
- **When EHC fails, FF resorts to systematic best-first search from the initial state**
  - FF maintains a priority queue of states (the state with the lowest h(S) value is stored first)
  - The front state in the queue is expanded each time
  - The successors are added to the queue, and so on (loop)
- **Searching on a plateau is expensive**
  - Systematic search is carried out on the part where the heuristic is worst

optimised and will explore every opportunity.

EHC: PROS AND CONS

- **EHC is fast**
  - Greedy algorithm takes better states when found
  - Can find solutions quickly
- **EHC is incomplete**
  - Suppose the solution was down one of those paths that we discarded (or never even generated) because another state looked better
- **FF is Complete**
  - If EHC fails, then start again from the initial state using best first search
  - Slower, but complete

EXTRA STATE PRUNING: HELPFUL ACTIONS

There are some additional considerations that FF has when expanding states, given it doesn't always make sense to expand based on the actions involved. Were trying to find the best actions to take as we move through the search space, but it would also help if we can clearly identify actions that are not going to be useful.

FF does apply some additional state pruning, given it can pull this information from the RPG heuristic. For example, actions that can achieve a goal in goal layer 1 are denoted as helpful actions, given they appear to be moving us towards the relaxed plan solution which in turn can move us towards the actual solution.

- **So far, when expanding a state S, have considered all applicable actions when making successors;**
  - However, not all of them are interesting
  - e.g. unloading a package that has just been loaded
- **In FF, extra search guidance is obtained from the RPG heuristic:**
  - Anything that could achieve a goal in g(1) is a helpful action
- **Or, in other words, the first actions in the relaxed plan, and others like them.**

Hence FF can then exploit these helpful actions to only become the successors it generates. Now this does lead to an issue in that while it becomes faster, it becomes even less complete than it was before. It is utilising information from the RPG heuristic as means to speed up the search. However, if this doesn't work, EHC can attempt to run again without the helpful actions constraints to see if that yields any desirable state.

LANDMARKS FOR PLANNING

Landmarks are a measure for understanding elements of a planning problem.

LANDMARKS: CONSTRAINT-BASED HEURISTICS

The alternative to using simple delete relaxation is to try and analyse the problem as it has been provided and from that establish constraints that would encapsulate facets of any valid solution to the problem. These constraints are what we call **landmarks**, information that must be true at some point in the final valid plan. This can include both facts as well as actions that can appear at some point. We can then order these landmarks to dictate the order in which they should be achieved.

- **Understanding and exploiting constraints that encapsulate facets of every possible solution of a problem.**
- **Some of these constraints can be denoted as landmarks of the solution.**
  - Landmarks **must be true** at some point in every plan generated.
- **Landmarks can be ordered to dictate the order in which they should be achieved.**
- **We aim to discover landmarks and their ordering automatically from the problem.**

LANDMARKS

There are three key facts we're interested in for this chapter

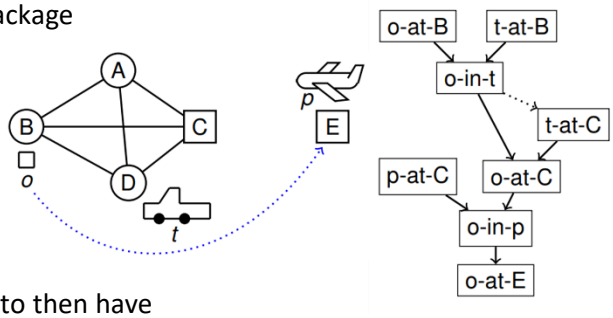
- **Fact Landmarks**: variables that need to hold specific value in at least one state in the final plan's execution
- **Action Landmarks**: An action that must be applied in the solution
- **Disjunction Action Landmark**: Set of possible actions that can be applied in the solution, giving us a little freer reign.

Returning to our example of FreeCell Solitaire. Landmarks in FreeCell could be the fact that the Ace cards appear on all the foundations, or the set of actions that move cards off a tableau and onto a foundation. There are many possible landmarks – either facts or actions – that will prove very useful at some point during the execution of the plan.

EXAMPLE

Let's look at the example of the logistics domain, where we have a package O and we want to get it from the starting location B and in the goal have it reach location E.

In order to do this, we will have to use the truck t and the plane p to get it there.



Through analysis, we can generate numerous fact landmarks for this problem (shown alongside). We can see at the top that having both package and the truck are good landmarks, since this allows us to then have the package loaded onto the truck. Once we get the package and plane to C, we can get the package onto the plane and ultimately to the final destination.

HOW TO WE FIND LANDMARKS?

It is important to find good landmarks as many landmarks can be rather uninformative. Technically every variable in the initial state is a fact landmark. But that's not at all useful since all of these get satisfied once you start searching. The problem is that finding landmarks is just as hard as planning itself (PSPACE complexity).

Finding landmarks is **P-Space Complete** since we know that planning itself is PSCPACE complete and in order to prove an action or fact is a landmark, it's the same as deciding if you can solve the problem without either

- the action itself, or
- in the case of the fact, the action that generates the fact

i.e., you are just planning again.

- **We also need to find good landmarks, given some will not be informative.**
  - The set of all possible actions is technically a disjunctive action landmark.
  - Every variable that is true in the initial state is a fact landmark.
- **But finding landmarks can be as hard as planning itself (PSPACE-complete).**
- **Can exploit the relaxed planning graph technique to generate landmarks.**

We can use the following ways to find landmarks

- **Deletion Relaxation with RPG**
- **Backchaining**
- **RPG Propagation**

FINDING LANDMARKS #1: DELETION RELAXATION WITH RPG

This is the easiest way to generate landmarks.

- Iterating through all actions, perform the following
  - Remove an action if it adds a fact to the planning problem
  - Build the relaxed planning graph
  - If the goal no longer appears in the RPG, then the action was a landmark of the problem

The problem with this method is that it is **slow to execute** since we're trying each of these actions one at a time and then essentially attempting to just run a relaxed plan solution as normal.

FINDING LANDMARKS #2: BACKCHAINING

In this instance we treat every goal as a landmark and then analyze how these goals are satisfied. Then find the actions that satisfy that goal and what preconditions they all share. If we have 3 actions that all have the same precondition A, that generates that goal landmark B, then we know that the precondition A is a landmark.

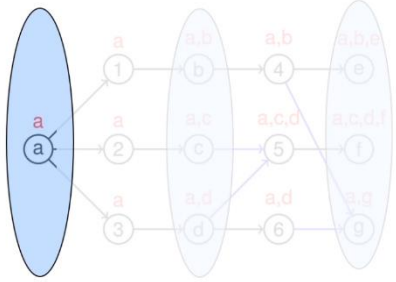
Plus, we also know that these new landmarks are ordered before the goal landmark.

- **Treat each goal (B) as a landmark, for each goal:**
  - Take intersection of preconditions of all actions that achieve goal (A)
  - If numerous actions share A and achieve B, **A is also a landmark**;
  - A is also ordered before (must be achieved before) the goal.
- **Repeat for all landmarks found.**
- **Can do a bit better (find more landmarks) by looking whether A is needed to achieve B for the first time (build RPG without A, see if B appears).**
  - e.g. suppose there is another location F joined to E, we can drive from F->E to achieve the goal but not unless we already went through E…

This can further be optimized by also looking to see whether A is needed to achieve B the first time it occurs – by running RPG without A and see whether B appears.

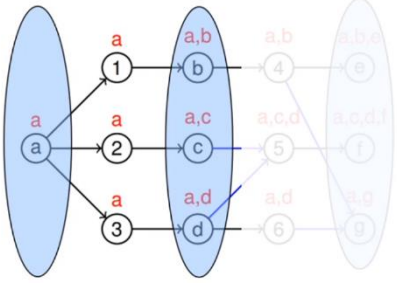
FINDING LANDMARKS #3: RPG PROPAGATION

The idea is to use RPG approach and pay attention to what facts hold true during the execution of a plan. This propagation, whereby we take the union of preconditions of actions that achieve specific facts, helps us to establish what facts could be landmarks for the problem.

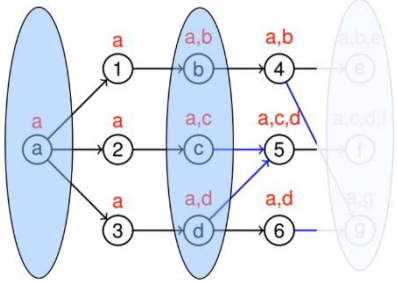


Consider this example, a is the only fact holding true in the initial fact layer.

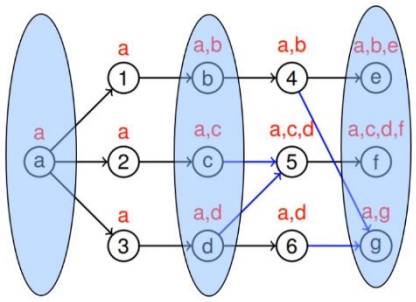
As we roll out all valid actions 1, 2 and 3 we now have more facts, b, c and d. This creates a situation where we have 3 parallel conjunctions of facts: (a, b), (a, c) and (a, d).



But as we generate the next set of actions, we see that action 5 requires c and d as preconditions. Hence, we now have the a, c, d as the landmark facts in this instance. As this rolls out across all actions, we see the different effects of actions as add effects are implemented, resulting in different fact unions being established.



As we move to the final fact layer and look at fact g, the only landmark that exists for it is a, because it is the only fact that is consistent with the preconditions of all the actions that will ultimately get us there.



RPG PROPAGATION: EXTRACTING LANDMARKS

Simply repeat the process until the goals appear and once, they have we keep generating new action/fact layers until the labels no longer change. At which point, we can then grab the goals in the final layer and check to see what fact layer unions are attached to them and we have out fact landmarks, generated within polynomial time.

- RPG Propagation finds all casual delete relaxed landmarks in polynomial time
- Build the RPG not just until the goals appear but until all goals appear and we reach a layer with identical labels to the previous one (labels stop changing)
- The landmarks are the labels on the goal nodes in the final layer
- Possible first achievers of B are achivers that do not have B in their label

LANDMARK ORDERING

It can be useful to know the order in which landmarks must be achieved. There are 2 types of orderings

Sound Ordering

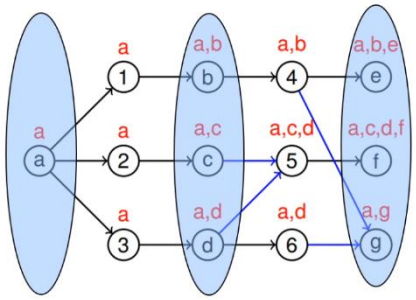
- **Necessary Ordering**: $A \rightarrow_n B$: A is always true one step before B
- **Greedy-Necessary Ordering**: $A \rightarrow_{gn} B$: A is true one step before B is true for the first time
- **Natural Ordering**: $A \rightarrow B$: A is true some time before B
- $A \rightarrow_n B \Rightarrow A \rightarrow_{gn} B \Rightarrow A \rightarrow B$

Unsound Orderings (used in heuristics)

- **Reasonable Ordering**: $A \rightarrow_r B$: if B was achieved before A then the plan should delete B to achieve A and re-achieve B after (or at the same time as A).
- **Obedient Reasonable Ordering**: $A \rightarrow_{or} B$: if B was achieved before A then any plan that obeys reasonable orderings must delete B to achieve A and then re-achieve B after (or at the same time as A).

So given what we have already seen in this RPG rollout, we can then determine what the landmark orderings are in this instance.

- a has a natural ordering over e, f and g as it occurs at some point prior to their appearance
- For f both c and d have greedy necessary ordering over f as they both occur in the preceding fact layer for the first time, while f also appears for the first time.



SEARCHING WITH LANDMARKS

If we have processed the problem before beginning the planning phase and have generated the landmarks (and their orderings), we can use them during the search phase in 3 different ways

- Landmarks as **Planning Subgoals**
- Landmarks as **Heuristic Estimates**
- **Admissible Landmark Heuristics** (not in the scope of the course)

LANDMARKS AS SUBGOALS

The approach to using landmarks as subgoals is to plan to each set of landmarks and repeat until all landmarks are satisfied. The following is the method

1) Given some landmark and a (partial) ordering on them
2) Set the goal to (a disjunction over) the first landmark(s) according to the landmark graph and find a plan
3) Update the initial state, plan to the next landmark(s). Repeat

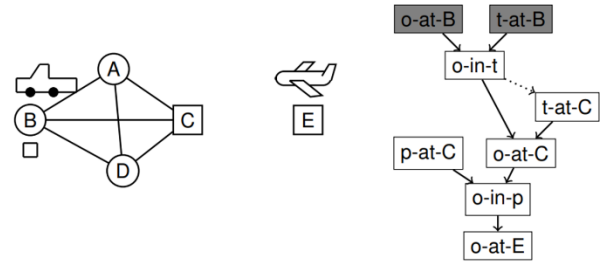
PLANNING BETWEEN LANDMARKS

Returning the logistics domain, in this instance, we have two goals

- One of the earliest landmarks: the package is in the truck
- Plane being at location C

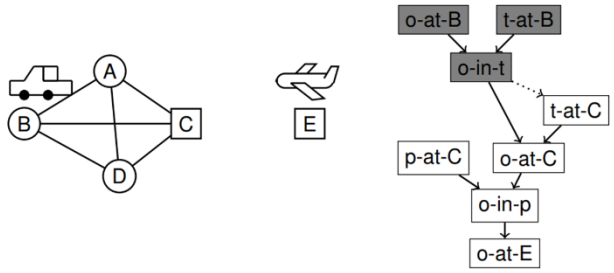
We can now plan towards solving one of them.

We put the package in the truck, satisfying the landmark and adding a new action to our partial plan, i.e. `Load-o-B`.



- Partial plan: Drive-t-B
- Goal: o-in-t ∨ p-at-C

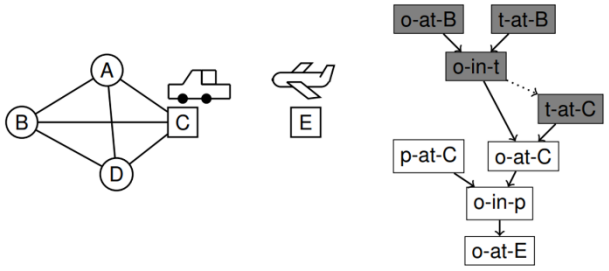
Now we have a new landmark to achieve, which is to have the truck at C, which we know happens some time before the package is at C, thus its ordered first.



- Partial plan: Drive-t-B, Load-o-B
- Goal: t-at-C ∨ p-at-C

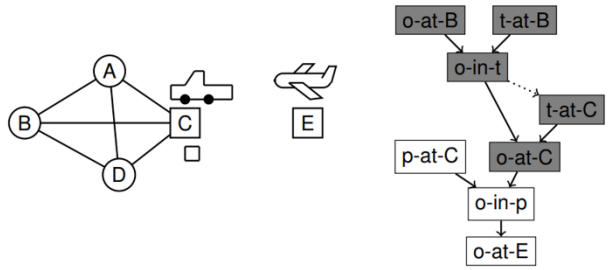
We drive the truck to C satisfying the landmark. Thus now the new landmark is `o-at-C`, i.e. the package being at C.

We also have to keep in mind that we have not worked on the `p-at-C`, i.e. the plane being at C landmark.



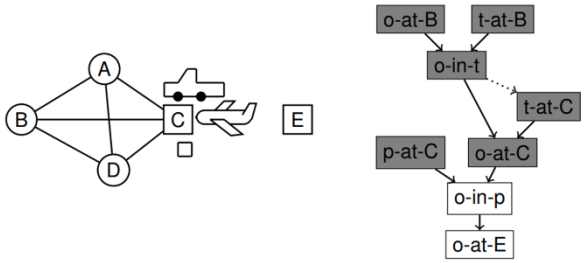
- Partial plan: Drive-t-B, Load-o-B, Drive-t-C
- Goal: o-at-C ∨ p-at-C

Once the package has been removed from the truck, satisfying the `o-at-C` landmark, we only have one landmark to work towards, and that is `p-at-C`.



- Partial plan: Drive-t-B, Load-o-B, Drive-t-C, Unload-o-C
- Goal: p-at-C

Now that the plane has reached point C as well, we have satisfied the landmark `p-at-C`.

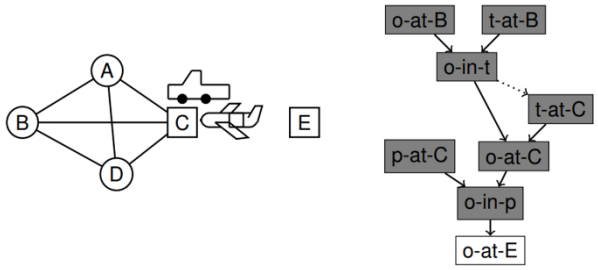


Now we have a new landmark, which is the package being in the plan (`o-in-p`).

- Partial plan: Drive-t-B, Load-o-B, Drive-t-C, Unload-o-C, Fly-p-C
- Goal: o-in-p

We have now satisfied all but one landmark (o-at-E). In order to satisfy this landmark, we will require 2 more actions, i.e. flying p to E and the unloading o to E.

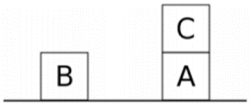
The last 2 steps are straightforward and get us to out goal state.



- Partial plan: Drive-t-B, Load-o-B, Drive-t-C, Unload-o-C, Fly-p-C, Load-o-p
- Goal: o-at-E

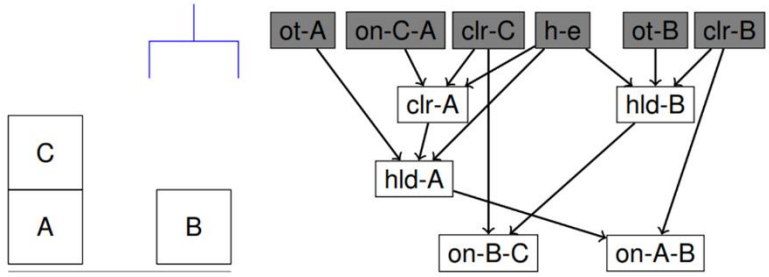
SUSSMAN ANOMALY

The Sussman Anomaly is a well-known situation in BlocksWorld, where each goal, if tackled in isolation, will impede the other one. Either the planner will put B on C and then A can't be moved. Or it puts A on B and forgets it still needs to put B on C.



We can consider the following landmarks for the initial state of the problem

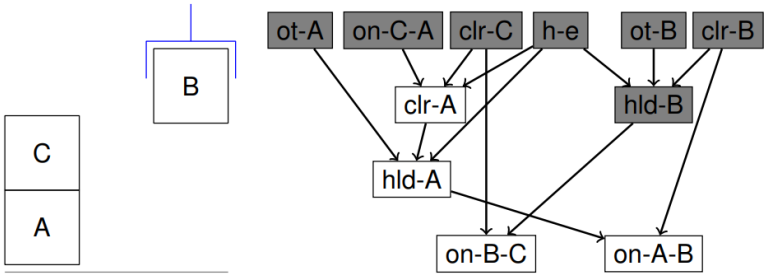
- Clear A
- Holding B



- Partial plan: ∅
- Goal: clear-A ∨ holding-B

Once we satisfy holding B, the following become our sub goals (landmarks)

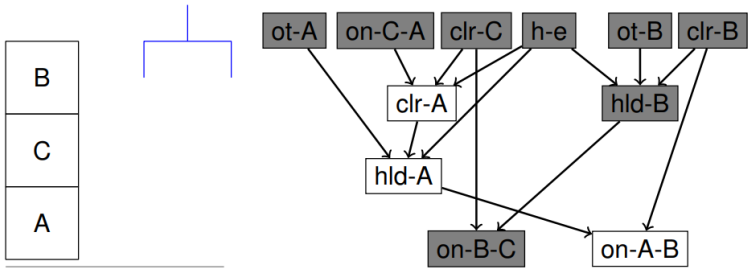
- Clear A
- On-B-C



- Partial plan: Pickup-B
- Goal: clear-A ∨ on-B-C

It satisfies the easiest one, i.e. places B on C (as it is already holding B; dropping B and clearing A is much more work).

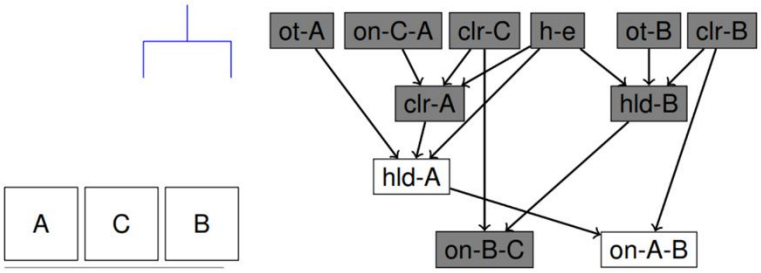
We are left with one sub goal in this case, which is `clear-A`.



This is the Sussman Anomaly. By satisfying a sub goal, we have created more work for ourselves.

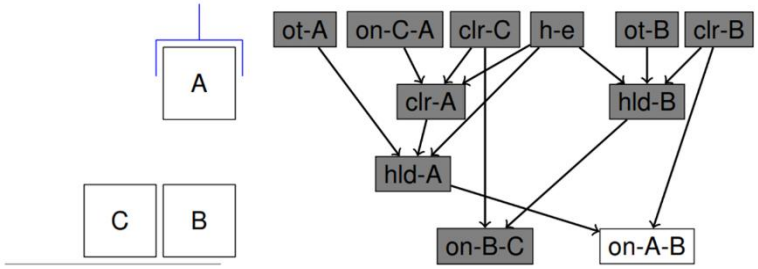
- Partial plan: Pickup-B, Stack-B-C
- Goal: clear-A

If we then clear A, by unstacking B and C, the sub goal is still `hold-A`.



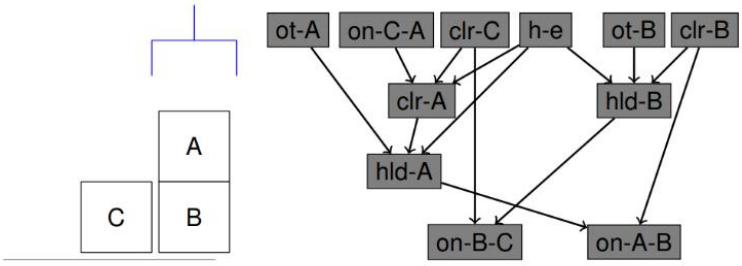
- Partial plan: Pickup-B, Stack-B-C, Unstack-B-C, Putdown-B, Unstack-C-A, Putdown-C
- Goal: holding-A

This leads to A on B being the new goal.

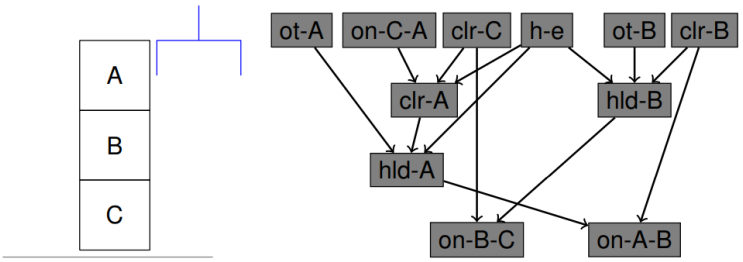


- Partial plan: Pickup-B, Stack-B-C, Unstack-B-C, Putdown-B, Unstack-C-A, Putdown-C, Pickup-A
- Goal: on-A-B

But by doing so, we've created more work for ourselves again. We still need to achieve B on C, but in order to do that, we will have to undo some of the work we have just done.



- Partial plan: Pickup-B, Stack-B-C, Unstack-B-C, Putdown-B, Unstack-C-A, Putdown-C, Pickup-A, Stack-A-B
- Goal: Still need to achieve on-B-C



This means that in order to achieve the goal state, we will have to start from a previous stat all over again, which isnt ideal at all.

THE GOOD AND THE BAD

- Can perform faster planning as the planner needs explore the search space to a lower depth
- Can lead to poor quality plans which are much longer, as sometimes landmarks need to be undone and achieved again if tackled in the wrong order
- Incomplete in problem with dead ends

- Partial plan: Pickup-B, Stack-B-C, Unstack-B-C, Putdown-B, Unstack-C-A, Putdown-C, Pickup-A, Stack-A-B, Unstack-A-B, Putdown-A, Pickup-B, Stack-B-C, Pickup-A, Stack-A-B
- Goal: ∅

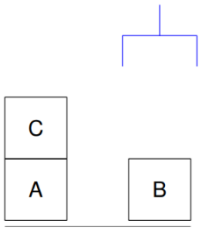
LANDMARKS AS HEURISTICS (LM-COUNT)

- Number of landmarks still to be achieved is a heuristic estimate.
- **LM-count**: A path dependent heuristic.
  - We can't tell which landmarks have been achieved just from whats true in the state

This method was adopted by LAMA (winner of the IPC 2008 Winner Satisfying Track).

PATH-DEPENDENT HEURISTIC

LM-count is known as path dependent heuristic. Unlike the heuristics we have seen till now, LM-Count cannot be derived solely from looking at the state and calculating the value.



If we consider this example from BlocksWorld, can we tell whether we've achieved the holding B landmark?
It can be argued that it is not, since the block isnt being held. But of course the plan is going to continue to evolve over time, and the landmark is just one fact that must hold true at some point during the plan. Thus in this case, the heuristic becomes path-dependent, because it is no longer a function of the observed state, but instead the path that we took in order to reach it.

- Achieved landmarks are a function of the path taken, not an observation of the state

Hence, LM-Count is an **inadmissible heuristic**, since one action could satisfy more than one landmark. Thus it can inflate the perceived value of a state as distance to the goal, despite the fact that this can actually push us much further away from the goal than out current state.

COMPUTING LM-COUNT

LM-Count can be calculated using the following equation

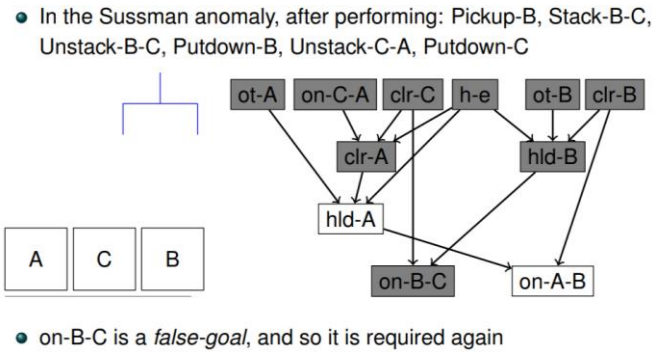
$$\boldsymbol{H(s,p) = \left(L \backslash Accepted(s,p)\right) \cup ReqAgain(s,p)}$$

Where \ means set difference.
For state s and path p, where,

- L is a set of all landmarks discovered for the problem
- $Accepted(s,p)$ are all the accepted landmarks. A landmark is accepted if its true in s and all its predecessors in the landmark graph accepted.

- $ReqAgain(s, p)$: Landmarks that we've seen but know we need to see again. L is required again if
  - Landmarks is false and it is a goal
  - Landmarks is false in s and is a greedy-necessary predecessor of landmark m (must be true the step before m), which is not accepted.

Consider the Sussman Anomaly example. If we have created the situation where we immediately attempted to tackle the on-B-C goal like before, it would be denoted as a false goal. Hence, we need to satisfy it again and from this position, achieving these landmarks is much more achievable than before and the search can be better directed.



DUAL HEURISTICS AND LOCAL SEARCH OPTIMISATION

COMBINING HEURISTICS

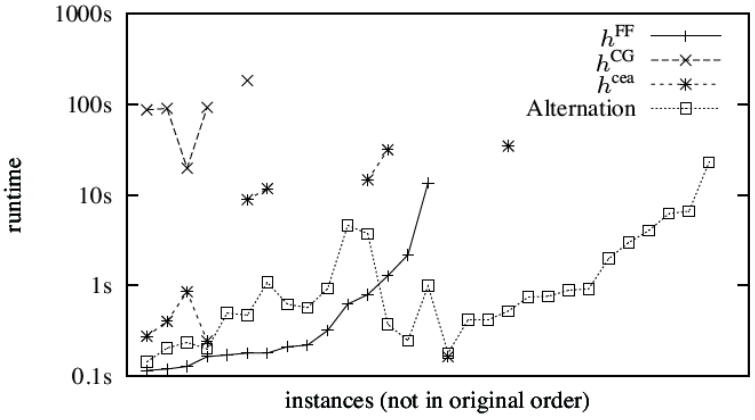
What if we had 2 heuristics; $h_1(S)$ and $h_2(S)$, both estimating the distance and/or cost to the goal?
We could have 2 open lists, one for each heuristic, and alternate between them,

- Expand node from open list 1
- Expand node from open list 2
- Expand node from open list 1

We can then evaluate successors with both heuristics and put them on both the open lists. In each case we're using different types of information to help us search. This can help us explore and assess the state space in two different ways at the same time.

Given alongside is a graph from a paper date back to the 2010 International Conferences of Automated Planning and Scheduling. The planner uses 3 different heuristics, namely

- FF heuristic (i.e. RPG)
- Casual graph heuristic used in the Fast-Downward Planner
- Context-enhanced additive or CEA heuristic from the author's work back in 2008



Each of these heuristics is performing well in some circumstances and not well in others. But we can see this alternation set, and this is where the planner is alternating between heuristics and as a result, the performance is more consistent across the board.

Thus, the **second heuristic is useful** in the following scenarios

- If one heuristic is on a **plateau**: Where there is no obvious next state to explore given all of the heuristics have stagnated at the same value, we could use the second heuristic to help us find a way out
  - Plateau successors are put on both open lists
  - Second open list will order them by second heuristic so will be guiding search on the plateau
- **Diversification** by avoiding expanding just the states one heuristic prefers.

LOCAL SEARCH

It's important to acknowledge that Enforced Hill Climbing (EHC) as we saw back in FF is a procedure known as Local Search. We start with a given state, expand it, generate all the successors and find the best one. We then transition to the best successor and repeat until we reach the goal.

Now the idea of the *best*successor is what is governed by the heuristic. We're not maintaining an open list this time like A* does, because we're only interested in small, local optimizations that we make near where we are. This process is often referred to as hill climbing in state space search, but is essentially a form of gradient descent, given we're aiming to take steps based on the heuristic value as means to reach the local minimum of the fitness space.

- **In four bullet points:**
  1) Start with a state S
  2) Expand S, generating successors
  3) S' = one of these successors
  4) S = S' and repeat
- **Heuristics are used at (3)**
  - e.g. choose the best successor
  - Known as **gradient descent** (or **hill climbing**).

THE TROUBLE WITH HEURISTICS

We know that the RPG heuristic is not perfect. It is giving us a relaxed plan solution which isnt always going to be reflective of the actual problem. But as discussed already, we're looking for something that will compute in polynomial time. Because otherwise the heuristic would prove useless and will slow the process down.

Now there are problems with this, given if we commit an action – which may well be the right action in this situation – it might not change the heuristic value in the direction we hope. It could go up; it could go down. It might not change at all.

- **The RPG heuristic is not perfect**
  - Few are, otherwise the problem would be easy!
  - … or the heuristic would be very expensive to compute
- **Making the right move doesn't always lower the heuristic value:**
  - It can stay the same; or,
  - It can go higher.
- **Or, worse, the move with the best heuristic value can be a really bad decision:**
  - With local search, this can lead to no solution being found.

And as we saw in FF, there are going to be cases where taking the state with best heuristic value might work out to be a bad thing. Given we may wind up in a plateau or in a dead-end and hence the use of best-first search as a fallback.

SNAGS WITH EHC

Enforced Hill Climbing isn't going to work well in certain situations. The heuristic will result in dead ends, we have to rely on the best first search algorithm as a fallback. This isn't ideal given we're going back to the initial state. We can't rely on the algorithm to have not jeopardised our search process and not put us in a dead-end.

In addition, relying on the system to search plateau's is also not going to help, given it can be quite expensive.

- **Sometimes can fail to find a solution:**
  - The heuristic can lead it to **dead ends**
- **If FF fails, resorts to systematic best-first search from the start:**
  - Priority queue of states (lowest h(S) first)
  - Expand the next state each time
  - Add successors to the queue, and loop
- **Also, searching on plateaux is expensive:**
  - **Systematic** search on the part where the heuristic is worst

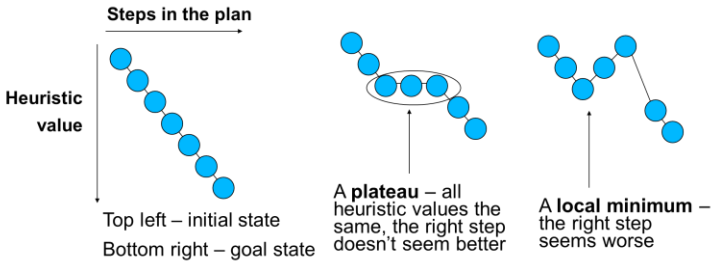
THE SEARCH LANDSCAPE

To help visualise some of the problems here, we can look at the landscape of search as it rolls out.

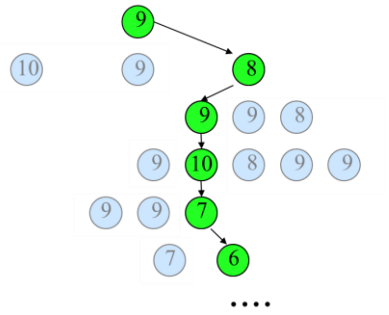
Ideally, we want the situation on the left (in the diagram given below); we start at the initial state with a high heuristic, we continue to find new states with lower values and continue onwards. Ultimately reaching the goal.

But there are two really nasty situations that we either wish to avoid or find means to get out of

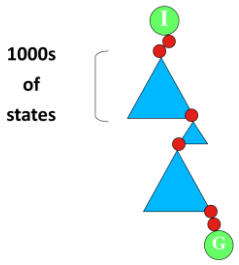
- Plateau: This is where the heuristic values are the same, no step in any direction seems like an improvement. The plateau might be short, it might be long, but we need to find a way to get out of it.
- Local Minimum: This means we've reached a state that is clearly the 'best', but there are no better options. There isnt even a plateau and no options in the proximity will improve the situation, but there are better options out there, so we want to find a way to avoid this from happening.



Consider the example given alongside. Using EHC, the algorithm goes from the initial state ($h = 9$) to the next state ($h = 8$). At this point, there is no better option in its proximity, however, there is a better state in the plan (i.e. $h = 6$). Thus in order to get there, the planner needs to leave the local minimum and then get to the optimal state.
In this case the planner uses breadth first search to get out of the local minimum and eventually find the optimal state with $h = 6$.



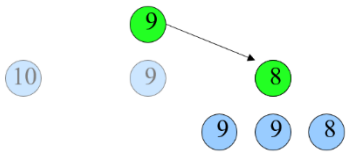
Even though this was feasible in this scenario, it might not be feasible to perform breadth first search in cases where thousands of states need to be traversed before reaching the optimal state. Consider the example given alongside, one might have to perform breadth first search only to get stuck in another local minimum through EHC. This process might have to be repeated multiple times in order to reach the goal state. Even though we are finding the goal, it is painfully sub-optimal. In this situation, the planner is mostly running breadth-first search rather than EHC. The heuristic is having very little impact on the performance of the search at this point.



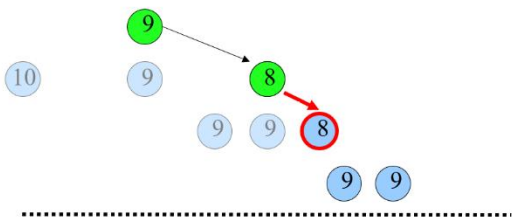
IDENTIDEM – AN ALTERNATIVE TO EHC

- **Replace systematic search on plateaux with local-search with restarts.**
- **As in EHC: record the best state seen so far.**
- When expanding a state S at the route of a plateau:
  - Choose one successor, **at random**, even if it's not better than the best seen so far (it's not as we're on a plateau);
- If more than *d* steps since the best state: jump back to the best and search from there again:
- *d* is the **probe depth bound**

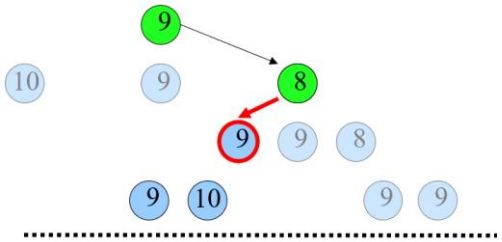
We operate the same as before, only this time, when a plateau is found, we choose one of the states at random, exploring down a path to a certain depth. If we do not reach a new improving step within a certain number of iterations – known as **probe depth bound** – we start again, but this time we search down a different path. The actual probe depth bound can increase over time if no solutions are found but is set to be an initial value on start.



Consider the example given alongside. Once we have reached the state with $h = 8$, we are stuck on a plateau. There is no better state at this point to go to using EHC. Instead of using BFS we can use Identidem to get out of the plateau. Let's say that initially, the probe depth bound (d) is 2.
Following the process of Identidem, we pick a node at random, and search along its nodes (until depth of 2) to find a better state. Assuming that we expanded on the right most node ($h = 8$), as shown alongside, we reach yet another plateau.



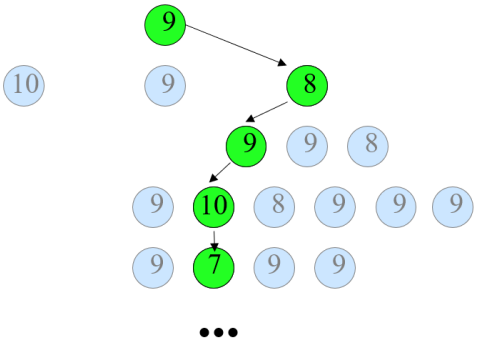
Thus, we randomly select another node. This time we select the left most node ($h = 9$), but still cannot find a better state.



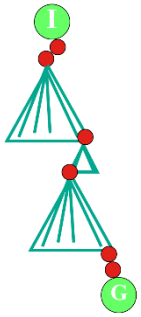
Once we have traversed the states with $d = 2$ and found no better state, we can increase the probe depth bound to 3 and search again.

This way we find the state with $h = 7$, which is better than the current state ($h = 8$). Thus, we can get out of the plateau.

The probe depth is important, given we want to find a way to increase the search, but also minimize how deep we search at different periods of time given we don't want to just conduct a deep random walk into the search tree.



And while ultimately this winds up being markedly like the normal behaviour of EHC, the big different is that now instead of this very expensive process using breadth first search. Instead, we're now reliant on this more structured local search through the plateau. This will cut down on excessive exploration that doesn't yield improvement in the best case and in the worst case it will prove no better than breadth first search.



LOCAL SEARCH – 2 KEY TERMS

- **Diversification**: Try lots of different things, to try to brute force around the weaknesses of the heuristic
- **Intensification**: Explore lots of options around a given state (e.g. near to where the goal might be)

Local search is a balancing act.

- Long probes = good for diversification
- Short probes = good for intensification

DIFFERENT NEIGHBOURHOODS

When expanding a state S, a neighbourhood function is used.

FF

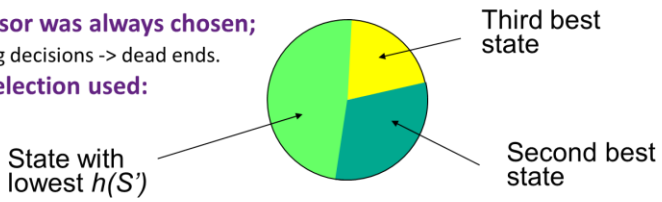
- When running this in FF, we're generating the simplest possible neighbourhood of states. It's all states reachable from S.
- The problem here is that not all these successors are going to be interesting

Identidem

- When running this in Identidem, we're only generating one successor, but we're evaluating them all first.
- This has pros and cons, given we're only evaluating the successors, but we can focus on only evaluating a subset of the neighbourhood each time Identidem does the local exploration.
- This ties in well with the restarts, given the subsets can switch up each time and help us diversify where we explore on each iteration.

SUCCESSOR SELECTION

- **In EHC, the 'best' successor was always chosen;**
  - Imperfect heuristic -> wrong decisions -> dead ends.
- **In Identidem: roulette selection used:**



- **Make a random choice, biased by heuristic values**

Once this evaluation is completed, we use a roulette wheel selection process: whereby if you think of it as a wheel that you spin for the selected region, the size of region of the wheel a given state has is biased based on the heuristic value, with the lowest having the largest segment. Having three to pick from at random has proven to work quite effectively, given we still achieve some diversification since there is randomness still being employed here.

DOES IDENTIDEM PERFORM BETTER THAN FF?

In short? Yes. But in truth, only most of the time.

- Systematic search is better in some domains
- Sometimes the parameters need changing
  - Random sample size, depth bound, whether to use roulette selection, etc.

SYSTEMATIC SEARCH VS LOCAL SEARCH

Systematic Search	Local Search
• Good for puzzles with lots of bad moves (keeping one successor = probably a wrong move) • Good at **proving optimality** • Are provably **complete** – they will find a solution if one exists	• Generally **quicker** at finding some solution • Using them repeatedly usually gets **near-optimal solution**s fairly quickly, but no proof • **Incomplete**, but can always run systematic search afterwards (e.g. EHC, then Best-First)
Systematic search is useful for proving optimality of a solution as well as finding a solution if one of them exists. Hence it can be very good in situations like puzzles, where if we hit a plateau of lots of bad decisions, we can feel our way out of that space.	Local search like that seen in Identidem is generally quicker at finding a solution that gets us out of that plateau, but is incomplete, we can't guarantee it's going to get us a solution at all. Hence with the likes of EHC we use best-first as a fallback.

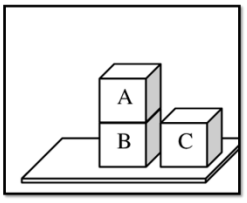
OPTIMAL PLANNING WITH SAS+

COMPLEXITY AND STATE SPACES

PDDL defines predicates for each domain. A proposition is a predicate with parameter bindings. A proposition is either true or false, at any given point. One proposition exists for everything that could possibly be true. This has its drawbacks.

Let's consider the propositions in a given state of the BlocksWorld domain. In this case, we have 5 propositions that are true (the ones highlighted alongside). But we are also encapsulating 7 other facts that are false, bringing it to a total of 12. This results in $2^{12} = 4096$ states (not all reachable). This is a ridiculous number of possible states, and while not all of these states are reachable, it is evidence of exponential growth in the state space.

s(A-on-B) = T	s(C-on-A) = F
s(A-on-C) = F	s(C-on-B) = F
s(A-on-table) = F	**s(C-on-table) = T**
s(B-on-A) = F	**s(A-clear) = T**
s(B-on-C) = F	s(B-clear) = F
s(B-on-table) = T	**s(C-clear) = T**



By adding just one new block, the number of propositions increased by a further 8, given we model where D is atop, but also whether the existing blocks are on D and of course if D is clear. This increases the number of unique combinations from 4096 to 1,048,576 (not all reachable).

Even though the PDDL actions will ensure that we only visit a subset of the total list, we can see that as problems increase in complexity, the state spaces are exploding in scale accordingly.

s(A-on-B) = T
s(A-on-C) = F
s(A-on-table) = F
s(B-on-A) = F
s(B-on-C) = F
s(B-on-table) = T
s(C-on-A) = F
s(C-on-B) = F
s(C-on-table) = T
s(A-clear) = T
s(B-clear) = F
s(C-clear) = T

S(A-on-D) = F
S(B-on-D) = F
S(C-on-D) = F
s(D-on-A) = F
s(D-on-B) = F
s(D-on-C) = F
s(D-on-table) = T
s(D-clear) = T

By having an **expressive** formalism, state spaces can begin to explode. We try to circumvent his by using heuristics to guide the search, landmarks to give us sub-goals to solve or use local search (like EHC), often **sacrificing completeness** of search to find a solution quickly. What if we instead **sacrifice expressiveness** for **efficiency**? What if instead of using this original encoding to define our search space, we change the expressivity of the problem to help reduce it?

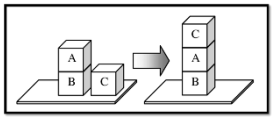
INVARIANTS

Invariants are logical formulae that are true in all reachable states in the planning problem. For example: $\phi = \neg(at\_uni \wedge at\_home)$.
The idea is to make such facts which seem obvious to us, obvious to the planner as well.

Testing whether an invariant is valid can be as hard as planning itself (as it involves proving that certain facts do not change during the solving of the problem). But, if we can find invariants and encode it in a way a planner can understand, we could potentially reduce the state space. Even finding obvious invariants could drastically improve our performance in solving the planning task.

MUTUAL EXCLUSION

It is one of the most commonly used invariant. Mutual exclusion (**mutex**) is an invariant over a set of propositions where states only can be true at any given point of time. This can be extended out to sets of literals.
Looking back at a previous example, being at university and being at home is a mutex, as only one of those two propositions can be true in any given state.
Example: in the Blocksworld domain, block C cannot be on both blocks A and B, thus that is an invariant.



$\neg(ConA) \vee \neg(ConB)$ thus $ConA$ and $ConB$ are **mutex**
Another example is $\{AonB, ConB, Bclear\}$ is also a **mutex** as every pair in the set is a mutex.

FINITE-DOMAIN STATE VARIABLES

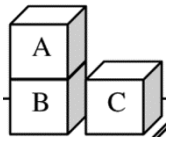
We can use invariant literals to reframe a problem using Finite-domain state variables. We create a variable v and a domain $dom(v)$ that expresses the range of possible values that can be assigned to v. We can then create **Finite Domain States** that express a state of a planning problem as an assignment to all variables in the state V.

$$s(v) \in dom(v) \quad \forall\, v \in V$$

Example, to express the previously mentioned mutex, we can use the following representation

$$v = below\_a, \qquad dom(below\_a) = \{b, c, table\}$$

Thus we can revisit BlocksWorld, but this time using propositions and Finite-Domain State Variables. For example, to express the state as shown in the picture given alongside, we would use the following 3 finite-domain state variables:



- $below\_a : \{b, c, table\} = b$
- $below\_b : \{a, c, table\} = table$
- $below\_c : \{a, b, table\} = table$

If we choose to encapsulate the other relevant information as well, we will need 3 more variables:

- $above\_a : \{b, c, nothing\} = nothing$
- $above\_b : \{a, c, nothing\} = a$
- $above\_c : \{a, b, nothing\} = nothing$

This means that we can represent the BlocksWorld domain using only $6^3 = 216$ states (as opposed to the 4096 which we had earlier).
There are even more constraints that we can apply to these: for example, above_a and below_a cannot have the same value.

What makes Finite Domain Representation (FDR) useful is the fact that we can translate any FDR back into the propositions mentioned earlier. We can create **Finite Domain Formulae**, whereby we can now express rules or constraints we wish to have over a given problem space but do so in a way that can later be reduced to their atomic propositions that we saw earlier.

PLANNING AS SATISFIABILITY (SAS+)

The aim is to translate PDDL into finite-domain representation encoding, capturing the mutex relations, so that we can then solve constrained-enforced variation.

An SAS+ planning task is a 4-tuple $\Pi =< V, O, s_0, s^* >$ with the following components:

- $V = \{v_1, \dots, v_n\}$: A **set of state variables** (or fluents) V, each with an associated finite domain D_v. If $d \in D_v$ we call the pair $v = d$ an atom.
- O a **set of operators**, where an operator is a triple $< name, pre, eff >$
  - $Name$ of the operator
  - Pre and eff are partial variable assignments (preconditions and effects)
- s_0 is the initial state, and s^* is a partial assignment of atoms representing the goal

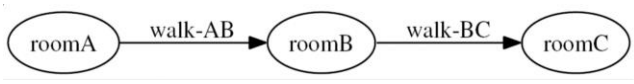
SAS+ OPERATORS (ACTIONS)

There are two sorts of conditions that roll PDDL preconditions and effects together:

- **Prevail Conditions**:
  - Variable = value that stays the same
  - When loading a package p_1 into a truck t_1 at location l_1: $< t_1 at\_l_1 >$
- **Pre-Post Conditions**
  - The value of a variable changes from one value to another
  - When loading a package p_1 onto a truck t_1 at location l_1: $< p_1, at_{l_1}, in\_t_1 >$

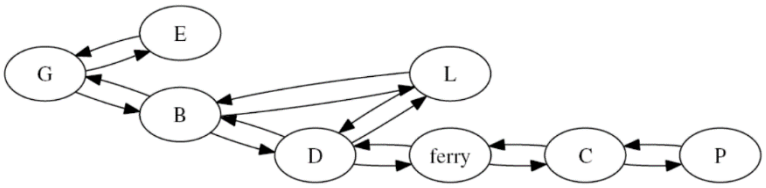
DOMAIN TRANSITION GRAPHS (DTGS)

- **Domain**: One for each variable referring to its domain
- **Transition**: pre-post conditions of operators change the values of a variable
- **Graph**: Transitions in the domain form a directed graph
- **Nodes**: the domain values



- **Edges**: An edge exits from $A \rightarrow B$ iff an operator exists with a pre-post condition $< v, A, B >$

Example: Consider the inter-continental logistics problem given alongside. The nodes of the graph represent the locations which the truck can be at. Thus the entire planning problem has been reduced to one variable which has 8 possible values (i.e. each node).



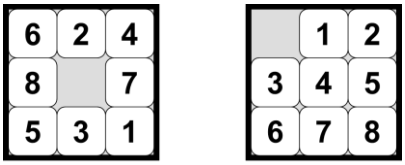
OPTIMAL PLANNING – PATTERN DATABASES

Aim: to store solution costs for sub-problems within the search space.

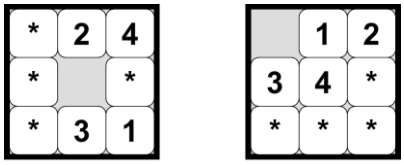
- The solution costs to sub-problems will provide an admissible heuristic to solve the actual problem
- Heuristic calculation becomes very fast (as it is now a database lookup)

This is achieved by searching backwards from the goal and recording costs the states. It is an expensive process, but only needs to be done once.

Consider the example given alongside. The image on the left is the initial state and the image on the right is the goal state.



In order to make the problem simpler, we can ignore some of the tiles as shown alongside. By removing numbers 5 through 8, we can calculate a solution that ensures that 1-4 are in the correct positions. By removing the numbers on the tile, we have created a **sub problem** of the original problem. Now, if we were to calculate the solution to this problem, the actual cost of the solution (in this case the number of moves made), would be either less than or equal to the number of moves required to achieve the optimal solution in the original problem. Thus, solving a sub-problem optimally will always produce an admissible heuristic to the original problem.



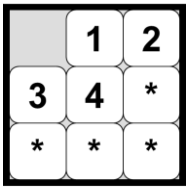
PATTERNS

A Pattern is a partial specification of a given state in the problem. This is actually a representation of what we could call an abstraction of the state space.

The **Target Pattern** is a partial representation of the goal state. We want to find all permutations of the target pattern, i.e., different partial specifications of the goal. Once all the target patterns are computed, we can then calculate the pattern cost, which is going to be the minimum number of actions to achieve the target pattern.

Pattern Database is a set of all permutations of the target pattern.

Pattern



Target Pattern

PATTERN DISCOVERY

In order to discover patterns for a given problem we search the abstraction of the state space using breadth-first-search, moving backwards from the goal state. Thus the actions applied are the inverse of the original. Once we reach the initial state, we can calculate how many actions are applied in the abstract state space and that's the **pattern cost**.

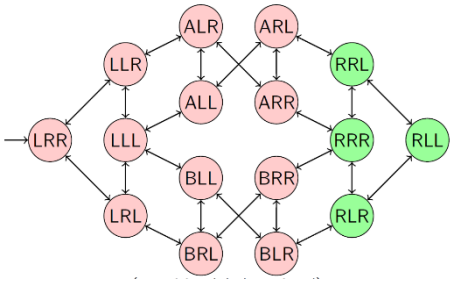
The reason we move backwards from the goal state is because we have a much shorter search time due the abstraction.

ABSTRACTIONS

Pattern Discovery explores the abstract state space. Our target patterns exist within the abstract state space. Since we only care about specific values in any given state (abstraction of the state space), the number of unique states is now smaller.

Abstractions are achieved by applying a function α on a state space S, such that $\alpha(S) = S'$, where S' is the abstraction that state space. It is possible for multiple state spaces to map to the same abstraction, i.e., $\alpha(S_1) = \alpha(S_2)$. Thus in **abstract state space transition system**, similar states are now indistinguishable.

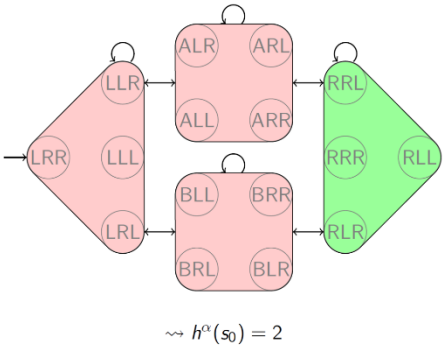
Abstraction Heuristic (target pattern cost): Heuristic estimates the cost-to-goal in the abstract transition system.



Example 1: Driver log problem with 2 trucks and one package. In the graph shown alongside, for the sake of cleanliness, labels have been omitted. Each node has 3 letters corresponding to the package, truck A and truck B. The trucks can be in either locations L or R, whereas the package can be in either locations L or R, or in trucks A or B.

Initial State: Package is at location L, and both trucks are at location R.

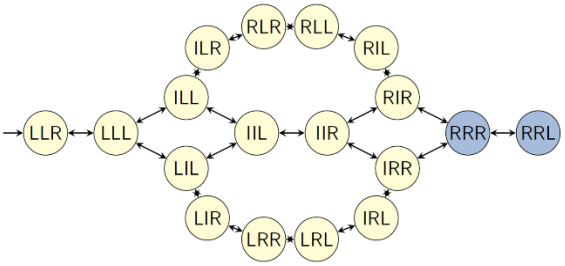
Goal State: Package is at location R (irrespective of the trucks location).



By applying the abstraction to only consider the location of the package, we can abstract the entire state space into four states. We can see the groups are predicated on whether the package, is either L, A, B or R. The transisions in the abstract state space, are either to the same abstract state or to another abstract state. Hence in this case our target pattern is the abstract state on the right hand, which is to have the package at location R, but it's actually an abstract state comprised of four different states.

We can search back and count that it will take only 2 actions to get from the abstract state which represents the initial state, to the abstract goal state. Thus the heuristic value is 2 in this case.

Example 2: Same domain, with a different state space as shown alongside. There are 2 packages and only 1 truck.



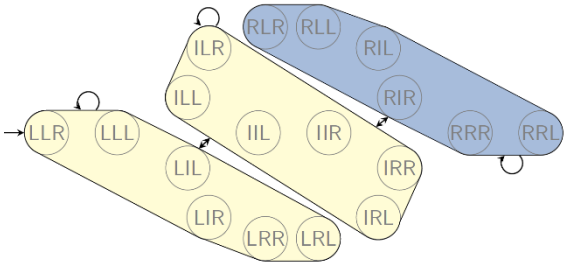
Initial State: Package 1 and 2 are at L and the truck is at R

Goal State: Both packages need to be at R (irrespective of the location of the truck)

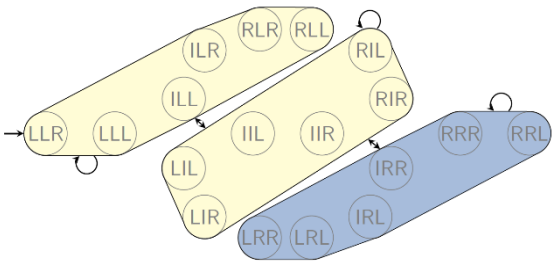
The first abstraction function we apply is only interested in the location of the first package. We have three abstract states.

- First Abstract Space: Package 1 is still at location L
- Middle Abstract Space: Package 1 is in the truck
- Final Abstract Space: Package 1 is at location R (irrespective of the location of the second package).

In this case as well, we have a heuristic value of 2.



If we apply a different abstraction function where we are only interested in the location of package 2, we see a similar effect. Once again there is a heuristic value of 2, but this time the method of exploring and breaking down the state space is slightly different.



PLANNING WITH PATTERN DATABASES

Planning with patter databases is done using SAS+ encoding of the problem.

- Create a set of all state propositions in mutex groups
- Construct abstract problem space(s)
- Construct Pattern Database
- Run planning process from initial state.

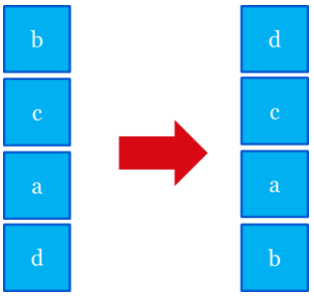
Example: BlocksWorld domain, with 4 blocks.

Initial State:

- B on C
- C on A
- A on D
- D on Table

Goal State

- D on C
- C on A
- A on Table
- Don't care about B



First, we generate the mutex groups using SAS+ encoding of the problem space. Given alongside is the collection of sets of mutex relations. These are used as the basis for our particular abstract state space searches.

- G_1= {(on c a), (on d a), (on b a), (clear a), (holding a)},
- G_2= {(on a c), (on d c), (on b c), (clear c), (holding c)},
- G_3= {(on a d), (on c d), (on b d), (clear d), (holding d)},
- G_4= {(on a b), (on c b), (on d b), (clear b), (holding b)},
- G_5= {(ontable a), true},
- G_6= {(ontable c), true},
- G_7= {(ontable d), true},
- G_8= {(ontable b), true}, and
- G_9= {(handempty), true},

Next, we split them into groups. We can do this in a process as simple as alternating between odd and even sets as shown alongside.

- G_1= {(on c a), (on d a), (on b a), (clear a), (holding a)},
- G_2= {(on a c), (on d c), (on b c), (clear c), (holding c)},
- G_3= {(on a d), (on c d), (on b d), (clear d), (holding d)},
- G_4= {(on a b), (on c b), (on d b), (clear b), (holding b)},
- G_5= {(ontable a), true},
- G_6= {(ontable c), true},
- G_7= {(ontable d), true},
- G_8= {(ontable b), true}, and
- G_9= {(handempty), true},

Next, we split the goals as well and say okay, for A the even goal which is (on C A), we then generate all the heuristic values for all combinations of facts across those sets with respect to that goal. So in the case of the even goal (on C A), there's not that many non-mutex pairings here that we can consider as possible patterns. So the list is quite short and working back from the goal state it's not as hard.

However, in the case of the odd goals, (on A B) and (on A C), the number of combinations of facts generating new patterns is quite large. Hence, we have a much larger number of possible patterns and their corresponding heuristic values.

EVENOUT(GOAL) = (on c a)

- ((clear a),1)
- ((holding a),2)
- ((on b a),2)
- ((on d a),2)

ODDOUT(GOAL) = (on a b), (on a c)

- ((on d c) (clear b),1)
- ((on a b) (clear c),1)
- ((on d c) (holding b),2)
- ((clear c) (clear b),2)
- ((on d c) (on d b),2)
- ((on a b) (holding c),2)
- ((on a c) (on a b) ,2)
- ((clear c) (holding b),3)
- ((clear b) (holding c),3)
- ((on a c) (clear b),3)
- ((on d b) (clear c),3)
- ((holding c) (holding b),4)
- ((on b c) (clear b),4)
- ((on a c) (holding b),4)
- ((on c b) (clear c),4)
- ((on d b) (holding c),4)
- ((on a c) (on d b),4)
- ((on b c) (holding b),5)
- ((on a b) (on b c),5)
- ((on d b) (on b c),5)
- ((on c b) (holding c),5)
- ((on a c) (on c b),5)
- ((on c b) (on d c),5)

PATTER DATABASES

Advantages:

- Heuristic calculations are not constant time (it's a lookup function)
- Reusable when solving the same problem

Drawbacks:

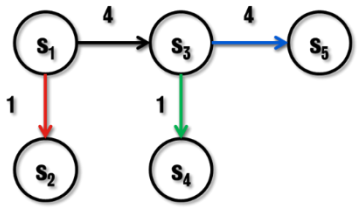
- Slow computation time to build the pattern database
- Reusability is quite limited (changing goal or increasing variables requires reconstruction of database)

OPTIMAL PLANNING – COST PARTITIONING

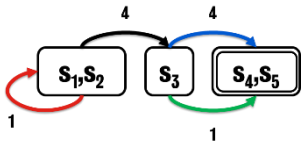
COMBINING HEURISTIC INFORMATION

If we want to achieve optimal planning, we need to ensure admissible heuristics are provided. But given planning is hard (NP-Hard in even the simplest cases), we can't expect any one heuristic to always work well in a given task. Also, no one heuristic can really grasp the complexity of larger domains. So we can **combine information** from different heuristics to solve this problem.

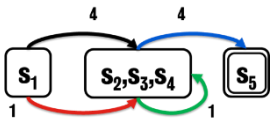
Example: consider this simple state space with five states with transitions and action costs identified, and each action is highlighted by a given colour. We then apply abstraction heuristics to that problem, and it generates two unique abstractions of the state space. In each case the topology and structure of that abstract space is slightly different. But it also results in two heuristics, which for now produce the same heuristic value for the initial state s_1 as we head towards goal state s_5.



Consider the abstraction given alongside (h_1). For this case, the following would be some of the heuristics: $h_1(s_1) = 5, h_1(s_2) = 5, h_1(s_3) = 1$



On the other hand, for h_2 (shown alongside) these values will be different:
$h_2(s_1) = 5, h_2(s_2) = 4, h_2(s_3) = 4$



We could combine the heuristics using h^{add} or h^{max} for example:

- $h^{max}(s_1) = \max(h_1(s_1), h_2(s_1)) = \max(5,5) = 5$
- $h^{add}(s_1) = h_1(s_1) + h_2(s_1) = 5 + 5 = 10$
- $h^{max}(s_3) = \max(h_1(s_3), h_2(s_3)) = \max(1,4) = 1$
- $h^{add}(s_3) = h_1(s_3) + h_2(s_3) = 1 + 4 = 5$

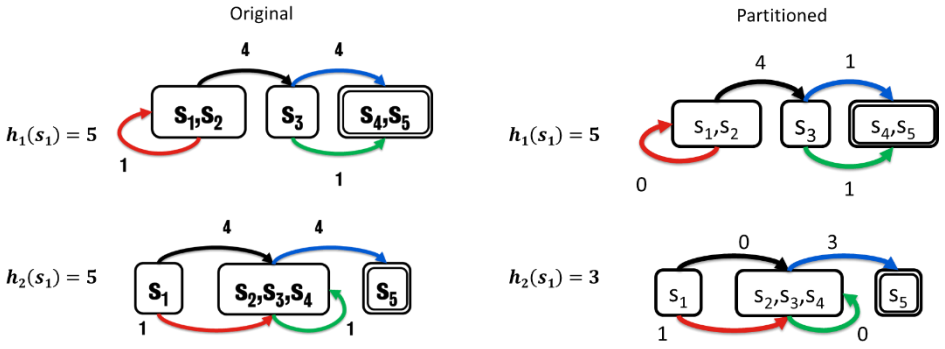
COST PARTITIONING

The aim is to split (partition)the action costs among the heuristics in such a way that the total cost implied by the heuristic does not exceed the original action cost, thus ensuring it remains admissible.

- Distribute the cost of each action across identical planning tasks.
- Combine arbitrary admissible heuristics into one single heuristic.
- However, the partitioning is designed such that sum of heuristics is now additive (and therefore admissible).

OPTIMAL COST PARTITIONING

Aim to find the best heuristic values for a given state among all the cost partitionings that we have available.

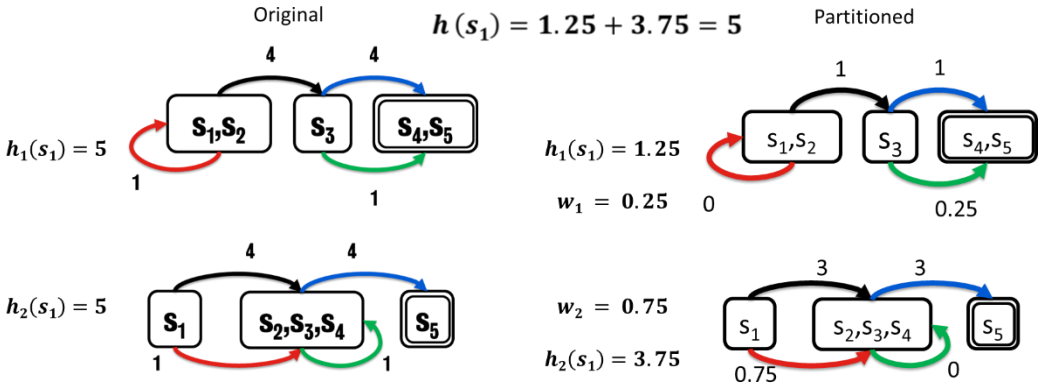


Consider the abstractions shown alongside. Now when we first look first at h_1, we can see the optimal cost is 5, given we take the black action which has a value of four, followed by the green action with a value of one. This results in 5, which is fine, but when we now consider h_2, we can't achieve the same goal using the same path. Instead we need to the red and then blue action to get 5, but the two actions that across both abstractions ensure a path to the goal are black and blue. Thus we can redistribute their costs such that we get an admissible heuristic each time that is also additive for both h_1 and h_2. In this case, we assign the full cost of the black action to h_1 and thus give it a value of 0 in h_2, while we need to redistribute the cost of four across both h_1 and h_2. We given it a value of 1 in h_1 and 3 in h_2. Resulting in h_1 being 5, h_2 being 3 and an additive admissible partitioned heuristic value of 8.

This requires us to analyze and recompute costs across all of the actions in each abstraction. It works and is relatively fast to compute, and by fast we mean it runs in polynomial time, but it still isn't all that feasible to run on larger problems. Given rolling this out in even small example state spaces is going to be quite time consuming to do.

POST HOC OPTIMISATION

Compute weight factors w to be applied to costs relevant to heuristic such that the sum of their calculations is **additive** and **admissible**. If we consider the example here, it's clear than in both cases the black and green actions are relevant given in both cases they get us to the goal. Meanwhile in h1 the red action is not relevant, given that keeps us in the same abstract state and the same applies to the green action in h2. It's clear than in h1, the black blue and green actions are relevant, while in h2, the black, blue and red actions are relevant.



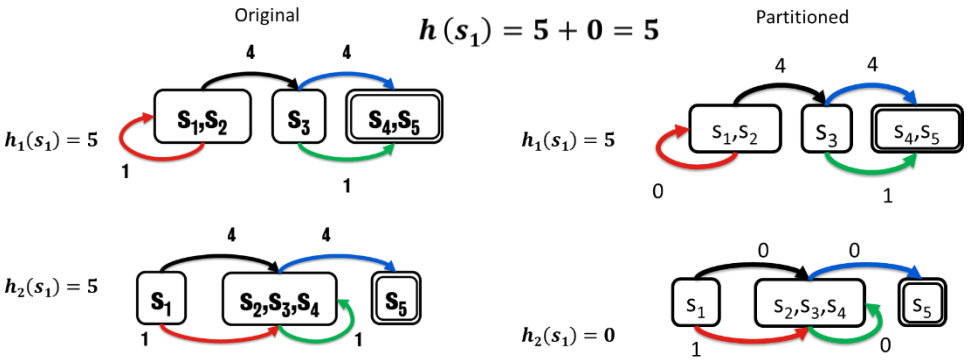
So next we calculate a weighting for each heuristic w1 and w2 – each of which being between 0 and 1 and sum up to a value of 1 – such that when we then multiply each cost of a relevant action by the weight and if it isn't a relevant action, we set it to 0.

So, if we return to h1 under post-hoc optimisation, we have calculated a weighting of 0.25 such that it now means the black and blue actions are now 1 each, and in h2 with a weighting of 0.75 they are 3 each. Meanwhile the red action in h1 is now 0 given it isn't relevant to the heuristic calculation and the same applies for the green action in h2. Now this is but one configuration, we could set weights w1 and w2 differently, but this gives us whole numbers for each action – which is easier to work with – and still retains the additive feature we so desire.

GREEDY ZERO-ONE COST PARTITIONING

Orders heuristics in a particular sequence then uses the full costs for the first relevant heuristic.

returning to our examples, the black blue and green actions are relevant actions in this context. Hence, we keep them as they originally were, with costs of 4, 4 and 1 respectively.



Clearly, as we've seen in the previous example on post-hoc Optimisation, the red action in h1 isn't relevant. So we scrub it and set it to zero.

Meanwhile, if we look at h2, the green action is again not relevant, so we set it to zero, but all the other actions are relevant. However, we have already used the black and blue action in this case, so we also set those a value of zero. Hence the only action that retains its cost value is the red one, given the green one isn't relevant. This actually means that the heuristic value is 0 for h2. Not terribly informative, but it still gives us an additive set of heuristics. But, it's also a little wasteful, given if we think about it, we stripped any useful information from h2 because already used the costs of relevant actions in h1.

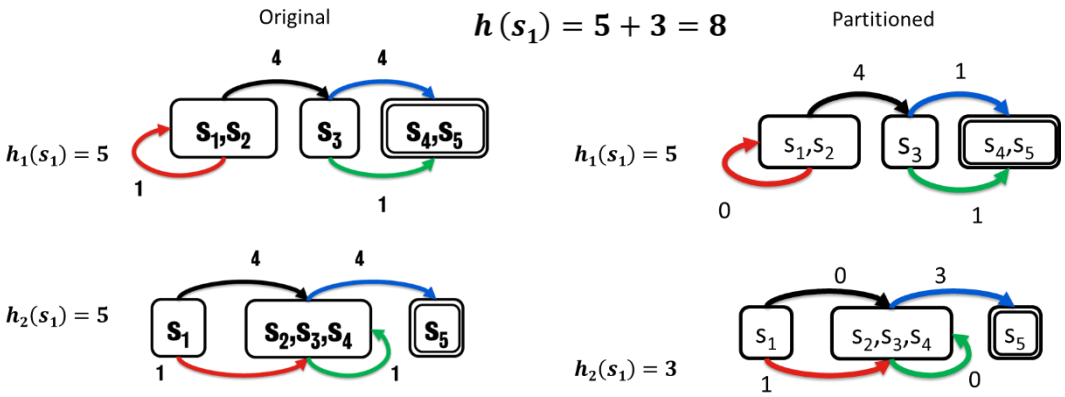
But what's interesting here, is that if we think about it, the cost of the blue action in h1 isn't relevant, given we don't use that to calculate the heuristic. Hence, we have a slightly improved version of this approach called saturated cost partitioning.

SATURATED COST PARTITIONING

Orders heuristics in a sequence, then

- Uses minimum costs to preserve estimate
- Uses remaining cost for other heuristics

The blue action is reduced to a cost of one in h1, bringing it in line with the green action and this means that for the blue action in h2, we can use that remaining cost value as part of the heuristic calculation.

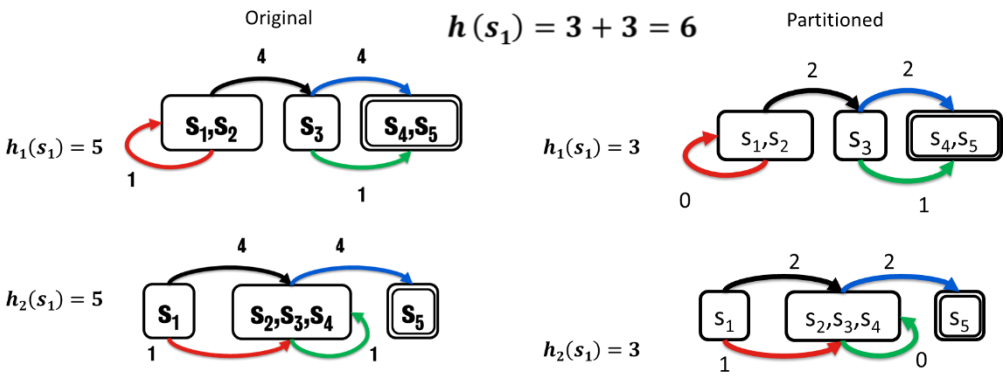


This now means that with saturated cost partitioning we have a cost of 8, which is not only again closer to the actual value and thus a little more informative but is also still admissible.

UNIFORM COST PARTITIONING

Distribute costs uniformly to all relevant actions (distribution across heuristics).

So, if we look at h1, then in this instance the red action is not relevant. But the black blue and green actions are relevant. Meanwhile, over in h2, the green action is not relevant, but the red, black and blue actions are relevant. If we consider the union of these two sets, then only black and blue actions are relevant in both. So, we will divide their total cost by the number of heuristics that use them. Meaning we then divide their cost by 2. So, this means that in both h1 and h2 the black and blue actions now only cost 2 in each case.



However, the cost of the red action in h1 is set to zero because it is not a relevant action, but the green action – given it is only relevant in h1, keeps its cost of 1.
And we can see the exact same thing with h2, only this time red is relevant and green is not. So, their costs are set accordingly.

GRAPHPLAN

GraphPlan generates a planning graph of fact and action layers, whereby we roll out all of the facts that are true on the current layer, then all of the actions we can achieve from the previous fact layer. Relaxed planning graph (RPG) heuristic is based on GraphPlan. One of the main differences between GraphPlan and RPG is that GraphPlan also captures the mutex relations that exist between facts in the action layer, such that it ensures actions that appear in the action layers can only occur at that time, provided the preconditions are not mutex.

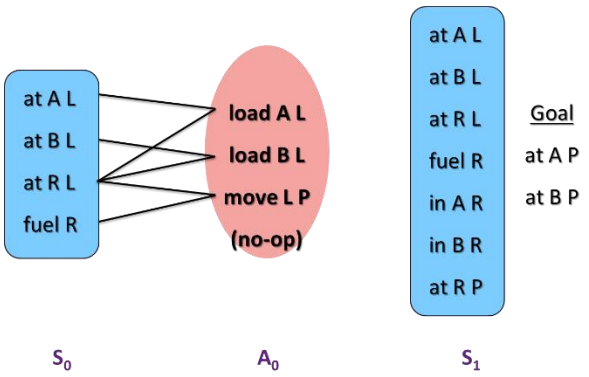
- The planning graph constrains search to a valid plan.
  - Finds shortest plans (i.e. shortest makespan).
  - Will search backwards from the end of the graph back to the initial state.
- Graphs can be built fairly quickly.
  - Graph construction is in polynomial time, but planning can be exponential time.
  - Will terminate with failure if there is no plan.

Example: Rocket Domain.

- Literals
  - $at\ X\ Y$: X is at location Y
  - $fuel\ R$: rocket R has fuel
  - $in\ X\ R$: X is in rocket R
- Actions:
  - $load\ X\ L$: Load X (onto R) at location L (X and R must be at L)
  - $unload\ X\ L$: Unload X (from R) at location L (R must be at L)
  - $move\ X\ Y$: move rocket R from X to Y (R must be at L and have fuel)

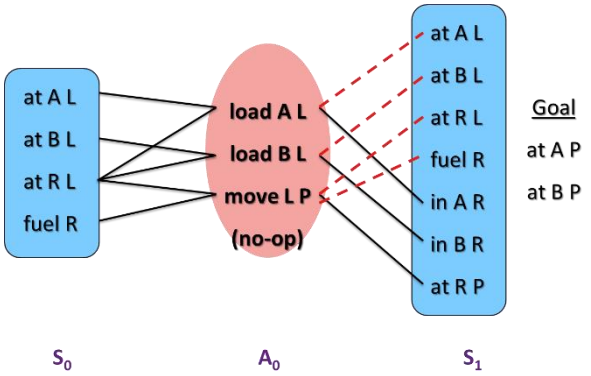
- Graph Representation
  - Solid black lines: preconditions/effects
  - Dotted red lines: negated preconditions/effects

In the initial state, we have packages A and B at location L. The rocket is at location L and we have fuel in the rocket. The goal is to have both packages at location P, which the rocket can fly to with a single action. We setup our initial proposition or fact layer S_0 and then find what actions are valid for us to execute in action layer A_0 – including the no-op, or doing nothing – plus we are tracking the preconditions required by each of these facts to enable us to execute each of these actions.

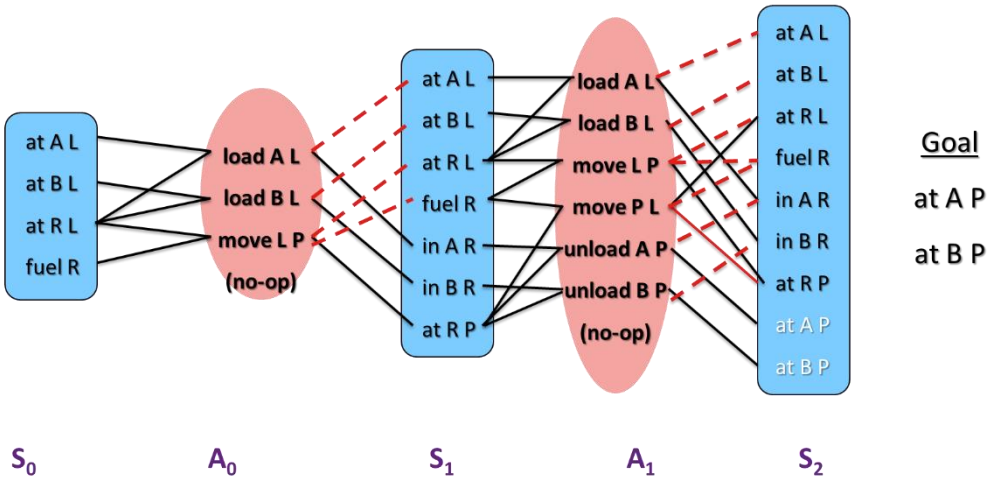


This then results in the subsequent fact layer S_1, which shows all the facts that could be true, as a result of the execution of those actions or just doing nothing and skipping a timestep.

But this time, let's also maintain a record of the effects of actions, what action created a fact or deleted it, which is going to be useful for us later during the planning process. So here, we can clearly see that loading the packages onto the rocket deletes the facts that A and B are at L, moving the rocket from L to P deletes the fact that it's at L and the fact that it has any fuel after the journey is completed.



So as we roll it out again, we're seeing how all of the preconditions and effects are being monitored throughout the graph and we see that the goal facts, of the packages A and B being at location P are now visible within the latter fact layer. So we have generated action layers based on available facts, then created new fact layers based on the new facts that can emerge, with the no-op maintaining the state as it currently is, plus in future actions layers we consider all of the possible actions that can now be executed in order to solve this problem and this continues on until those goal facts.



All of this is pretty much the same as the RPG heuristic, except we're paying attention to the delete effects of actions, whereas in RPG we simply ignored them thanks to the delete relaxation. What is important to recognise here, is that **this will not result in a valid plan yet**. This is because we have not yet considered the **mutex relations**. Mutexes across facts and actions help ensure that we find complete and sound plans.

VALID PLAN

In order for a valid plan to emerge in GraphPlan, we are reliant on a much stricter set of rules than we were when we use this for RPG:

- Actions at the same level don't interfere with one another
- Each action's preconditions are true at that point in the plan
- Goals are satisfied at the end of the graph
- Two actions or literals are mutex if no valid plan could contain both

Mutexes

There are 3 different types of mutex relations:

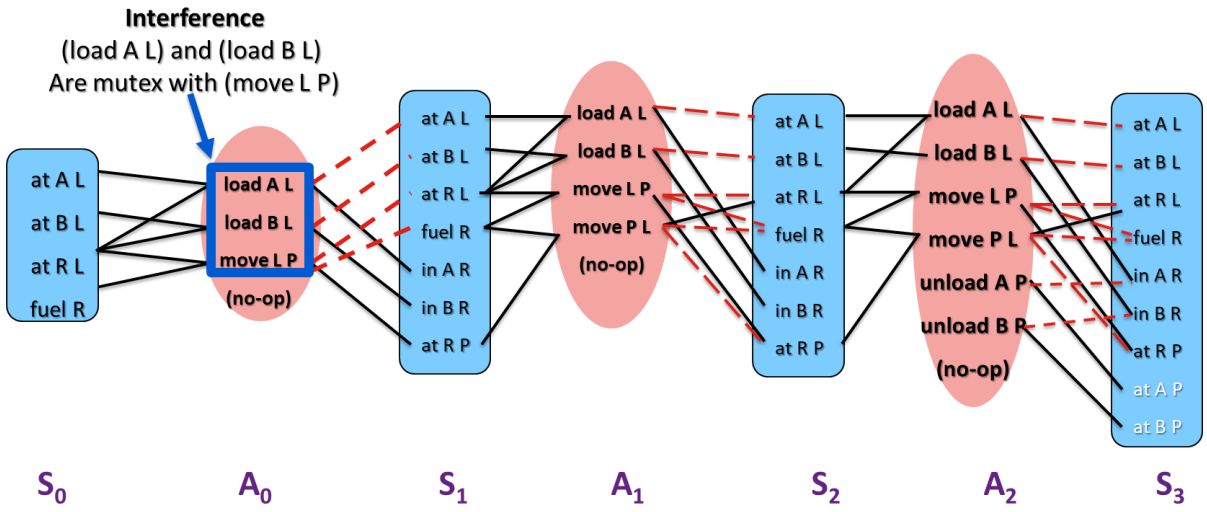
- **Interference**: Two actions that interfere with one another where the effect of one negates the precondition of another
- **Competing Needs**: Two actions where any of their preconditions are mutex with one another
- **Inconsistent Support**: Two literals that are mutex given all ways of creating both are mutex.

Thus, we can extend the planning graph using mutexes in the following way:

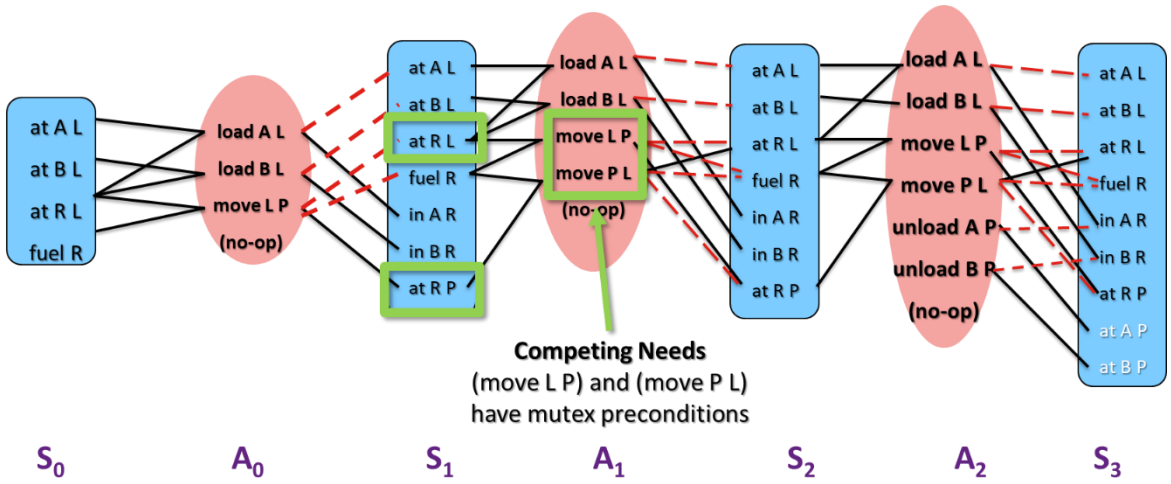
- **Action Layers**
  - Include all actions (including no-op) that have all their preconditions satisfied and are not mutex.
  - Mark as mutex all **action pairs** that are incompatible
- **Fact Layers**
  - Generate all propositions that are the effect of an action in preceding fact layer.
  - Mark as mutex all pairs of propositions that can only be generated by **mutex action pairs**.

FINDING MUTEXES

Consider the previous example with an addition fact layer S_3, from which we can potentially build a solution from. This is because now we can handle the mutexes that have emerged.



In A_0 we have **interference**; the 2 load actions both have a mutex relationship with the move action. While the two load actions themselves could be executed concurrently, they cant be executed on the same action layer as move, because th effect of the move action is mutex. The 2 load actions do not contradict each other.



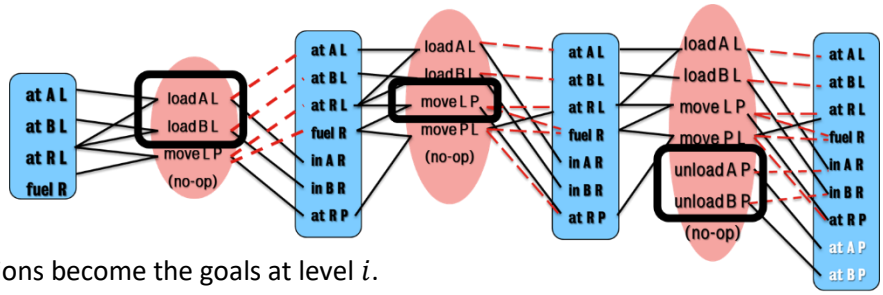
We can see **competing need mutexes** appear and a good example of this is the second action layer, given we now have a situation where both moving the rocket from L to P and vice versa are available on the same fact layer. This is of course mutex, but unlike the interference mutex (where the action of one impedes the effect of another), in this case the preconditions are mutex. Because the in the previous fact layer, the fact that the rocket is at L and P are mutex, hence these actions are also mutex in turn. These facts appear as mutex, because the action (move L P) has a mutex action pair with the no-op. This allows us to very quickly catch all the mutex facts that appear in the second fact layer.

BASIC GRAPHPLAN ALGORITHM

- Grow the planning graph (PG) until all goals are reachable and none are pairwise mutex
  - If PG levels off (reaches a steady state) first, fail the plan
- Search backwards from the max fact layer for a valid plan
- If no plan is found, add a level to the PG and try again

PLAN GENERATION

- Backward chain on the planning graph.
  - Complete all goals at one level before going back
- At level i, pick a non-mutex subset of actions that achieve the goals at level $i+1$.
  - The preconditions of these actions become the goals at level i.
- Build the action subset by iterating over goals, choosing an action that has the goal as an effect.
  - Use an action that was already selected if possible. Do forward checking on remaining goals.



PLANNING AS SAT

BOOLEAN SATISFIABILITY PROBLEMS (SAT)

SAT problems are a form of deductive reasoning whereby we attempt to encode a problem into a more succinct expression. This means we have a series of variables with operators applied to them. The aim is to find a combination of values for the expression that results in true. In the event the function returns true, we consider the problem to be satisfiable, otherwise it is considered unsatisfiable.

$F = A \wedge \bar{B}$
Satisfiable

$F = A \wedge \bar{A}$
Unsatisfiable

WHY USE SAT FOR PLANNING?

SAT and planning have significant overlaps. In both cases you're trying to find assignments of variables that yield a positive/true outcome over a given function. In the context of planning, we're trying to find a valid configuration of variables at different stages of the planning process such that it yields a true outcome, meaning that there is a combination of states, when put in the right order satisfy the constraints of the actions in the planning domain and enable us to transition from the initial state of the problem to the goal.

- SAT solving is a broad and rich research domain in and of itself.
- Typically used in software verification, theorem proving, model checking, pattern generation etc.
- There are existing SAT solvers established within this community that can be adopted.

PLANNING AS A SAT PROBLEM

- **Planning Problem**: $\boldsymbol{\rho = (\Sigma, s_i, s_g)}$, where
  - $\Sigma \rightarrow$ state transition system
  - $s_i \rightarrow$ initial state
  - $s_g \rightarrow$ goal state
- **Idea**: Propositional variables for each state variable
  - Create unit clauses that specify the initial state and the goal state
  - Create clauses to describe how variables can change between two time points t and $t+1$
- **Solution**: Create a SAT formula ϕ such that
  - If ϕ is satisfiable, then a plan exists for the planning problem ρ
  - Every possible solution for ϕ results in a different valid plan for ρ

SAT ENCODING: VARIABLES

- **State Variables**
  - A set of propositional variables $\boldsymbol{v_i}$ that **encode the state** after $\boldsymbol{i}$ steps of the plan.
  - There is a finite number of steps of the plan, encoding as $\boldsymbol{T}$: the **planning horizon**

$$v_i \; \forall \; v \; \in V, \; 0 \; \leq i \; \leq T$$

- **Action Variables**
  - A set of propositional variables $\boldsymbol{a_i}$ that **encode the actions applied** at the $\boldsymbol{i^{th}}$ step of the plan.

$$a_i \; \forall \; a \; \in A, \; 1 \; \leq i \; \leq T$$

STATE VARIABLES

- Our planning formulation $\boldsymbol{\rho \; = \; (\Sigma, \; si, \; sg)}$
- **Initial State**
  - Unit clauses encoding the initial state.

$$v0 \; \forall \; v \; \in si \;\; \text{and} \;\; \neg v0 \; \forall \; v \notin si$$

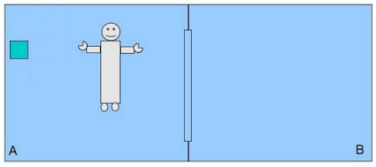
- **Goal State**
  - Unit clauses encoding the goal state.

$$vT \; \forall \; v \; \in sg$$

Example: Gripper domain. There are 2 locations, one box to move from A to B.

$$\{at(r1, locA), at(b1, locA), free(left), free(right)\}$$

For the sat encoding at $T = 1$, we encode the initial state as all the facts that are true at timestep 0, and we denote the timestep in each of these statements.



Initial State:

$$at(r1, locA, 0) \wedge at(b1, locA, 0) \wedge free(left, 0) \wedge free(right, 0) \wedge \neg at(r1, locB, 0) \wedge \neg at(b1, locB, 0) \wedge \neg holding(left, b1, 0) \wedge \neg holding(right, b1, 0)$$

Goal State:

$$at(b1, locB, 1)$$

SAT ENCODING: ACTIONS

For all the actions that exist in the domain, we need to encode the preconditions and the add and delete effects of the actions. But we need to do it for each time step i up to the planning horizon T. Thus, in the action formalisation we need to encode the preconditions, add and delete effects of each action:

$$ai \Longrightarrow \{\wedge f \in precond(a) fi\} \wedge \{\wedge f \in effectAdd(a) fi+1\} \wedge \{\wedge f \in effectDel(a) \neg fi+1\}$$

Thus, the formalisation consists of all the facts that exist at a given timestep i, which is at most one less than the horizon.

- We list all of the facts that are true at timestep i, followed by what facts are true in time step $i+1$, thanks to the add effects
- As well as the facts that are now no longer true as a result of the delete effects of the action.

Example: The same gripper domain.

- **Pickup Box b** using Gripper g on Robot r in Location x.
  - pre: $at(b, x),\ at(r, x),\ free(g)$
  - eff: $\neg at(b, x), \neg free(g),\ holding(g, b)$
  - SAT Encoding at $T = 0$

$$pickup(b1, left, locA, 0) \Longrightarrow at(b1, locA, 0) \wedge at(r1, locA)\ free(left, 0) \wedge holding(b1, left, 1) \wedge \neg at(b1, locA, 1) \wedge \neg free(left, 1)$$

- **Move Robot r** from Location x to Location y.
  - pre: $at(r, x)$
  - eff: $at(r, y),\ \neg at(r, x)$
  - SAT Encoding at $T = 0$

$$move(r1, locA, locB, 0) \Longrightarrow at(r1, locA, 0) \wedge at(r1, locB, 1) \wedge \neg at(r1, locA, 1)$$

THE FRAMING PROBLEM

The Framing Problem – we need a way to capture information that does not change between time steps i and $i+1$.
The following 2 approaches can solve this:

- **Classical Frame Axioms**:
  - State which facts are not affected for each action
  - List every fact that isn't affected by an action at a given timestep, i.e., fact should persist into the next time step
  - This relies on one action being executed per time-steps so axioms can be applied.
- **Explanatory Frame Axioms**:
  - Instead of listing what facts are not changed, explain why a fact might have changed between two time steps, i.e., if a fact changes between i and $i+1$, then an action at step i must have caused it
  - Identify what action would have caused that information to change as we moved between time steps

EXPLANATORY FRAME AXIOMS

Focus on how fact changes between two time steps. In this case if we work at the actions from the example, we identify that in order for the fact that $r1$ is not at location B on the time step 0, but then this is in the location at time step 1, then that must mean that the robot was part of a move action at time step 0. Specifically, the move action with $r1$, going from $loc\ A$ to $loc\ B$.

Examples:

$$\neg at(r1,\ locB,\ 0) \wedge at(r1,\ locB,\ 1) \Longrightarrow\ move(r1,\ locA,\ locB,\ 0)$$
$$\neg holding(left,\ b1{,}0) \wedge holding(left,\ b1,\ 1) \Longrightarrow pickup(b1,\ left,\ r1,\ locA,\ 0)$$

This approach enables for parallelism in the planning process, given two actions could be executed in parallel if they have the same preconditions at time step t and their effects don't conflict. You can more easily identify this by catching whether or not an effect of one action appears within one of the explanatory frame axioms for an action that isn't the action you're running against.

EXCLUSION AXIOMS

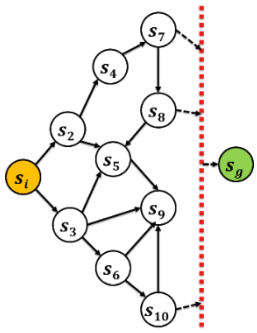
Returning to the notion of mutual exclusion, we identify what actions cannot occur at the same time.

- **Complete Exclusion Axiom**: only one action at a time.
$$\neg move(r1,\ locA,\ locB,\ 0)\ \vee\ \neg move(r1,\ locB,\ locA,\ 0)$$
- **Conflict Exclusion Axiom**: Prevents invalid actions on the same timestep.

SAT PLANNING – SOLVING THE PROBLEM

The collection of axioms is converted into conjunctive normal form and then fed to a SAT solver. SAT solver will iterate out (increasing T) until it can find an assignment of truth values that satisfy ϕ. Once reached – provided planning is total order – there will be exactly one action a such that $a^i = true$.

PARTIAL ORDER PLANNING (POP)

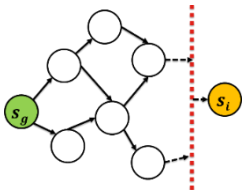


FORWARD SEARCH (PROGRESSION SEARCH)

- Searching forward from the initial to goal state can be problematic due to the nature of the problem space
- We may wind up creating redundant steps in our plans
  - This could be the fault of the heuristic
  - Or is a reflection of corner of the search space we fall into
- We may find ourself in a situation where we're caught in a loop
  - Heuristic search should avoid this, but uninformed search is vulnerable
- Depending on the algorithm, we can be at the mercy of the state space to actually find the solution

REGRESSION SEARCH

Searching backwards from the goal state. Instead of aiming of searching for a state that holds all goals, we search for assignments of the goal facts.



- Find actions with add effects that create goal facts.
- Then add (unsatisfied) preconditions as subgoals to achieve a plan.

STRIPS (The Stanford Research Institute Problem Solver), is arguably the first planning system dating back to 1971. It used **regression search** from the goal state, but it was **incomplete**, since if it couldn't satisfy one given precondition based on a valid action it has explored, it bottoms out and assumes the planning process is impossible.

REGRESSION PLANNING

- **The set of actions $A(s)$ applicable in s are the operators $op \in O$ that are relevant and consistent.**
  - Namely, for which $Add(op) \cap s != \emptyset$ and $Del(op) \cap s = \emptyset$
  - Operators that add something in the state, and don't delete anything in the state.
- **The state $s = f(a, s)$ that follows the application of $a \in A(s)$ is such that:**

$$s = s - Add(a) + Pre(a)$$

This means we consider all possible actions and assess whether they result in new facts being added to the state, while also not removing information from the state. These are considered to be the appropriate actions we can consider and considering we're starting from the goal; this can be problematic given it could mean we avoid actions that are temporarily destroying goal facts only to replace them later. But ultimately, the state transition is then achieved by subtracting the add effects of the valid action, but also adding the preconditions, such that we then attempt to solve them next.

WHY SEARCH BACKWARDS?

- State spaces when searching forwards vs backwards are not symmetric.

Forward Search	Backward Search
One initial state	A set of goal states
Apply action a to state s? One value unique successor s'	Multiple states where a can be applied to reach s'

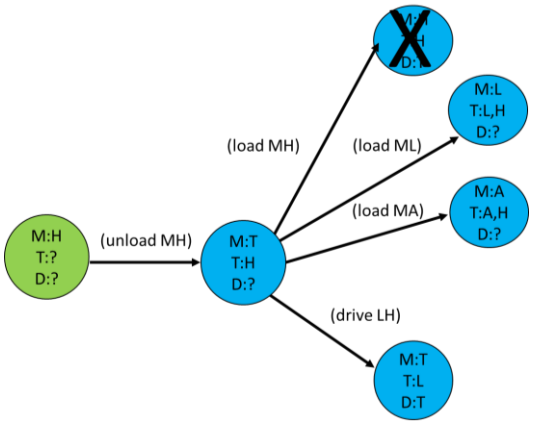
REGRESSION SEARCH – EXAMPLE

Consider the example given alongside; in the situation, the DVD needs to be delivered to the home by amazon. In the initial state, we had the DVD at amazon, so someone needed to get in a truck, load the DVD onto the truck and then drive it to the house. The location of the truck and driver are not relevant here since they are not critical goal facts. Now the only action that will create the desired goal fact is to unload M from T at location H. Given that this now ensures that the DVD is at my home, at H, it requires that we solve the precondition, which is that the truck is also at my H in order for the unload to happen.

(at mydvd amazon)
(at truck amazon)
(at driver home)
(path home amazon)
(link amazon london)
(link london myhouse)

We could have applied the following actions for this to happen

- Loading MyDVD onto the Truck at H (home)
- Loading MyDVD onto the Truck at L (London)
- Loading MyDVD onto the Truck at A (Amazon)
- Driving the Truck from L to H (London to home)



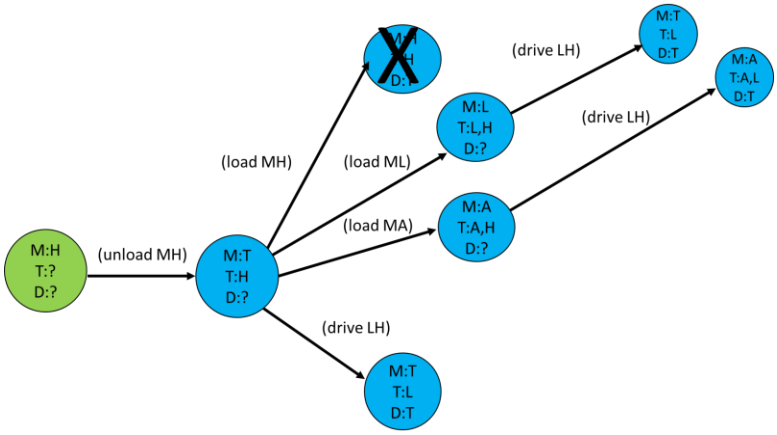
Now as mentioned, it's important to factor in whether or not an action is consistent. In this case, the first action, loading MyDVD onto the truck, isn't a good action here. Why? Because yes it will satisfy the precondition to have MyDVD on the truck, but it also deletes a goal fact of MyDVD being at H. So, it's not applicable and can we ignore it.

Driving from L to H makes a lot of sense here. But you'll notice that also the other two load actions for loading MyDVD onto the truck. Now we're only interested in whether the add effects satisfy our current subgoal, which is to have

MyDVD on the Truck. These are valid paths of the state space to explore, even though it leads to the possibility in that the truck is in one of two different places in each of these scenarios.

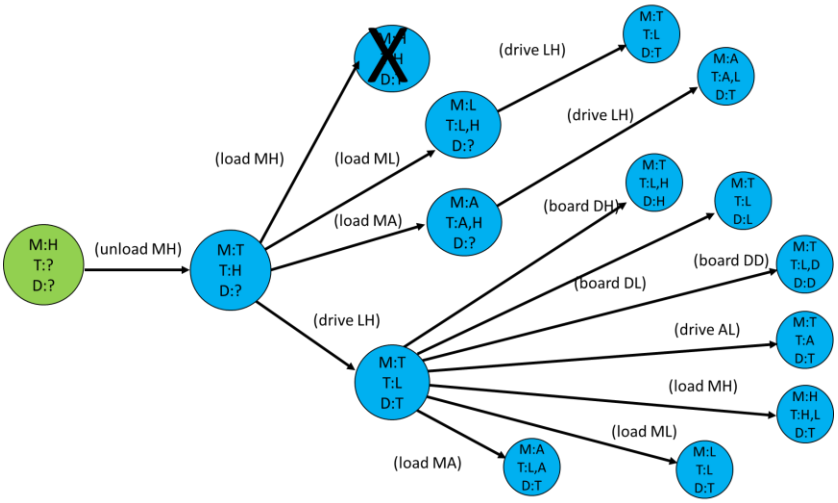
Looking at the top two states, there is again a need to satisfy two preconditions:

- In the first instance, it's that the driver is in the truck and we know that would happen if the driver has driven the truck somewhere, as we already saw with the first drive action from L to H. Now this gives us a more concrete state, but it doesn't address the fact that somehow the drive between L and H happens before loading M onto T at L?
- Meanwhile the second rollout sees us also add a drive action. Which means we can now confirm the driver is in T. But we still don't really know where the Truck is at this point. It could be either at A or at L given we've driven away from H to L.



Looking at the 3rd unexpanded state, its much of the same process, with varying valid and consistent actions that allow us to enumerate possible situations from the goal state.

But the key thing, is we're getting closer to a valid initial state, given that Drive AL driving from Amazon to London is a state where M is in T, T is at A and D is driving T. So, we only must get M loaded onto T, for D to board T at A which in turn requires D to walk from their home to Amazon. At which point the plan is satisfied. It's a different way in which to do it, but we can find a good solution by rolling out in this direction.



But it is possible that we could have found other valid actions throughout this search graph. In fact, we found actions such as the driver boarding the truck and the package being loaded onto the truck that are useful, but just not at that point in time. Hence, it would useful if we could establish that certain actions are good to have in the plan, but just don't commit to them entirely.

PARTIAL ORDER PLANNING: SEARCHING IN PLAN SPACE

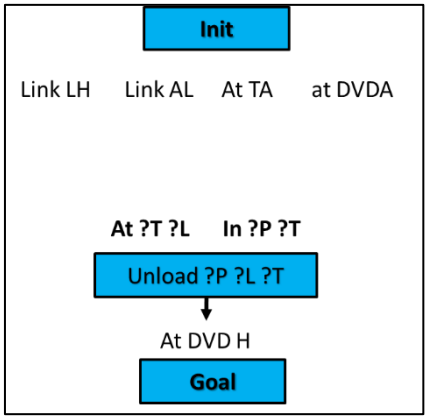
The aim of POP is to plan in a lifted presentation of the state model and find actions that can be applied but not dictate the order immediately; generate a partial order plan and then try and find the optimal solution that ensures the orderings are valid. Thus, either

- Pick an **unresolved goal**
  - Select an **action** to achieve it (or the initial state)
  - Add to the plan

- Add a **causal link** (identifier that acknowledges which actions meet which preconditions of other actions; helps identify the ordering of actions in the final plan) and add its preconditions as open goals

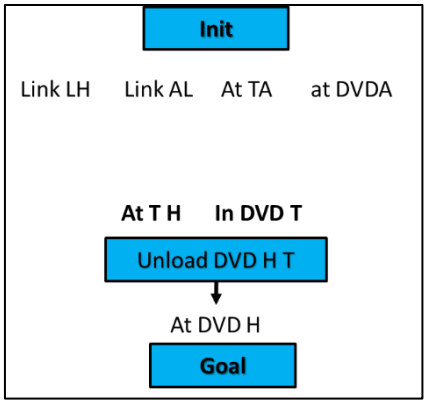
- Find an **unbound parameter** and select a binding
- Find a **threat** (when the search process discovers an existing ordering of actions that will break bindings and causal links), resolve by **promotion** or **demotion**

Example: Consider an example given alongside, where we start with the initial and goal state and then we consider how to achieve the goal fact of having the DVD at H.



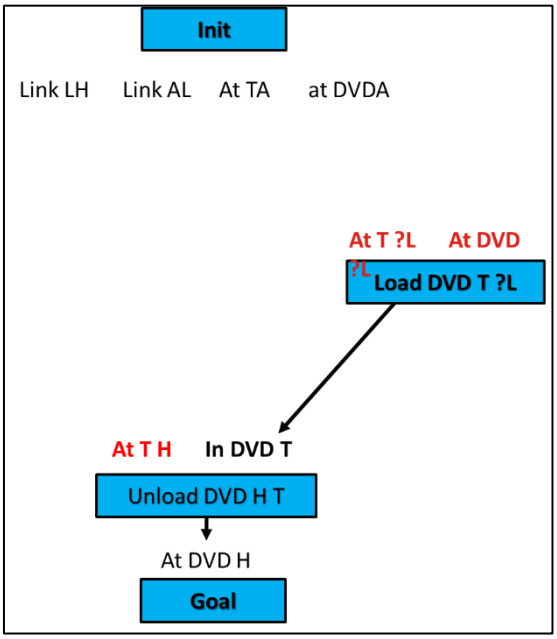
We could consider using the unload action given that would achieve the desired effect, we then need to consider the preconditions involved and what assignments to all of these variables are required.

We can quickly fill in the possible values for this, with truck T being used and the DVD being at location H. We can try a variety of combinations, but the one shown alongside is the one that works.



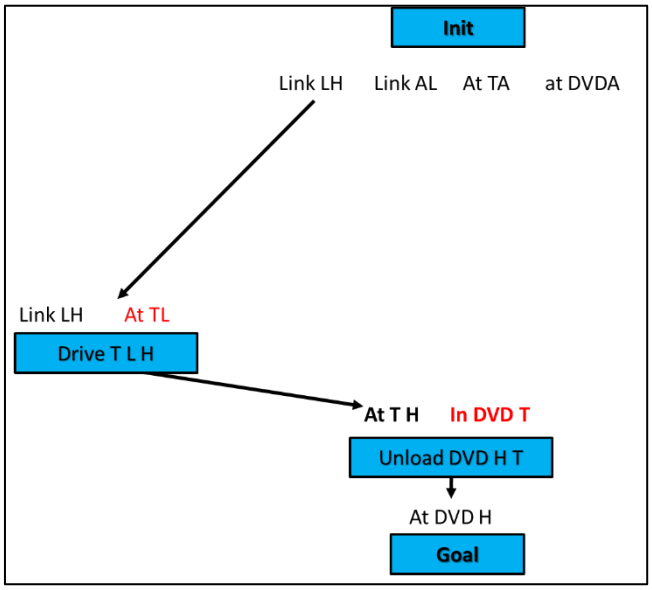
This not only sets up the preconditions, but also sets the new goal to solve, i.e., getting the truck to H and the DVD in the truck.

So next we consider how to satisfy loading the DVD onto the truck at L, but that doesn't work. Because the facts, given there is an inconsistency within some of the facts of where T is. If we assigned the variable L to be H, then it violates the goal we've already solved. And doesn't give us useful information of how the truck got to location H in the first place. So, let's scrap that for now.



We can establish that driving from L to H will get us to the correct destination and also, it generates a causal link in saying that Driving from T to H is required before we unload the DVD. Plus, there's also a causal link dependency on the drive action, because in order for it to work, we know that Link LH needs to be a valid fact.

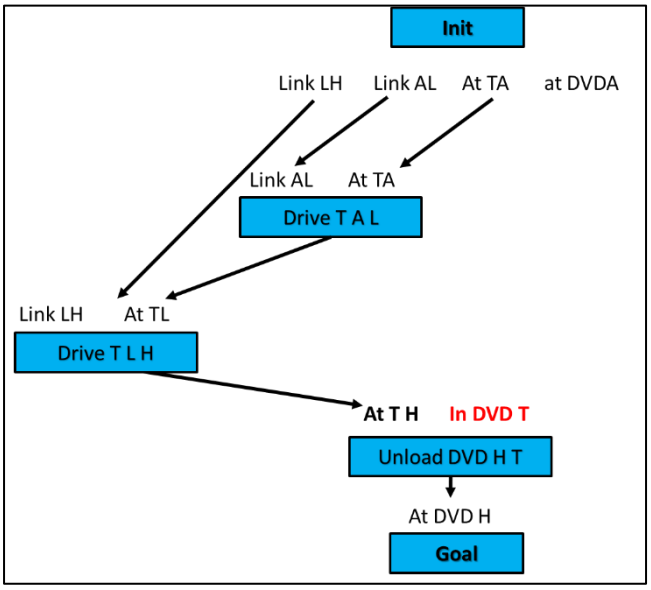
So, we know at some point before executing this action, we have to be at L in order to reach H. Now we still have two facts we need to resolve, how did the DVD get in the truck in the first place, and how did the truck get to location L before it drove to H.



So now, having tackled how the truck reached L, we've achieved a lot of useful information: we now know that Driving T from A to L can satisfy the need for T to be at L before it drives on to H.

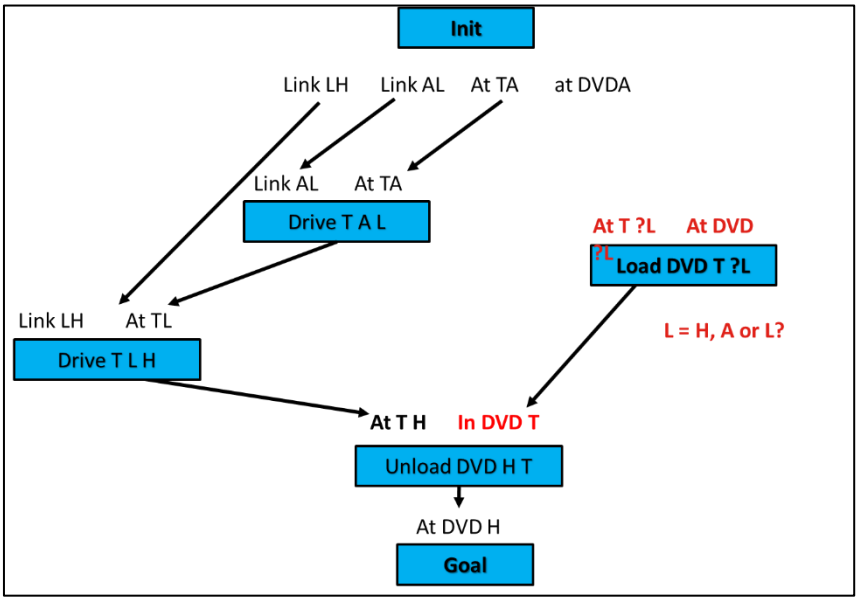
But also, we create three new causal links. We establish that Drive T from A to L is required before we then drive it from L to H. But also, we link the preconditions of Driving from A to L to the initial state. So, we know that driving T from A to L not only has to happen before we drive to H, but it also needs to happen after the initial state.

This is coming along nicely, but we still don't know about the DVD being in the truck. Perhaps we can revisit that action from before?

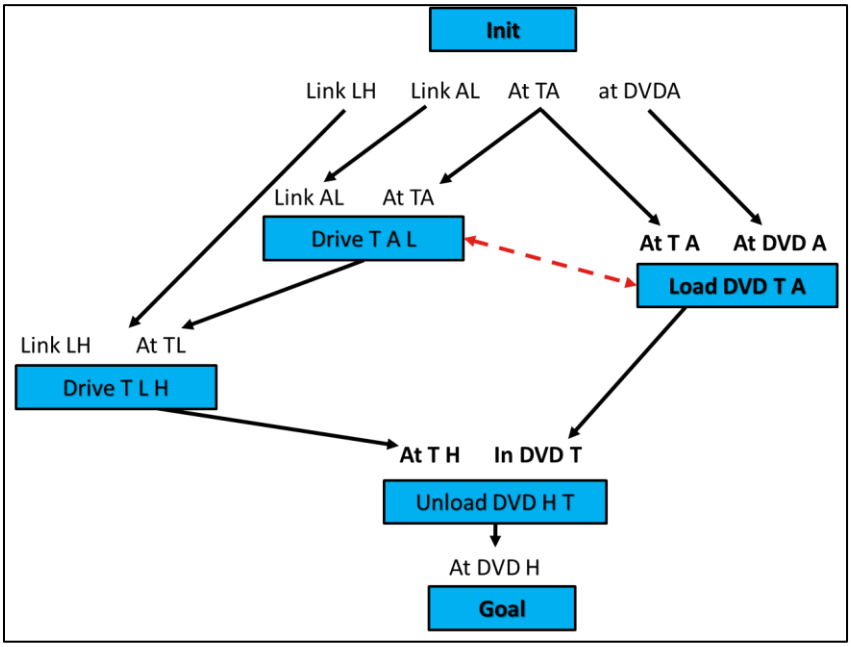


So, if we reconsider the load action, we now have a range of values that the location can be set to inside the load action. It could A, or L or H.

So, let's find the best one that fits in. We know going through this example the answer is A, but what happens when we assign it to A?



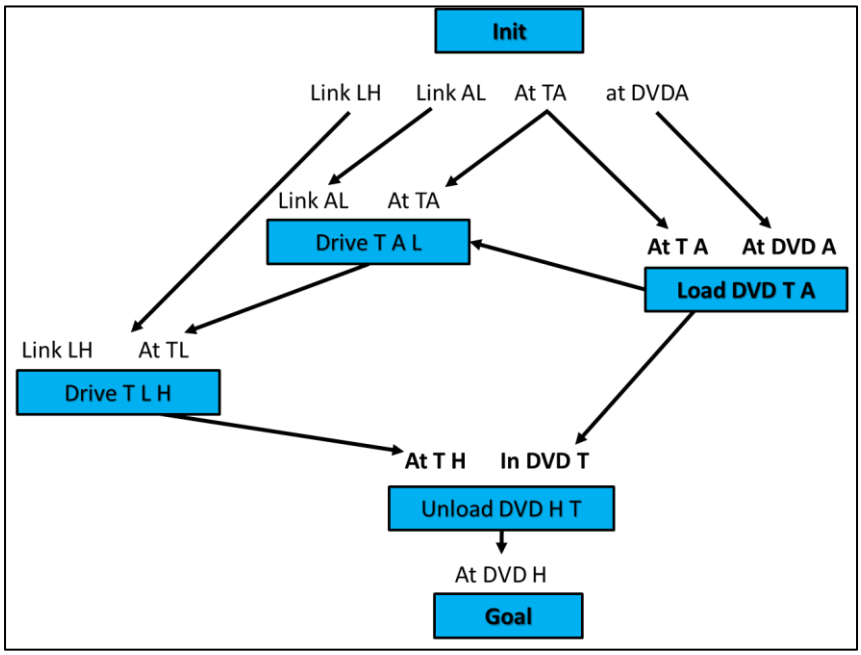
By assigning the value of the location in the load action to A, we now see some nice causal links established between the location of both T and the DVD in the initial state and how that will ultimately satisfy the rest of the plan.



We've actually satisfied not just the goal fact, but also all of the preconditions of the facts relied on in order to give us the final solution. But there is still the issue of what the final order is.

And in fact, we now have a threat that's emerged in the partial plan. Both the Load and Drive actions appear to be applicable off the back of the initial state. But we can see that the Load action requires the truck to be at A, as does the drive action, but we know that the drive action will result in the truck being at L.

We go ahead and dictate the ordering such that the load action has to go before it, given that the Drive action threatened the causal link between At TA in the initial state and the load action, we promote the Drive action forward to past the conflict. This now ensures – based on the orderings – we will load the DVD onto the truck before we are driving away from Amazon.



As a result, we now have a successful plan ordering that will ensure a complete transition of actions from initial state to goal and can call it quits having successfully solved the problem.

SUMMARY

Even though forward search is the most popular, it is not the only process to find solutions. Regression search can help reduce state spaces to some degree, but causal relationships are not properly encoded and makes it difficult to find good plans.

Meanwhile POP planning can help us find nice totally ordered plans, but it has its own problems, given we now have issues with threats and bindings that increase the branching factor and increase the complexity of the problem space.

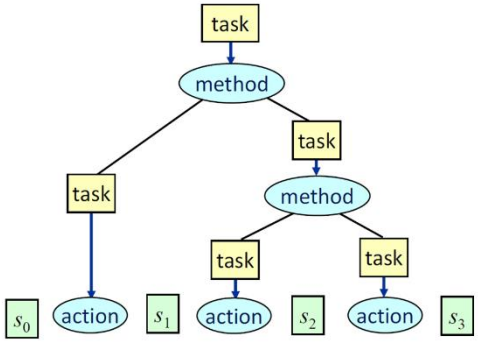
HIERARCHICAL TASK NETWORK (HTN) PLANNING

The aim is to reformulate a typical planning problem into a different format; instead of goals that we seek to achieve, we have tasks we wish to complete. Thus, in HTN planning we have

- **Tasks** to complete instead of **goals** to satisfy
  - Tasks are a little more abstract than a typical planning problem.
- **Methods** to decompose tasks into **subtasks**:
  - Enables for gradual decompositions of complex tasks into smaller and more manageable tasks.
  - A subtask could be a single action, or another method in and of itself.
- We enforce **constraints** of the problem in the **methods and actions**

FORMALISATION OF HTN PLANNING PROBLEM: (Σ, M, A, s_0, T)

- $\Sigma \rightarrow$ a state variable planning domain, this includes the following
  - $S \rightarrow$ set of all state spaces
  - $A \rightarrow$ set of all actions
  - $\gamma \rightarrow$ state transition function, which can be applied to a state s to produce a new state s'
- $M \rightarrow$ a set of methods that are designed to decompose tasks.
- $s_0 \rightarrow$ the initial state of the problem
- $T \rightarrow$ a list of tasks to complete



$$\Sigma = (S, A, \gamma), \qquad s \in S, a \in A, \qquad \gamma(s, a) \rightarrow s'$$

HTN Planning defines **2 TYPES OF TASKS**:

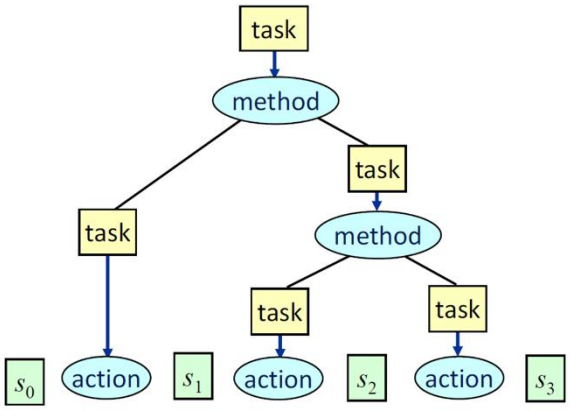
- **Primitive Tasks**: translate to single planning action
- **Compound Tasks**: translate into a collection of one or more tasks
  - Tasks which require a method to be applied to. Once the methods preconditions are satisfied, these can be decomposed into more tasks.
  - Compound tasks can be decomposed into 1 or more tasks (not 2 or more tasks), as they can also be used to apply additional constraints on an action that the original action's preconditions don't provide.

PLANNING PROCESS

- Recursively refine each task into subtasks.
- Ensure tasks can then be grounded with literals

As can be seen from the diagram given alongside, we start with the original task we were given to complete and given this is a compound task the method is applied to find a valid permutation of possible tasks based on the current state.

This is then decomposed into two tasks and we approach them from left to right. We visit the first task and as we decompose this into a primitive task, which translates into an action, which then results in the plan moving from s_0 to s_1. Meanwhile, we can't find an action that matches the task on the right, so we once again need to run a method to find a valid decomposition of this task, which in this case



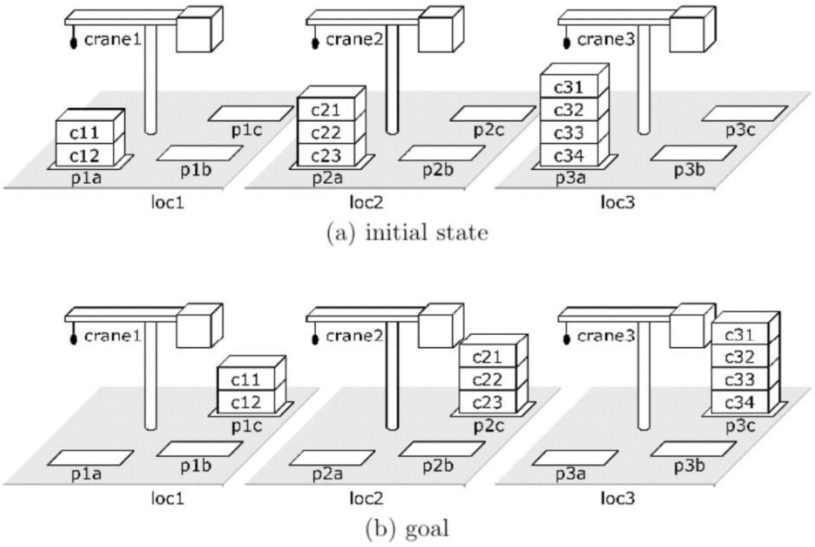
is two more primitive tasks, enabling for us to move from s_1 to s_2 and then finally s_3

What this allows is for a top-down approach to behavioural design. If you think about how plans are usually constructed, it's more of a bottom-up approach, whereby the individual actions are selected and the resulting plan emerges from the planner finding out that putting actions together in a specific sequence yields an intelligent response. We then have to go back and correct this as we realise the search is taking us down dead-ends, but ultimately, it's more focussed on individual actions being glued together in the correct sequence.

However, in this case, what we're seeing is that the resulting plan is at the mercy of the task networks. By embedding specific constraints on how a task is executed, we're expressing a total order over a subset of actions that will yield a resulting behaviour that is exactly what the designer intended.

HTN EXAMPLE

Let's consider the dockworker or DWR example, which is a common example to see for HTN planning in textbooks and such is the dockworker or DWR domain. Now this domain is largely built around robots driving around and leaving containers on pallets that in locations, with the cranes responsible for picking up and moving objects around on the pallets at that location. A crane only has access to the pallets that are connected to it, so for example as we can see in this diagram, crane 1 can only move containers c11 and c12 and they can only be moved to locations p1a p1b and p1c. Because those are the three pallets attached to that location.



However, the task we have here is to move these containers from one pallet to another at each location, but to do so in such a way that preserves the original order. Hence as you can see in our initial state, in location 1 we have c11 atop c12 and they're both sitting on p1a. However, the goal is not only for them to be on p1c, but also that they retain that order of c11 being on top and c12 being on the bottom.

Thus,

- **Task**: Moving containers from one pallet in the yard (location) to another.
- **Goal**: Moving containers from original pallet X to destination Y, while preserving the order of the containers on the new pallet.

This is sort of an extreme example of problems such as the Towers of Hanoi and even blocksworld in that we have a limited number of locations to move objects around but have a strict ordering on their final destination. When we consider this as task, it's straight forward, we want to move one stack from its current location p, to a destination location q.

- **Task**: $move - stack(p, q)$
- **Parameters**: p (original location), q (destination location)

There are only 2 **ACTIONS** that we can execute; taking a container off of a pallet and putting it on another one. Both actions require the crane (k) and the pallets that the container is either being lifted from or put onto. Since both the crane and the pallets are related to the same location, belong and attached predicates are used in the following actions:

- **Take** $< \boldsymbol{k}, \boldsymbol{l}, \boldsymbol{c}, \boldsymbol{p}, \boldsymbol{o} >$
  - Preconditions: $belong(k,l)$, $attached(p, l)$, $empty(k)$, $in(c,p)$, $top(c,p)$, $on(c,o)$
  - Effects: $holding(k, c)$, $top(o, p)$, $not(in(c,p))$, $not(top(c,p))$, $not(on(c,o)$, $not(empty(k))$
- **Put** $< \boldsymbol{k}, \boldsymbol{l}, \boldsymbol{c}, \boldsymbol{p}, \boldsymbol{o} >$
  - Preconditions: $belong(k,l)$, $attached(p, l)$, $holding(k,c)$, $top(o,p)$
  - Effects: $in(c,p)$, $top(c,p)$, $on(c,o)$, $not(top(o,p))$, $not(holding(k,c))$, $empty(k)$

We can start the HTN planning process by defining some **METHODS** (shown alongside). The Recursive-Move task is recursive because it is asking us to solve the original task after it does everything else it needs to do. But for a recursive task like this to work, it also needs a base case (a situation where we can ensure that the recursion stops) To address this, there is the Do-Nothing task. This task checks whether only the pallet is at the top of a given pile, which means that the pile is empty. If the pile is empty, we can apply this method which will effectively stop the recursive-move task from continuing to run, given we've completed moving every object. So now we consider how the move-topmost-container task could be implemented. We have another method entitled take-and-put, which is rather self-explanatory. In this instance, we are aiming to take the topmost item on a given pile and then move it to another one. The actual tasks are as expected, to execute both the take and put actions in that order. But what is interesting is that the preconditions of the method are essentially reinforcing that the grounding of variables for each task lines up in such a way that it is possible for these two actions to happen back to back. Hence, we ensure that the two locations both use the same crane but also that we have the correct grounding of variables for each of these actions such that we don't have to wait until the preconditions fail before realising that this isn't valid.

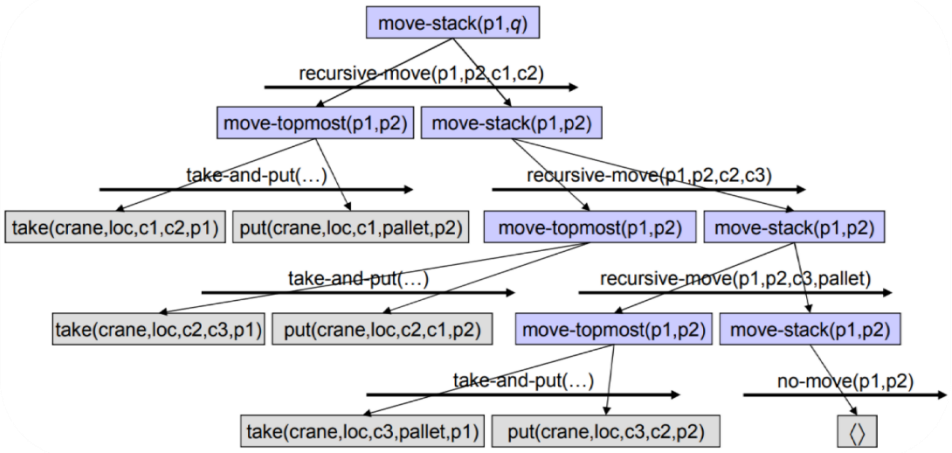
Recursive-Move $(\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{c}, \boldsymbol{x})$
Task: $move\text{–}stack(p,q)$
Preconditions: $top(c,p)$, $on(c,x)$
Subtasks:
$move\text{–}topmost\text{–}container(p,q)$
$move\text{–}stack(p,q)$

Do-Nothing $(\boldsymbol{p}, \boldsymbol{q})$
Task: $move\text{–}stack(p,q)$
Preconditions: $top(pallet,p)$
Subtasks (none).

Take-and-Put $(\boldsymbol{c}, \boldsymbol{k}, \boldsymbol{l1}, \boldsymbol{l2}, \boldsymbol{p1}, \boldsymbol{p2}, \boldsymbol{x1}, \boldsymbol{x2})$
Task: $move\text{–}topmost\text{–}container(p1,p2)$
Preconditions: $top(c,p1)$, $on(c,x)$, $attached(p1,l1)$, $belong(k,l1)$, $attached(p2,l2)$, $top(x2,p2)$
Subtasks:
$take(k,l,c,x1,p1)$
$put(k,l2,c,x2,p2)$

If we consider how this will roll out, we can see that moving one stack from p1 to another location, will roll out as shown alongside.

In this example, we're moving a collection of three containers from one location to another. And this is how that would roll out for that one pile.



But this still doesn't solve the problem, given we can now move one pile of containers from one position to another, but when we do so, **the ordering is now upside down**. So how do we resolve this? This can be resolved by creating another **METHOD** (shown alongside).

Move-Stack-Twice(p1, p2, p3)
Task: $move{-}stack(p1,p3)$
Pre: $None$
Subtasks:
$move{-}stack(p1,p2)$
$move{-}stack(p2,p3)$

In this case, we create a task that moves a stack twice, by running the move stack task twice but on a collection of three parameters. Our main locations we wish to move between and an intermediate. So if we were to consider the three pallets in location 1, p1a p2b and p1c, we would list them in this order, given we can then move the collection into an upside down configuration in the intermediary location, followed by then moving it back into the correct orientation in the final location. So, in essence, we can then solve the moving of one stack from a given location to another using the one task, which will then be decomposed into the recursive task, generating the take and put actions in the correct sequence.

Move-Each-Twice()
Task: $move{-}all{-}stacks()$
Pre: $None$
Subtasks:
$move{-}stack(p1a,p1b)$
$move{-}stack(p1b,p1c)$
$move{-}stack(p2a,p2b)$
$move{-}stack(p2b,p2c)$
$move{-}stack(p3a,p3b)$
$move{-}stack(p3b,p3c)$

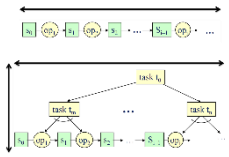
We can then roll this out one step further, by solving the entire problem using one method (Move-Each-Twice), which is to then call all these other methods.

This is naturally a rather trivial example and not intended to give us a complicated problem to solve. By using these methods for compound tasks, we can then solve the entire problem rather easily.

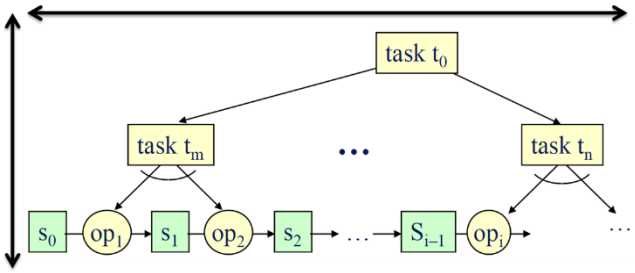
SEARCHING IN HTN

Unlike state-space planning where you either search forward or backward, we have the option of searching in 2 direction in HTN:

- **Backward/Forward**
  - Selecting Tasks
- **Up/Down**
  - Processing tasks into sub tasks or actions



When we reach a given task, it's important that we know what the current world state is prior to any attempt to roll it out. If we look at this example here, where tasks t_0 is rolled out into two more tasks t_m and t_n it's critical that before we attempt to enumerate task t_n that we have rolled out t_m before it. Given in this case, if there were only two tasks, then the outcome of task t_m is going to be used to check the preconditions of t_n as well as the actions that it translates into. Hence this exposes one of the issues that exists with this approach, in that **it is reliant on total ordering of tasks** such that it can rely on the state model.



TOTAL VS PARTIAL ORDERING

- The simplest approach (STN planning) is reliant on ordering constraints.
- Forward search required given need to test for applicability of the state.
- Interweaving actions need to be written into a more concise formalism.

- That said, partial ordering is feasible, but requires a more complex algorithm to achieve it.

STN (SIMPLE TASK NETWORK) VS HTN PLANNING

- HTN planning is a more general implementation of STN.
- Variety of formalisms and algorithms that are employed.
    - Forward Search – SHOP2
    - Plan-space Planning – O-Plan
        - Constraints can be associated with tasks and methods.
        - i.e. What facts are true at the before, during and after a task/method is executed.
- Use of goals/subgoals instead of tasks/subtasks.
    - GDP/GoDeL

MARKOV DECISION PROCESS (MDP)

PLANNING UNDER UNCERTAINTY

During planning the following issues can arise:

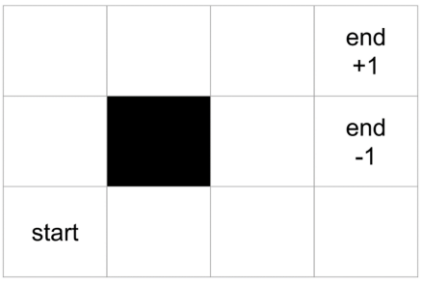
- **Partial Observability**: We might not know which state we are in
- **Non-Deterministic Actions**: We don't know if actions will succeed as intended
- **Non-Deterministic Effects**: Actions might not execute as intended resulting in possible new effects.

In **Classical Planning**, uncertainty is not handled, as we assume the following

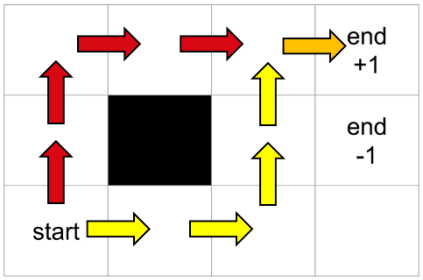
- The state is known
- All states in the problem are known
- That we know all actions from the current state
- That a given action will always execute as intended
- That the successor states of action execution are also known

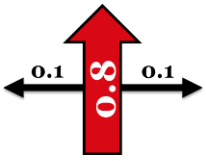
However, this is not always the case with even some of the most trivial examples, for example: a coin toss.

Example: Consider this example here, where we have a problem space whereby, we start in the bottom left corner of the world and there are two goal states, each with a utility value of +1 or -1.
Ideally, we want to find the goal state with a utility of 1. The actions you have are to more in any direction from the current cell, so if you're at start you can move up or right. Attempting to move left or down would fail in this instance given those are the environment

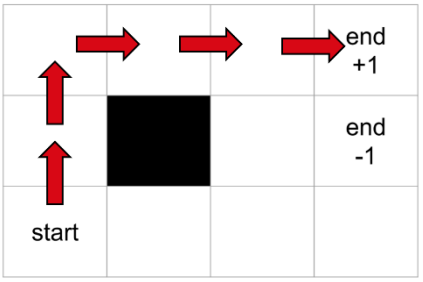


Now if we use **A*** with a simple distance to goal heuristic (either using Manhattan or Euclidean distance), then the path to an optimal solution is very easy to find. In fact, there are two optimal solutions, with the red and yellow paths splitting at the initial state and then converging at the position left of the goal state, where they both take the same action denoted in orange. In fact, even using something like breadth-first search will find the answer to this question. And while it would be more demanding if the search space was any bigger, it's still easily solvable.





We can add randomness to the problem – in any state, any action applied only has an 80% success rate. But there's a 10% chance the agent will go either direction perpendicular to the intended direction.



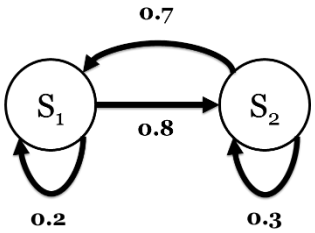
Thus, with this uncertainty, the original plan $(Up, Up, Right, Right, Right)$ has a success rate of $0.8^5 = 0.327$.

MARKOV CHAINS

Markov Chain: a finite set of discrete states, each of which then has probability distributions for how we handle transitions between them. This allows us to capture the idea that sometimes state transitions don't work out as prescribed.

Markov Property: assumes that any future state outcomes is only influenced by the current state and the probability distribution. In other words, the states visited prior to this one do not influence the outcome for the current state. If this doesn't hold true, then the previous states we've already visited could influence the probability distribution and that would mean it would not be reasonable to assume that these probabilities mean anything.

Example: (shown alongside) there's a probability that the transition from S_1 to S_2 doesn't work and instead we find ourselves in the same state we're currently in. Essentially there's a 20% chance that the transition to go from S_1 to S_2 turns into a no-op and we stay in S_1. Same with S_2 only this time there's a 30% chance it's going to happen.



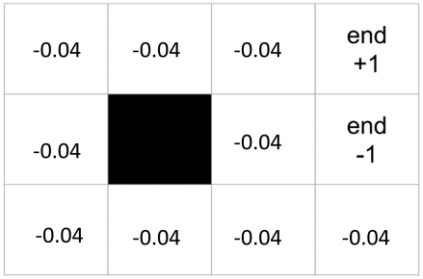
MARKOV DECISION PROCESS (MDP)

A sequential decision-making problem for a **fully observable** but **stochastic** environment.

- MDP is now built around **transition model**: T(s, a, s')
- Similar to existing **state transition**, but now encodes P(s'|s, a)
- We will need a **utility function**, that will help determine value of an action against the probability distribution based on state **rewards** (values that influence the decision-making process).

Example: We can now encode rewards for each state. We then hope to achieve the maximum utility, which is going to be sum of all rewards as we reach the goal based on the actions that we took and the states we were in.

Like we said earlier, the rewards are going to help incentivise us to make good actions. This actually completely removes the notion of cost of actions to get to the goal state, but that is something than can with some effort be encoded into the reward values for a given state.



In this case, we've set the reward values as -0.04 for every single state. We don't have to make these rewards uniform, but for the sake of the example we're going to keep it really simple. Although later in these slides I'm going to change the reward values given this will influence how we solve the MDP.

If we assume the original optimal solution from start to end, then we can say that the utility value of that resulting plan is 0.8, given we earned 1.0 for reaching the goal and then were penalised a total of 0.2 by starting in a state with a reward of -0.04 and then visiting four more states on the way to the goal.

In this case, all the rewards are negative, because if we made the rewards positive there is no real incentive to reach the goal. If positive rewards are always given and the system is told to maximise the utility, then it would simply go on random walks that avoid the goal states. So instead, by making the rewards negative we reinforce the need for optimal solutions, given the more actions taken, the greater the overall penalty in the utility function.

MDP Components for all States $s \in S$

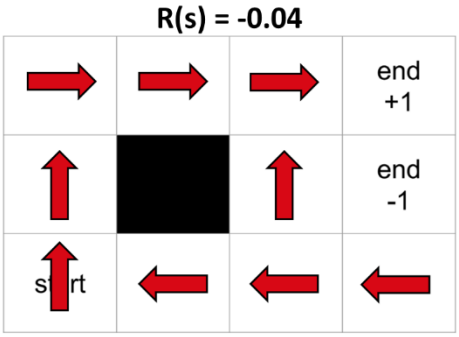
- Transition Model: $T(s, a, s')$
- Initial State of the Problem: s_0
- Reward function for a given state $R(s)$

MDP solution is not a plan of actions. MDP solution is a **policy**: π

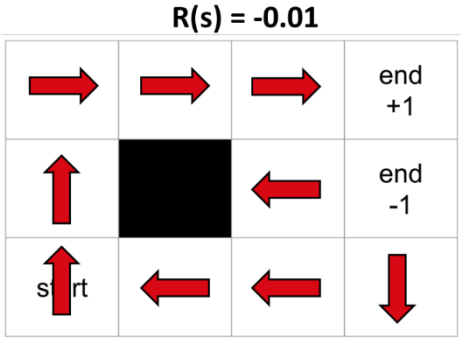
- For any state $s \in S$, $\pi(s)$ denotes what action should be taken in that state.
- This means you can different resulting solutions from the initial to goal state, given the stochastic nature of the environment.
- Hence the optimal policy ($\pi*$) will yield the highest expected utility in each situation.

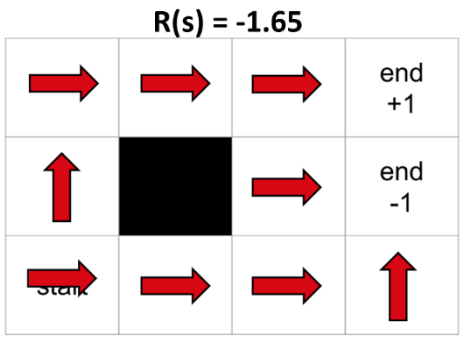
Optimal policy attempt to balance the risk against the reward. Results in actions that will yield the best utility regardless of how successfully we are in execution. These are highly sensitive to action probabilities.

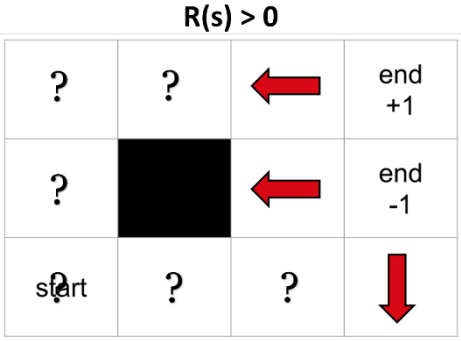
In the example given alongside, this policy ensures (given the probability distribution of these actions), that it will yield the best utility for every state.



Given below are some examples of how the optimal policy can change based on the reward value.







Alternative Approaches to Solve MDPs

- Value iteration
- Policy Iteration
- Linear Programming

Planning Under Uncertainty holds provided we have this complete model, i.e. The transitions, reward functions and fully observable states. When one or more are not available, this becomes a reinforcement learning problem (model free) and we need to interact with the environment to learn these features.

PARTIALLY OBSERVABLE MARKOV DECISION PROCESS (POMDP)

HIDDEN MARKOV MODEL

What happens if we actually don't know for certain what state we're in anymore? If we don't know for sure what state we're in, we need to rely on some information that would allow us to make some sort of education assumption as to what state we're in.

Thus, the **Hidden Markov Model** is exactly the same as the original Markov Models, i.e., we model the states, the probability transitions between states using some reward function, but, we cant actually see the Markov Chain, i.e., we can't see the states (even though we know they exist).

PARTIALLY OBSERVABLE MARKOV DECISION PROCESS (POMDP)

This is a MDP where we can't see the states, instead we are reliant on the observations that give us a rough approximation of what state we could be in and based on the actions we take, it will influence where we are.

- Modelling probabilistic transitions between states.
- Next state is only determined by probabilistic transition between states.
- Does not factor the history of previous states.
- But is unsure of what state we are in, reliant on observations to build history of states visited.

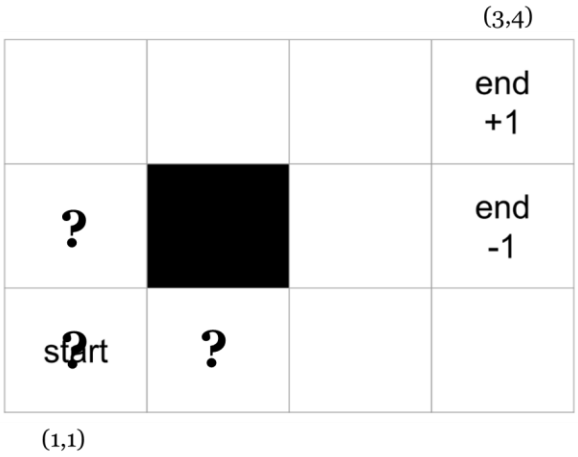
A POMDP is **comprised** of the following:

- **States** (S), **Actions** (A)
- **Transition probabilities** $P(s'|s,a)$
- **Reward** accrued $R(s)$
- A **belief vector** b – the probability distribution of which state we are in
- A **sensor model** $O(o|s)$ which indicates based probability we would have seen an observation in a given state
    - the probability that we received a given observation in a given state.

It's important once again to state the MDP exists, the states exist, we just can't see any of it. We just don't know what the current state is. Hence, we need to use the information of observations to help us figure that out.

BELIEF VECTOR

Looking back at the grid example, in order to make intelligent decisions, we need to take the observations and compute a **belief vector** using subsequent observations. The belief vector is the probability that we are in each state. And this is a useful thing to model, given there is a chance that based on the observations we could be in multiple states. If we consider that there's an 80% chance we are in the starting state, which is that cell 1,1 in the bottom left hand corner. And a 10% chance we're either in the cell above it or the cell to the right, then the belief vector will look like it does on the bottom of the screen. With 0.8 in the first entry, assuming that this list enumerates from 1,1 up to 4,4 but ignores 2,2 given that this cell is not possible to ever be at. Thus, it's not a valid state.



$$\boldsymbol{b(sini_t)} = \{\mathbf{0.8, 0.1, 0, 0, 0.1, 0, 0, 0, 0, 0, 0}\}$$

given observation, ignoring the blank space in (2,2)

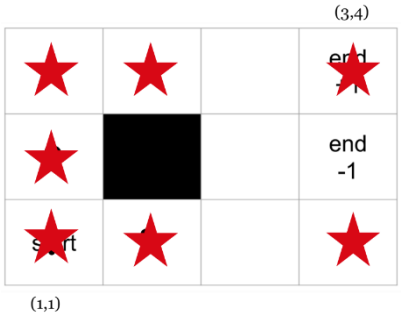
$$\boldsymbol{b(s_{init})} = \{\frac{\mathbf{1}}{\mathbf{9}}, \frac{\mathbf{1}}{\mathbf{9}}, \frac{\mathbf{1}}{\mathbf{9}}, \frac{\mathbf{1}}{\mathbf{9}}, \frac{\mathbf{1}}{\mathbf{9}}, \frac{\mathbf{1}}{\mathbf{9}}, \mathbf{0}, \frac{\mathbf{1}}{\mathbf{9}}, \frac{\mathbf{1}}{\mathbf{9}}, \frac{\mathbf{1}}{\mathbf{9}}, \mathbf{0}\}$$

But of course, all of this relies on us receiving that observation. To begin without any observation, we have a 1/9 probability of being in any state that isn't the goal

CALCULATING BELIEF FROM OBSERVATION

Now given our belief state, we need to be able to compute that based on the observations. In this case, we can use the observations as evidence to support a given probability.

Let's say that in each state, we receive an observation of how many walls there are in proximity of the robot. We don't get the orientation, just the number. In this example, there's no cell where that value is less than 1. In fact, the only cell that could have been is 2,2, but it's blocked. Hence the value of this observation is only ever 1 or 2. And based on that we can then calculate what the probabilities are for the belief state, based on the observation.



In this case, if we see two walls, then we know we are in one of seven states in the grid, which I have marked, as being the possible states. Hence the belief vector sets a value of 1/7 or 0.143 for each of these cells and sets 0 for everything else.

$$O(o = 2\,|s) = \frac{1}{7}\ \forall\ s\ \in \{s_{11}, s_{21}, s_{41}, s_{12}, s_{31}, s_{32}, s_{34}\}$$

Now it's important to acknowledge the having this knowledge, such that I can say with confidence that based on the signal I know what state I am in, has to be something that is pre-built into the model. If I can't determine that (1,1) has two walls surrounding it beforehand, then receiving this observation really isn't important.

The belief state needs to be updated based on actions taken in the world. But given we're unsure of the state we're in and whether action execution is successful, we must factor that into the belief state update. Update upon taking action a and observing o (shown alongside), where

$$b'(s') = \alpha\, O(o|\, s') \sum_{s \epsilon S} P(s'|\, s, a) b(s)$$

- $O(o|s')$ is the probability that we are likely to see observation o at a state s' given action a
- $\sum_{s \in S} P(s'|s,a)b(s)$ is the sum of successful action transition from state s to s', given the belief state that you were in that state to begin with.
- α is the **normalising constant**, that makes the belief state sum up to 1.

Example: Assuming we have no observations and execute the *Up* action, and then observe one wall, what's the new belief state (Assuming α=1)?

$$b(s_{init}) = \left\{\frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0\right\}$$

$$b(s') = \{0, 0, 0.02, 0, 0, 0.1, 0.1, 0, 0, 0.1, 0\}$$

Example: Assuming we have no observations and execute the *Up* action, and then observe one wall, what's the new belief state? Typically, the normalisation factor can be considered as 1/ the sum of observation probabilities in s' multiplied by the probability of reaching s' based on action a, given belief state s. By doing this instead, we result in a much stronger set of probabilities for the belief state.

$$b'(s') = \frac{O(o|\, s', a) \sum_{s \epsilon S} P(s'|\, s, a) b(s)}{\sum_{s' \in S} O(o|\, s', a) \sum_{s \epsilon S} P(s'|\, s, a) b(s)}$$

$$b(s_{init}) = \left\{\frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0\right\}$$

$$b(s') = \{0, 0, 0.06, 0, 0, 0.31, 0.31, 0, 0, 0.31, 0\}$$

NUMERICAL PLANNING

Till now, planning has been focused on predicates with True or False values, but this is not always the case in the practical applications. This is possible using **state variables**

As shown in the example given alongside, total-cost is a state variable. We can create many state variables and have effects on them or use them as conditions in the preconditions of actions.

```
(increase (total-cost) 3)
```

NUMERIC VARIABLES IN PDDL

Consider the example given alongside. Just like we define predicates for a PDDL problem, we can also define functions.

In this case we have a function for `time-to-walk` and `time-to-drive` between locations. Similarly, we also have values for `driven` and `walked` which can keep track of how much of each activity we have done during the plan.

```
(:predicates
   (at ?obj - locatable ?loc - location)
   (in ?obj1 - obj ?obj - truck)
   (driving ?d - driver ?v - truck)
   (link ?x ?y - location) (path ?x ?y - location)
   (empty ?v - truck)
)
(:functions (time-to-walk ?l1 ?l2 - location)
   (time-to-drive ?l1 ?l2 - location)
   (driven)
   (walked)
)
```

Thus, in addition to assigning propositions that are true in the domain, we also need to provide initial values for all numerical variables.

NUMERIC PRECONDITION

As mentioned earlier, numerical values can be used in preconditions for actions. These numeric preconditions comprise of the following

- A comparison operator: $>, >=, =, <=, <$
- A left- and right-hand side written using constants, the values of functions, and the operators $+, -, *, /$

```
(>= (fuel) (* 5 (distance ?from ?to)))
(<= (height truck) (height barrier1))
      (>= (number-of-widgets) 3)
```

The preconditions are written in prefix form. Thus, the examples can be rewritten as follows

```
(fuel) >= (5 * (distance ?from ?to))
(height truck) <= (height barrier1)
      (number-of-widgets) >= 3
```

NUMERIC EFFECTS

A PDDL numeric effect comprises of the following

- A variable to update
- How to update it: assign (=), increase (+=), decrease (−=)
- A right-hand-side formula, as in preconditions

```
(decrease (fuel) (* 5 (distance ?from ?to)))
 (increase (number-of-widgets) (capacity))
 (assign (battery-charge b) (max-charge b))
```

The effects are also written in prefix form. Thus, the first example (given above) can be rewritten as follows

```
fuel -= (5 * (distance ?from ?to))
```

METRIC (OBJECTIVE) FUNCTION

Now we have numeric state variables we can define a metric function for the planner to optimize. Some examples are given alongside. These metrics refer to the values of the variables after the plan has finished executing, i.e., in the goal state. Defining a metric function is **optional** for numeric planning problems.

```
(minimize (total-cost))
(minimize (+ (fuel-used) (*2 (wages))))
```

PLANNING WITH NUMBERS

- Search is straightforward as these are discrete effects (happen instantaneously) so
  - We check numeric preconditions are satisfied before applying actions
  - Update the values of the variables according to effects when we apply actions

The more interesting point is to guide the search using a heuristic.

NUMERIC RELAXED PLANNING

Since relaxed planning works by ignoring delete effects (as shown alongside), for numeric relaxed planning, we must ignore numeric delete effects.

In the example given alongside, we can say that `(decrease (fuel ?x) 1)` is a delete effect using intuition, as it deletes a unit of fuel which is needed to perform a `move` action. Thus, it can be ignored in the relaxed problem.

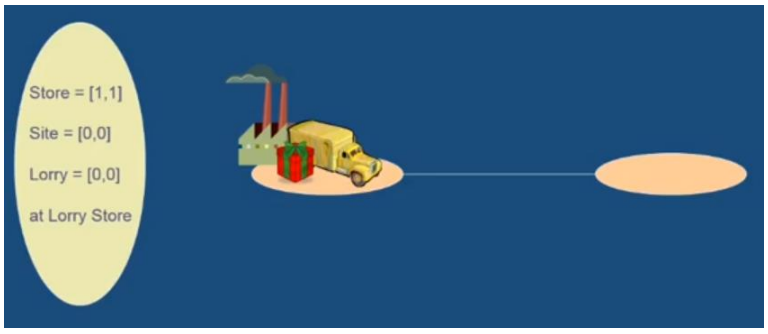
```
(:action move
  :parameters (?x - object ?from ?to - location)
  :precondition     (and
                    (at ?x ?from)
                    (>= (fuel ?x) 1)
                )
  :effect (and
                (at ?x ?to)
                (not (at ?x ?from))
                (decrease (fuel ?x) 1)
))
```

But what if there is an action `(decrease (undelivered-packages) 1)`, where the goal is `(=(undelivered-packages) 0)`. In this case, it is not a delete effect, as we want fewer undelivered packages. Ignoring this effect in the relaxed problem would make the problem unsolvable.

Thus we need to **Generalize Relaxing Numeric Effects**:

- Maintain bounds: upper and lower bound on each numeric variable
- Use lower bound for preconditions $v \leq c$ and upper bound for $v \geq c$

Example. There are 2 packages at the store. These packages need to be delivered to the site.
The brackets represent the lower and upper bounds of the values, i.e., $[lower, upper]$, the minimum and maximum value that variable can have. In the initial state, the store has 1 package, thus the lower and upper bound are 1. As there are no packages in the lorry or on the site, the lower and upper bounds for them are 0.

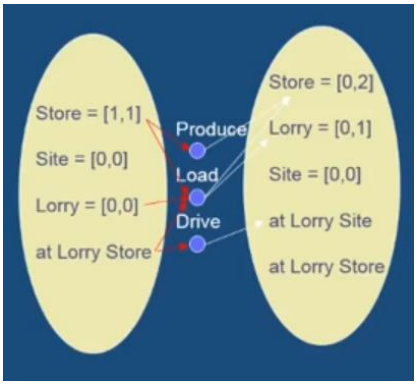


There are 3 actions applicable in the initial state

- `Produce` packages at the factory (store). This action has no precondition, we can produce whenever we want
- `Load` packages onto the lorry

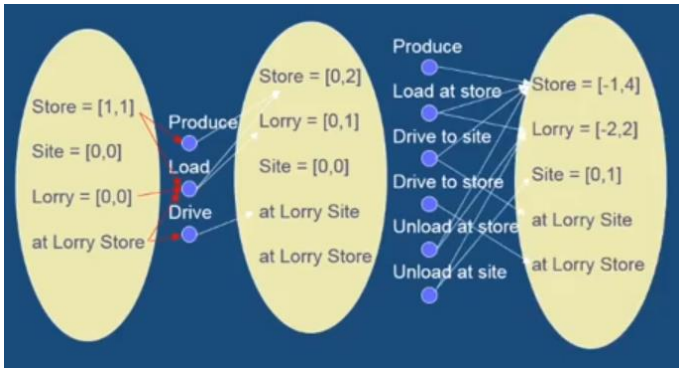
- `Drive` the truck to the site

Thus, if we apply these actions to the planning problem, we get the state shown alongside.



- We can `produce` more packages (as there are no preconditions).
  - Increases upper bound of `store` to 2
- We can `load` the package onto the lorry
  - Increases upper bound of `lorry` to 1.
    - Lower bound remains unchanged as no actions have been applied to decrease the lower bound
  - Decreases lower bound of `store` to 1
- We can `drive` the `lorry`
  - Adds the proposition: `at Lorry Site`

In this state, we can apply the following actions: `Produce`, `Load-at-store`, `Drive-to-site`, `Drive-to-store`, `Unload-at-store`, `Unload-at-site`. When we apply these actions, we can see the following effect on the variables:



Store:

- Upper bound increased by `produce` and `unload-at-store` to 4
- Lower bound decreased by `load-at-store` to -1

Lorry:

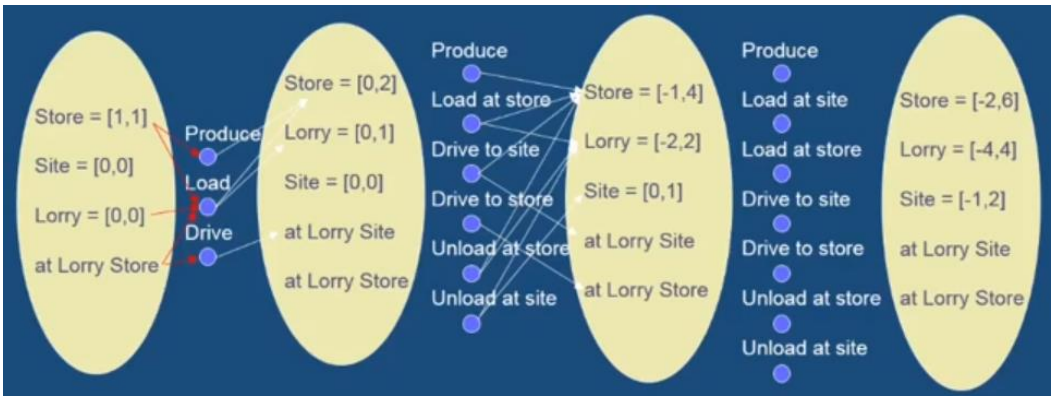
- Upper bound increased by `load-at-store` to 2
- Lower bound decreased by `unload-at-store` and `unload-at-site` to -2

Site:

- Upper bound increased by `unload-at-site` to 1
- Lower bound remained unchanged.

In addition the actions previously applicable, `Load-at-site` can now be applied to the RPG.

The final stage of the planning graph is shown alongside. Since the upper bound of site has reached 2 (using the same method used for the previous steps), we have reached the goal state.



The **termination condition** in numeric RPGs is

- All propositional goals are true
- All the numeric values should have a range which includes the goal value

Solution extraction for numeric problems is the same as normal RPGs.

SEARCHING FOR A BETTER SOLUTION

Once a solution has been found we can continue searching:

- We can easily compute the cost of the plan so far, simply be evaluating the metric function in the current state
- Assuming the cost monotonically increases as we add actions to the plan, we can prune states for which $cost(S) > cost^*$, where $cost^*$ is the best cost we have found so far
- Alternatively, we can just restart research, adding a goal ($cost < cost^*$)

The problem here is that a heuristic is not necessarily providing guidance to find better solutions, it might keep sending us back to the same one.

PLANNING WITH PREFERENCES

Usually, when we are planning, the objective is to just find a sequence of actions which would yield the goal state. But this might not always be the best plan, as we might not only care about the goal of the problem, but also how the goal is achieved. On the other hand, if a planner can't reach the goal, it would be useful to know that the plan was able to satisfy some of the problem (this specific case is called *Over Subscription Planning*, where we have more goals than what the planner can achieve).

Preferences are things that we would like to be true, but don't necessarily have to be satisfied before the plan delivers a solution. They can be of the following types:

- **Simple Preferences**: soft goals and preconditions
  - `(p0 (at end (at rover waypoint3)))`
- **Trajectory Preferences**: condition on the plan
  - `(p1 (always (>= (energy rover) 2)`
  - `(p2 (sometime (at driver costa-coffee))`
  - `(p3 (at-most-once (at truck Birmingham)))`
  - `(p4 (sometime-after (at Birmingham) (at Glasgow)))`
  - `(p5 (sometime-before (at Birmingham) (had-lunch)))`
- **Temporal Preference**: withing t units, we need to achieve a
- **Metric Function**: Is used to associate weights with preferences as some preferences are more important than the others, as these are soft goals and don't need to be satisfied in a plan. An example is shown below which can be used to minimize fuel usage and suggest that `p0` and `p1` should not be violated.
  - `(minimize (+ (fuel-used) (*2 (is-violated p0)) (*5 (is-violated p1)))`

Planners handling preferences (not all planners can handle all preferences):

- Simple Preferences: *YochanPS, Keyder & Geffner translation.*
- Simple and Trajectory Preferences: *Hplan-P (non-numeric),* ***LPRPG-P*** *(propositional and numeric).*
- Simple, Trajectory and Temporal Preferences: Mips-XXL *(numeric),* ***OPTIC***.
- Competition Domains: SGPlan

MODELLING PREFERENCES IN PDDL 3.0

Consider the example, where we have the following objectives for the day

- Get to work
- If I do that, get a coffee on the way
- If I get a coffee, go to the loo after
- Play the piano (Work/Life balance achieved!)

If we use the classical approach, then we would have a PDDL like the one shown alongside. But this would mean that at the end of the day, Amanda is still at work.

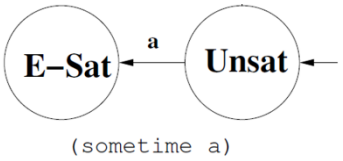
```
(:goals  (and
       (at amanda work)
       )
)
```

That should not be the case; what we want to say is that at some point during the day, Amanda should be at work.

Thus, we need to use preferences have the following preferences:

- `(preference p0 (sometime (at Amanda work)))`
- `(preference p1 (sometime (at Amanda piano)))`

Given alongside is the automaton representing the `sometime` preference. When we enter the automaton, we are in the *Unsat* (unsatisfied) state. Once `a` is satisfied, we reach the *E-Sat* (eternally satisfied) state, i.e., this condition can now never be unsatisfied.



(sometime a)

Each preference needs to have an associated cost for violating the preference:

- `cost(p0) = 100`
- `cost(p1) = 5`

Thus, using preferences, we can remodel the PDDL as shown alongside. This would mean, that end of the plan, Amanda would be in her bed, but at some time, she should be at work, and at some time she should be at piano.

```
(:goal (and
    (at amanda mybed)
))
(:constraints (and
    (preference p0 (sometime (at amanda work)))
    (preference p1 (sometime (at amanda piano)))
))
```

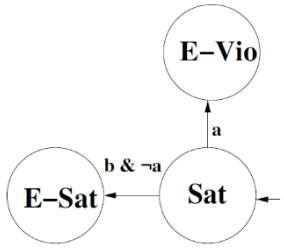
Looking back at the goals for the day, we wanted to visit the coffee shop on the way to work (if we go to work). In this case, `sometime` cannot be used as it does not impose any ordering on when we go to the coffee shop (it needs to be before work). Thus, we can use the `sometime-before` preference:

- `(preference p2 (sometime-before (at Amanda work) (at Amanda coffeeshop)))`

The automaton for `sometime-before` is given alongside. For our example,

- a = `(at Amanda work)`
- b = `(at Amanda coffeeshop)`

Thus, if we go to work, without going to the coffee shop, we would reach state *E-Vio* (eternally violated). But if we go to the coffeeshop while we haven't been to work yet, then we reach the state *E-Sat* (eternally satisfied).



(sometime-before a b)

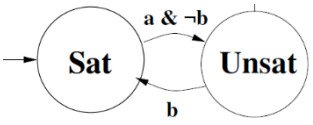
The other objective for the day was: if I get a coffee, then I will go to the loo. In this case we cannot use `sometime` or `sometime-before`; we can use `sometime-after`.

- `(preference p4 (sometime-after (at Amanda coffeeshop) (at Amanda loo)))`

This would imply that Amanda would go to the loo at some point after having been to the coffee shop, i.e., if Amanda went to the coffee shop, then at some point Amanda will go the loo.

The automation for `sometime-after` is given alongside. For this example, we have

- a = `(at Amanda coffeeshop)`
- b = `(at Amanda loo)`



(sometime-after a b)

If Amanda goes to the coffee shop and not the loo, then we reach the unsatisfied state. Once at that state, if b is satisfied, then we go back to the satisfied state. Thus, there is no eternally satisfied or eternally violated state, i.e., there is always an opportunity to satisfy the preference, but there is also an opportunity to violate the preference. It is important to note that to go from *Sat* to *Unsat*, b needs to be false, if b becomes true before a becomes true, and it is never made false, then we cannot move to the *Unsat* state.

It is important to note that the following are not the same

- `(sometime-before (at 84manda work) (at 84manda coffeeshop))`
- `(sometime-after (at 84manda coffeeshop) (at 84manda work))`

As these are not inverses:

- If I got to work, then I have to have coffee first
- If I drink coffee, then I have to go to work afterwards

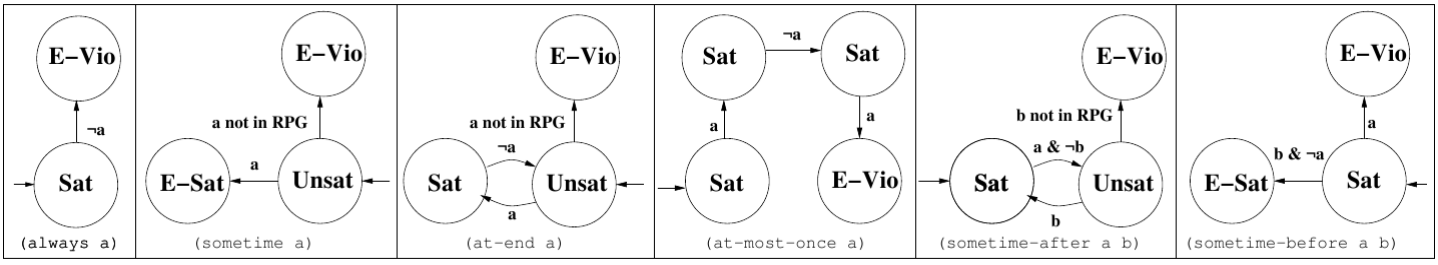
It is also important to note that the preferences `(and (sometime-before (b) (a)) (sometime-after (b) (c))` will not force the order `a, b, c` (could be possible), but these preferences can also be satisfied by the order `b, c, a, b`.

We can also add **Goal Preferences** as shown alongside.

```
(:goal (and
    (at amanda mybed)
    (preference p5 (switched-off phone))
))
```

LPRPG-P: REASONING WITH PREFERENCES

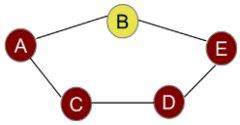
The following are the automata for the other preference conditions (can be generated automatically from PDDL3). At each state, maintain an automaton corresponding to each preference; each starts in initial position and is updated according to the initial state. Each time an action is applied (state is expanded), the automaton is updated if condition from the current position fires. Thus, the **preference violation cost** $PVC(S)$ is simply the sum of violation costs of all automata (need to be defined in the problem) that are in *E-Vio*.

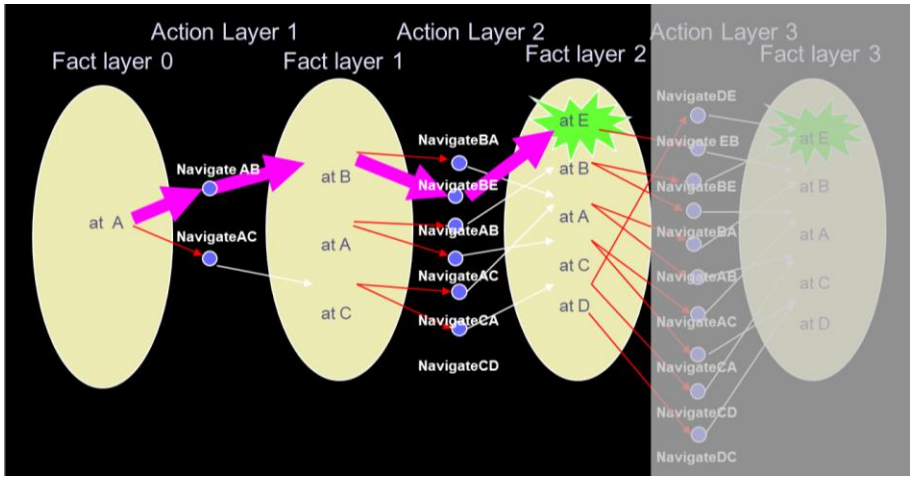


In **LPRPG-P**, **search** continues until a solution is found, i.e., a plan that satisfies all the hard goals (preferences may or may not be satisfied). Once a goal is found, continue search, while pruning all the states where $PVC(S) >$ cost of best solution so far. Therefore, it is important to consider the preferences which are in the *E-Vio* state as for the preferences which are in *Unsat* state, we can still perform some actions to potentially satisfy them.

Planning heuristics generally focus on finding a path to the goal in a relaxed problem. Typically, these heuristics look for a short path to the goal. If we have a problem with no hard goals and only preferences/soft goals, the RPG heuristic value is zero for every state. Thus, the main focus of LPRPG-P is to have guidance about how to satisfy preferences and how difficult it will be to do so.

Example: Consider the example given alongside. Initially, we are at state A, and we would like to get to state E.



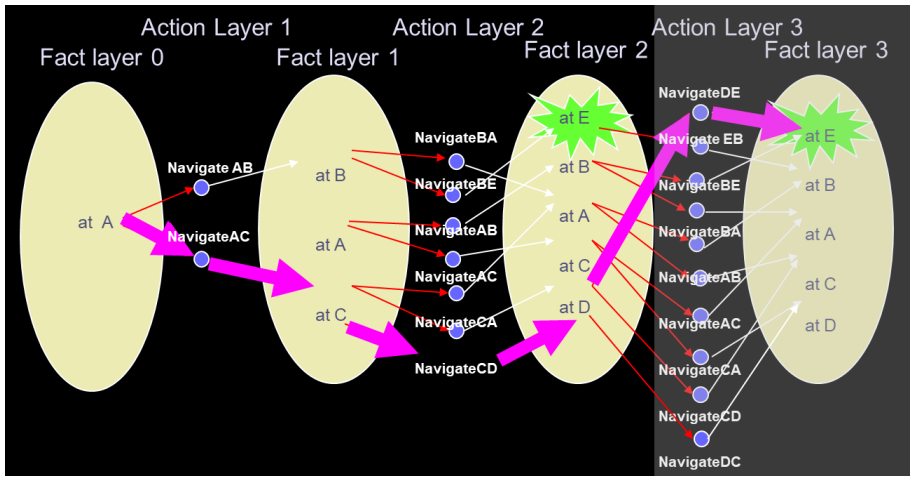


Thus, in the RPG shown alongside, F_0 just has `at A` (initial state). This expands to F_1 and F_2 as shown alongside. The goal for the problem has appeared in F_2, the relaxed plan can be computed by tracing the actions. The plan in this case is A, B, E.

But let's say that we have a preference as follows: `(preference p1 (always (not (at B))))`

Thus, the plan we are looking for is A,C,D,E and not A,B,E. But we cannot reach that plan as the states have not been expanded to that point since we already found our goal in F_2.

Thus, we need to expand the planning graph to F_3, so that we can get the plan, A, C, D, E.



REQUIREMENTS FOR PREFERENCE-AWARE RPG PLANNER

- **Termination Criterion**: Stop building the graph when goals appear insufficient.
- **Mechanism for selecting right achiever**:
  - The earliest achiever (fact layer with the goal) is not always the best one
  - Arbitrary (random) achiever could miss something good
- Effectively we need to **track knowledge about preferences** whilst building the RPG.

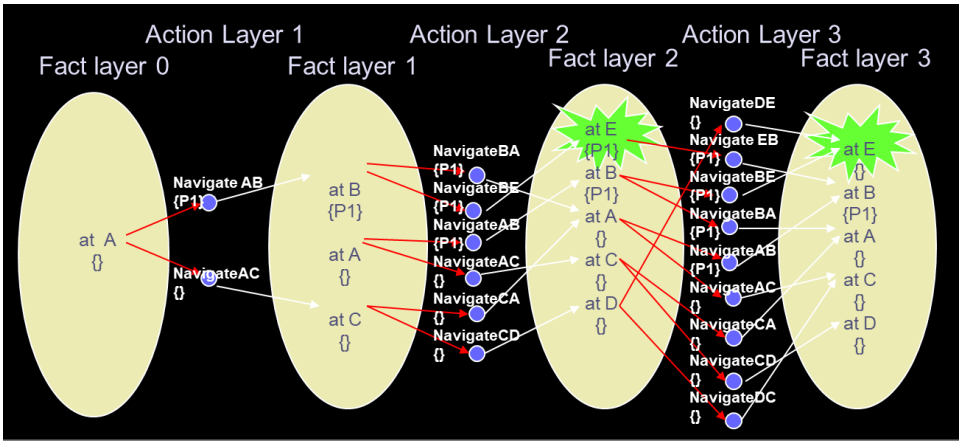
A PREFERENCE AWARE RPG

- At each fact layer we maintain a **set of preferences** for each fact
  - These are the preferences that are violated in achieving the fact at this layer
  - For facts that appear in the current state this is empty
  - The preference violation set for a newly appearing fact is that of the action that achieves it, and the union of those for the action's preconditions
  - If a new path to a fact is discovered, us the lowest cost of its existing/new sets
  - Otherwise the set at that layer is the set from the previous layer

- Now we can build the RPG to the point at which (**Termination Criterion**)
  - No new actions appear
  - No preference violation sets are changing

This does mean that must build more layers in the RPG; thus, it generates potentially longer relaxed plans that satisfy preferences.

Example: Looking back at the previous example, if we use the preference aware RPG, we can maintain a set of violations at each stage.



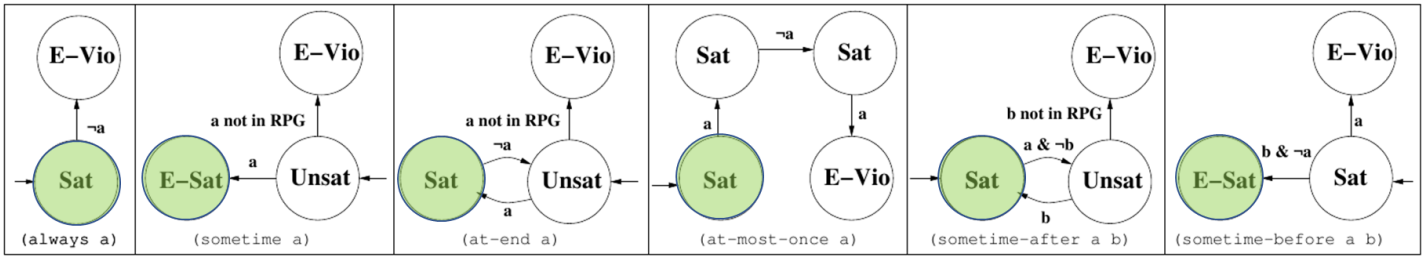
Since our preference `(preference p1 (always (not (at B))))` suggested that we should not go to `B`, the actions that lead to `B` violate that preference. Thus, the actions which do not violate any preferences, have an empty preference violation set. So, when we look at F_2 in this case, we can see that `at E` has `p1` in its preference violation set. Even though we have reached the goal we will keep searching until our new termination condition. This way we get to the F_3 which does not violate the preference `p1`.

Important: Thus, the preference violation set for a newly appearing fact is that of the action that achieves it, and the union of those for the action's preconditions.

OPTIMISTIC AUTOMATON POSITIONS

In PDDL 3, there is always a best position in the automata so we can maintain the best position possible for RPG layers (the further from *E-Vio* the better). Thus, we can say that an action A violates preference P when applied in layer l if it activates a transition from (all) automaton position(s) in l

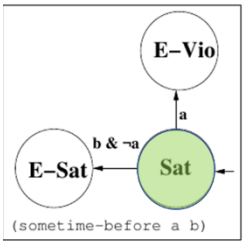
- To *E-Vio* (e.g. sometime-before, at-most-once)
- To *Unsat* and the facts required to activate the transition back to a *Sat* state are not yet in the RPG (e.g. sometime-after)



REAPPEARING ACTIONS

If an action appears later in the RPG with a lower preference violation cost:

- Add it again with precondition that made the automation transition
- Propagate to its effects and actions that rely on them by building further action layers until no further change in violation sets.



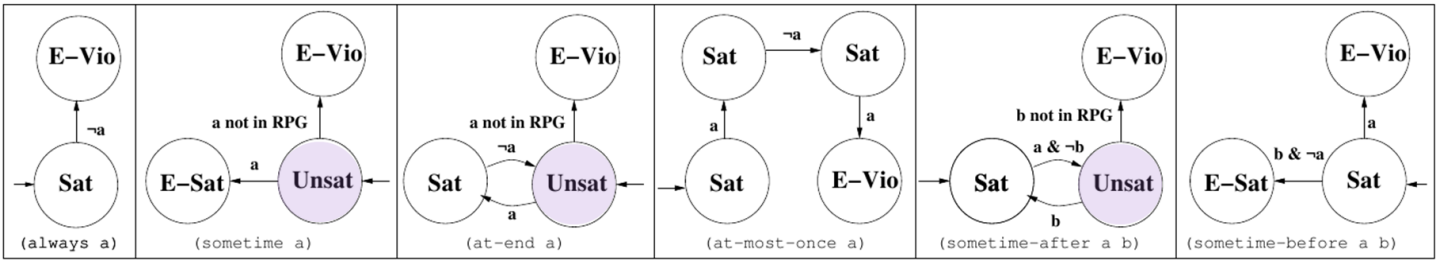
Then continue building RPG as normal. **Termination Criterion**: no other facts/actions appearing, and preference violation sets unchanging

SOLUTION EXTRACTION – GOALS

Not all preference problems have hard goals, so we need to decide what goals to achieve.

- Some soft goals are explicit in the problem, e.g. `at-end`
- Others are implicit in currently *Unsat* preferences that we would want to move to *Sat* or *E-Sat* (`sometime-after, sometime`)
    - Thus, for the sates in purple below, we want to add to the goal, the facts that perform the transitions to either *Sat* or *E-Sat* states.

For goal generation, look to automata: any automaton which is currently *Unsat*, add the condition on its transition to *Sat* as a goal in the RPG. More generally disjunction over paths to *Sat*.
If the transition does not appear in the RPG, then preference is *E-Vio* not *Unsat*.



SOLUTION EXTRACTION – RP GENERATION

In order to generate the relaxed plan, we now consider the following

- Add **preconditions/goals** at earlier layer they appear with their **lowest cost violation set**
    - This must be before the layer the action for which it is a precondition was applied

We also need to consider which **achievers** to use:

- Select the achiever that caused this fact's cost to be updated at this layer (recorded during graph building)
- This is the achiever that caused its cost to decrease and is thus on the lowest cost path.

There's no need to worry about adding extra goals (e.g. if we had a (`sometime-before a b`) we don't need to add `b` as a goal if we choose `a` in extraction because the action that doesn't break that preference already has `b` as a precondition)

IMPROVING LPRPG-P USING DISTANCE-COST PAIRS

Consider the PDDL goal given alongside. We can assume that the cost of violating each preference is 1. So, if we violate all preferences, then the cost will be 4.

- **Distance to Go**
    - The distance to reach the goal state of the problem, i.e., minimum number of actions to call something a goal state.
    - In this case we can say its 0 since we don't have any goals, just preferences
- **Cost to Go**
    - Cost of getting to the goal state
    - If we say that the distance is 0, i.e., we violate every preference, then the cost to go is 4.

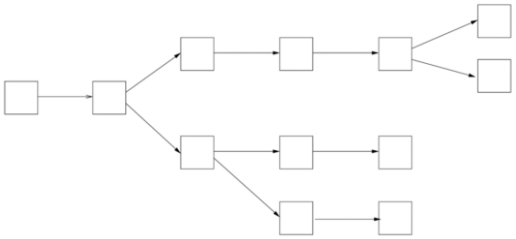
```
(:goal (and
 (preference p1a
    (at package1 s1))

 (preference p2a
    (at package2 s0))

 (preference p3a
    (at package3 s2))

 (preference p4a
    (at package4 s0))
 ... )
)
```

- $c(S) = 0$, when we apply all the actions to meet all the preferences, so that there is no preference violation cost.

Thus, for every problem, there is a trade-off between distance to go and cost to go. Thus, we have **distance-cost pairs**. Example:

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

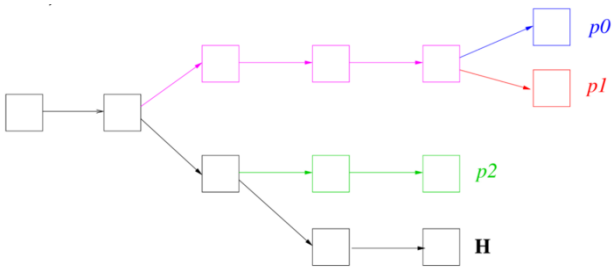
We can build these pairs using RPGs. Consider the example shown alongside, which is an abstract representation of an RPG. Each box is an action, each vertical column of boxes represents action layers of the RPG and the arrows between the boxes represent the dependencies.



If we start by pretending all goals are hard goals, i.e., all preferences are also hard goals, and create an RPG, then these actions achieve the following

- Meet the actual hard goals (if any)
- Meet some preferences

We can record this by building the relaxed plan backwards. All the actions going back from the hard goal are coloured in black, the preferences are in other colours. **Annotations** are these labels which have been associated with the RPG actions.



In this case the following are the **Distance-Cost pairs**:

- *5 actions = cost 3*
  - i.e., we achieve only the hard goal H, which requires 5 steps. All 3 preferences are violated.
- *7 actions = cost 2*
  - We satisfy the hard goal H and $p2$, but $p1$ and $p0$ are still being violated
- *11 actions = cost 1*
  - We satisfy $H, p2, p1$ but $p0$ is still being violated
- *12 actions = cost 0*
  - We satisfy all hard goals and preferences.

There were other distance cost pairs to consider in the above problem as well: 9:2 ($H, p0$ satisfied) and 10:1 ($H, p0, p1$ satisfied). Trying all combinations of preferences would give many more distance cost pairs, but thing will increase the time taken to compute the heuristic exponentially. Thus, the compromise is to use a **Greedy Algorithm**: start with the hard goals and then add preference on at a time:

- **Greedy by Length**: Add the preference that requires the fewest actions to be added to the relaxed plan in order to satisfy it (minimise distance to go)
- **Greedy by Cost**: Add the preference that reduces the cost of the resulting relaxed plan by the most (i.e. the most expensive one that is still violated).

But statistically and in practice, there is no difference between the 2 approaches listed above.

Thus, we have a trade-off in search; we want to explore good states that are close to satisfying all the preferences (high distance, low cost), but also prioritise reaching better states quickly (low distance, higher cost). The solution is to use **dual open-list search**:

- One open-list sorted by $h(all)$ the length of the relaxed plan to satisfy the preferences
- One open list sorted by $h(< C)$ the shortest distance from a distance cost pair which has a cost less than the cost of the current solution,

Search by alternating and putting new states on both open-lists.

TEMPORAL PLANNING: INTRODUCTION AND MODELLING IN PDDL 2.1

Motivation: In general, activities have varying durations, for instance, loading packages onto a truck is much quicker than driving the truck. Thus, to tackle such scenarios, we can use **Durative Actions**.

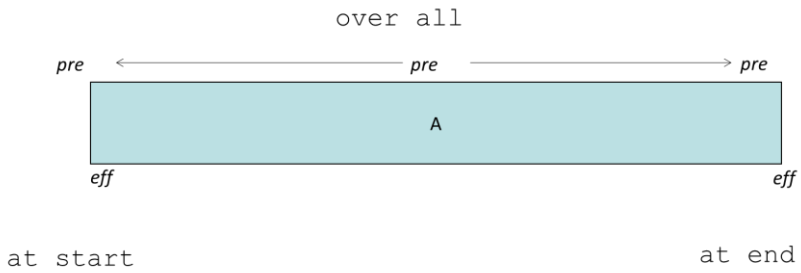
In **classical planning** (for example: the FF planner), all actions are treated as instantaneous actions, i.e., with no duration. Once we get the classical plan, we can apply the durations of actions to that plan and get a **temporal plan**. This process is referred to as **Temporal Graph Compression Planning** and uses classical planning algorithms. Again, in practice, temporal plans are much more complex.

DURATIVE ACTIONS

Durative actions were introduced in **PDDL 2.1**.

A durative action introduces the concept of preconditions and effects at various times during an action:

- **`at start`**: at the beginning of the action
- **`overall`**: during the action, i.e. **invariant conditions**, which need to be true during the execution of the action
  - E.g.: while loading a package onto a truck, the truck must remain at the location throughout the execution of that action.
  - **Effects cannot have this specifier.**
- **`at end`**: at the end of the action



Consider the example given alongside. The following are the basic differences between a normal action and a durative action:

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

- In the definition of the action itself, there is the keyword **`durative-action`**, which tells the planner to expect a durative action.
- As this action is a durative action, it has a **`duration`**. In this case it is a simple fixed time unit. This duration can be a time unit range, or even dependent on other variables.
- The keyword `precondition`, is now **`condition`**
- The condition has the following **specifiers**:
  - **`overall`**: the truck must be at the location throughout the execution of the action
  - **`at start`**: the object (to be loaded) must be at the location at the beginning of the action
- The effect of the action also has some specifiers:
  - **`at start`**: the object is not at the location at the beginning of the action
  - **`at end`**: the object is in the truck at the end of the action

SELF-OVERLAPPING ACTIONS

Consider the example given alongside. The specifier for one of the effects has changed from `at start` to `at end`. This might seem logical but can cause problems with the semantics of the planning problem; once the action is applied, there is nothing stopping from the action to be reapplied during the execution of the action (as the object is still at the location, which is the precondition), or for some other action to change the location of the package. This could result in the package being in two different trucks. Thus, it is important to say that the package is not at the location anymore at the beginning of action.

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

Therefore **temporal planning is Exp-Space hard**, while **classical PDDL planning is P-Space hard**.

Example 2: To open the barrier, the person needs to be at the barrier at the start of the action. At the beginning of the action, the barrier is opened and at the end of the action, the barrier is closed. Thus, the only time the barrier is open, is during the execution of the action.

```
(:durative-action open-barrier
  :parameters
    (?loc - location ?p - person)
  :duration (= ?duration 1)
  :condition
    (and  (at start (at ?loc ?p)))
  :effect
    (and  (at start (barrier-open ?loc))
          (at end (not (barrier-open ?loc)))))
```

In this case, again, if the specifier for the effects is changed to the effects shown alongside, then the following problem would occur. The semantics of PDDL specify that **DELETE actions are applied before ADD actions**. Thus, at the end of the action, the barrier would be closed and then it would be opened. Thus, the effective effect of the action would be that the barrier is now open. If the semantics were reversed, then the barrier would never be open, and the planner would never be able to come up with a solution.

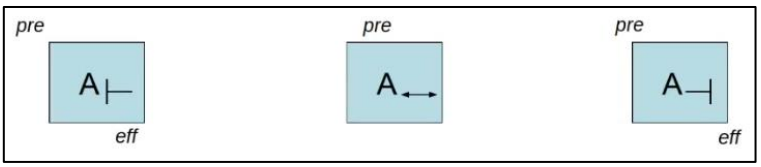
```
(at end (barrier-open ?loc))
(at end (not (barrier-open ?loc))))
```

TEMPORAL PLANNING CHALLENGES AND DECISION EPOCH PLANNING

DURATIVE ACTIONS IN LPGP (LINEAR PROGRAMMING AND GRAPH PLAN)

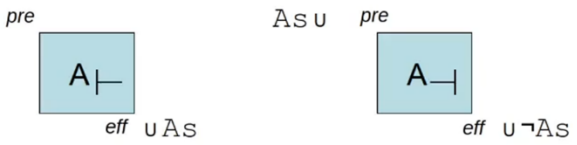
Instead of looking at the action as a single action, we can break it down into 3 parts since we have preconditions and effects at the start (`at start`) and end (`at end`), and we have preconditions during the execution of the action (`overall`), as follows:

- A_{start} ($A_{\vdash}$): all the start conditions and effects
- A_{inv} ($A_{\leftrightarrow}$): invariant action of A, which has all the invariant conditions
- A_{end} ($A_{\dashv}$): conditions and effects at the end



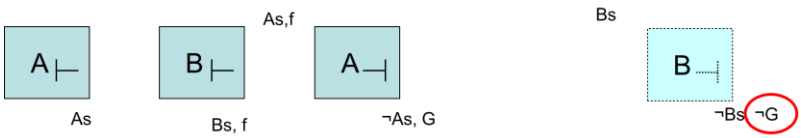
SNAP ACTIONS

Thus, we can treat $A_{\vdash}$ and $A_{\dashv}$ as two separate instantaneous actions in PDDL. We can do this by adding an effect for$A_{\vdash}$ (in this case `As`) and adding that as a precondition for $A_{\dashv}$. Once $A_{\dashv}$ is completed, we can delete `As`



PROBLEM OF PLANNING WITH SNAP ACTIONS

What if $B_{\dashv}$ interferes with the goal (shown alongside)? Since the goal needs to **persist** in a plan, this would create an invalid plan. Thus, no actions should be executing during a goal state, i.e., if you have reached a goal state, then all the actions must first finish executing before you can consider that state to be a goal state.

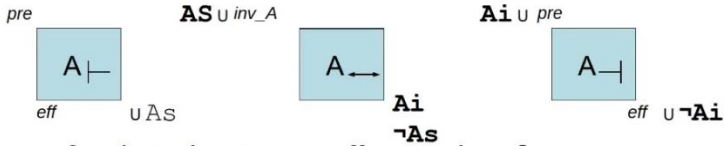


Solution: Add $\neg As, \neg Bs, \neg Cs$ to the goal (or make this implicit in a temporal planner), i.e., all the goals must be achieved, and no actions must be executing.

SNAP ACTIONS WITH INVARIANTS

We can apply the same logic to invariant actions as well.

- A_{start} ($A_{\vdash}$) adds a `As`
- A_{inv} ($A_{\leftrightarrow}$) requires `As` to be true. Once $A_{\leftrightarrow}$ has finished execution, it deletes `As` and adds `Ai`.
- A_{end} ($A_{\dashv}$) requires `Ai` to be true. Once it has finished execution, it deletes `Ai`.



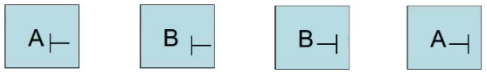
This **enforces the order $A_{\vdash}$, $A_{\leftrightarrow}$, $A_{\dashv}$** during the execution of the action. However this **does not enforce that other actions cannot take place between $A_{\vdash}$ and $A_{\dashv}$**. All it ensures is that $A_{\leftrightarrow}$ will take place at some point between $A_{\vdash}$ and $A_{\dashv}$

Solution: In every state where `As` is true `inv_A` must also be true. Thus violating an invariant then leads to a **dead-end**.

```
(imply (As) inv A)
```

ACTIONS WITH DIFFERENT DURATIONS

What if the duration of A is 10 and the duration of B is 5 but the planner comes up with the ordering shown alongside? Thus, even though the plan is **logically sound**, it is not **temporally sound**. So we need to ensure that all temporal constraints are met.



DECISION EPOCH PLANNING

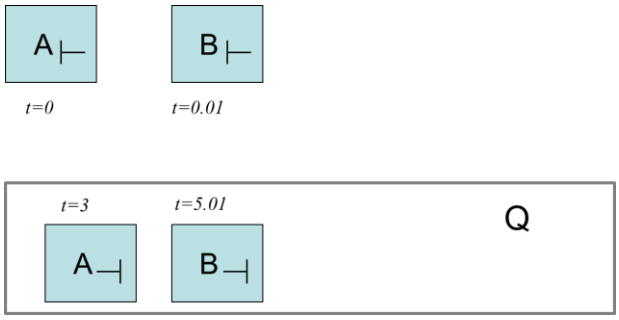
The idea is to search with **time-stamped states** and a **priority queue** of pending end snap-actions.

Planners: Sapa, Temporal Fast Downward

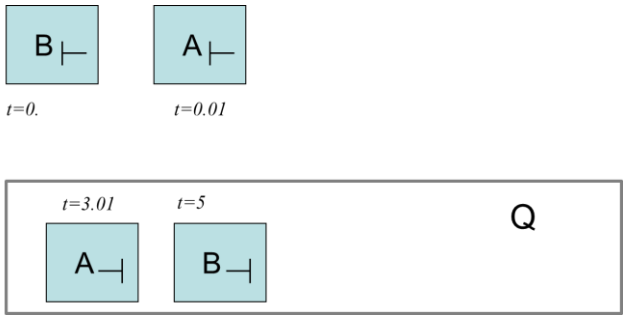
Thus, in a state S, at time t and queue Q, either

- Apply a **start snap-action** $A_{\vdash}$ (at time t)
  - Insert $A_{\dashv}$ into Q at time $(t + dur(A))$
    - Thus, we must know the duration of A
  - $S'.t = S.t + \varepsilon$
    - Time stamp of new state is equal to the timestamp of the old state plus ε, which is a very small number (usually 0.01) and can be parameterized
- Remove and apply the first **end snap-action** from Q
  - $S'.t$ set to the scheduled time of this, plus ε

Example: Initially we start of with no snap-actions and an empty queue. Then we can add $A_{\vdash}$ to the plan, thus we add $A_{\dashv}$ to the Q along with timestamp $t = 3$, since the duration of A is 3
Next, we can apply $B_{\vdash}$ at $t = 0.01$, as we have added ε to t. Since we have added $B_{\vdash}$, we add $B_{\dashv}$ to the Q with timestamp $t = 5.01$, since the duration of B is 5.
Thus, now we can either execute another action, or we can add $A_{\dashv}$ to the plan at $t = 3$. We cannot add $B_{\dashv}$, since the first thing in Q is $A_{\dashv}$. Once $A_{\dashv}$ has been added, then we can add $B_{\dashv}$ afterwards.



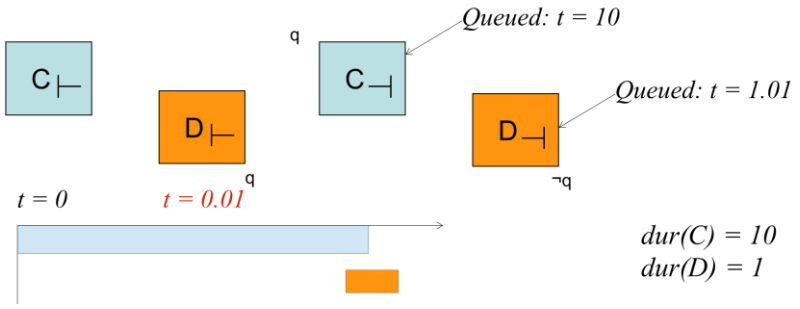
If we start the plan with $B_{\vdash}$, we can add $B_{\vdash}$ to the plan with timestep $t = 0$ and we add $B_{\dashv}$ to Q with timestamp since $t = 5$, since its duration is 5.
We can then add $A_{\vdash}$ at timestamp $t = 0.01$ and $A_{\dashv}$ with timestamp $t = 3.01$ since the duration is 3. Since 3.01 is smaller than 5, $A_{\dashv}$ will be placed in front of $B_{\dashv}$ in Q so that temporal consistency can be maintained.



LIMITATION OF EPOCH PLANNING

The restriction with epoch planning is that we **must fix start- and end- timestamps** at the point when the action is started as it is used for the priority queue.

Consider the example given alongside, this example is analogous the walking through the barrier example. C denotes the walking action and D denotes the open-barrier action. Thus, we would want to start the open barrier action right before the end of the end walk action so that we can walk through the barrier while it is open.



But the limitation of decision epoch planner is that an action can only start ε time after the previous action. Thus $C_{\vdash}$ is added at $t = 0$ and $D_{\vdash}$ is added at $t = 0.01$. But this will mean that $C_{\dashv}$ is queued to end at $t = 10$ and $D_{\dashv}$ is queued to end at $t = 1.01$.

A Decision Epoch planner can **only start actions ε time after the previous action**. Thus if an action needs to start at any point other than ε time after the previous action for a solution, then we can use decision epoch planners to solve the problem.

CRIKEY 3 AND SIMPLE TEMPORAL NETWORKS

CRIKEY 3 is a **forward search** planner that uses **simple temporal networks**.

SIMPLE TEMPORAL PROBLEM

All the constraints are of the following forms:

- **Sequential Constraints**: Timestamp of step $i + 1$ should come atleast ε time after timestamp of step i

$$\varepsilon \leq t_{i+1} - t_i$$

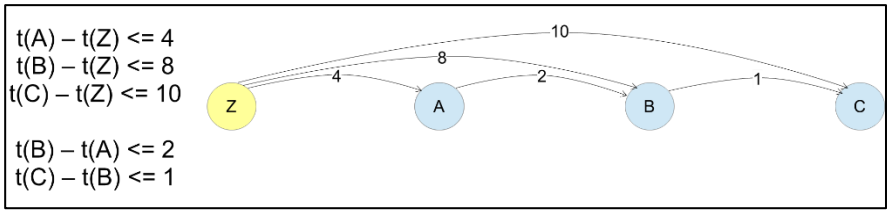
- **Duration Constraints**: the duration of an action will be between an upper and lower bound (each is optional)

$$dur_{min}(A) \leq t(A_{\dashv}) - t(A_{\vdash}) \leq dur_{max}(A)$$

This is a **Simple Temporal Problem**, which can be solved in **polynomial time**. The problem is that it is used during every state during search and is thus used many times.

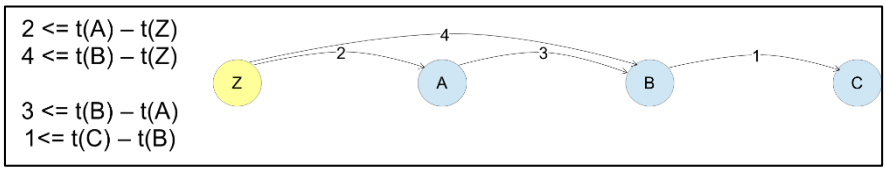
LATEST POSSIBLE TIMES

Let's say that we have these constraints on the problem, e.g., A can occur at most 4 timesteps after Z, or B can occur at most 2 timesteps after A. Thus, if we wanted to find the **latest time** to get to C, we can take two different routes, i.e., from $Z \rightarrow C$ (10), or $Z \rightarrow A \rightarrow B \rightarrow C$ (7). Looking at the **shortest path** in the graph will give us the **latest time** at which an action can occur.



EARLIEST POSSIBLE TIME

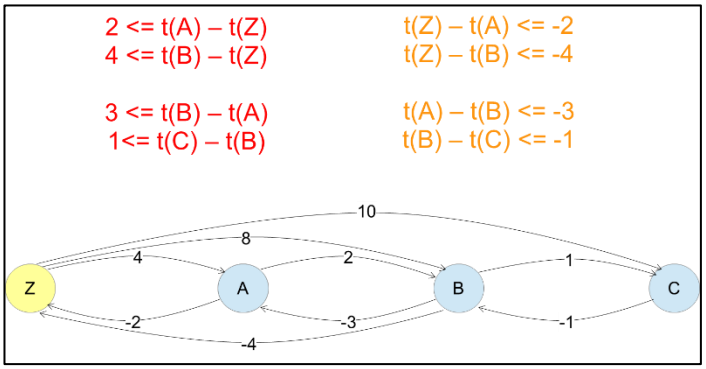
The **longest path** to an action, gives the **earliest timestep** at which the action can occur. Example: there are 2 paths to B; $Z \rightarrow A \rightarrow B$ (5) or $Z \rightarrow B$ (4). In this case, we will consider $Z \rightarrow A \rightarrow B$. Thus the earliest time B can occur is 5 timesteps after Z.



To compute the longest path, we can use an algorithm to find the shortest path by computing the shortest negative path from point A to point B.

PUTTING THEM TOGETHER

We can combine the 2 graphs to get the one shown alongside.



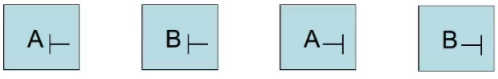
If we can see that the longest distance from X to 0 (i.e. the earliest possible time X can occur after 0) is greater than the shortest distance from 0 to X (i.e. the latest possible time X can occur after 0), then temporal constraints cannot be satisfied and there is no solution for the plan.

SIMPLE TEMPORAL NETWORK

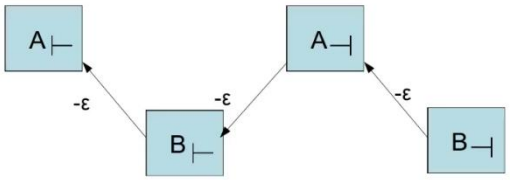
Thus, we can map simple temporal problems (STPs) to equivalent digraphs (shown above):

- One vertex per time-point (and one for time 0)
- For $lb \leq t(j) - t(i) \leq ub$
  - An edge $(i \rightarrow j)$ with weight ub
  - An edge $(j \rightarrow i)$ with weight $-lb$
- $lb \leq t(j) - t(i) \quad \rightarrow \quad t(j) - t(i) \leq -lb$

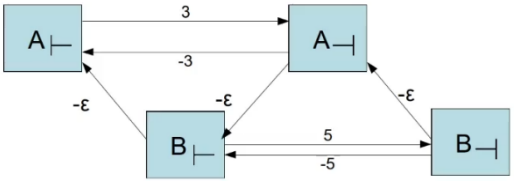
Example: Consider the example shown alongside of a planning problem with actions A and B. The planner has decided to apply the snap actions in the order shown alongside during forward search.



The first constraint we get is the **ordering constraint**, which says that each action must occur **at least** (**minimum separation**) ε time units after the previous action. Unlike decision epoch planning, STN planning doesn't enforce that each action must come exactly ε time after the previous action, but it says that it must come at least ε time after the previous action.



The second constraint is the **duration constraint**. In this case both actions A and B are of fixed durations, thus both their forward and backward edges have the same weight. If there was a duration inequality, then the weights would change accordingly.



If we run the plan with a shortest path algorithm, the plan will tell us that the earliest point A can start is 0 and the earliest point B can start is 0.01. Thus, the earliest time A can end is 3 and the earliest time B can end is 5.01.

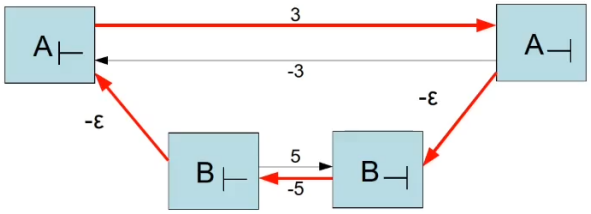
Thus, for the solution of the plan, we can solve the shortest path problem from/to zero:

- $dist(0, j) = x \rightarrow t_{max}(j) = x$
- $dist(j, 0) = y \rightarrow t_{min}(j) = -y$

Now we have a duration during which the action j can be applied, i.e., at any point between $t_{min}(j)$ and $t_{max}(j)$. Then we can simply select $t_{min}(j)$ as the time of execution for j in order to solve the plan in the shortest amount of time.

NEGATIVE CYCLES IN STNS

If the maximum time that can elapse between two timepoints (in this case 3), is less than the minimum time that can elapse between two timepoints (int this case $5 + 2\varepsilon$), i.e., there is a negative cycle, then time constraints are being violated and we cannot schedule this plan.

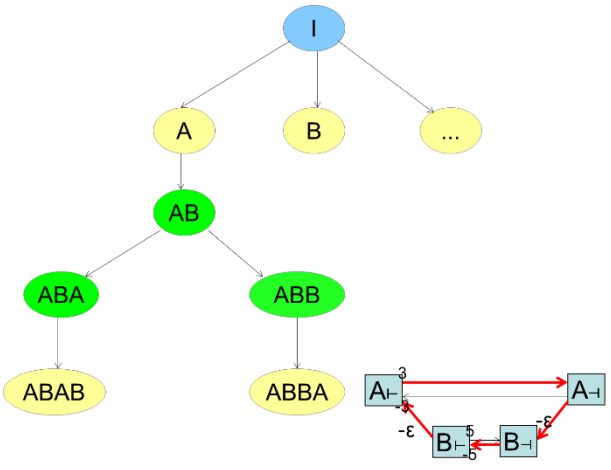


Thus, a shortest path algorithm needs to be used that can identify negative cycles (e.g., Bellman-Ford).

CRIKEY 3

It is a forward state space planner, which is based on FF style search, i.e., it performs Enforced Hill Climbing followed by Best First Search.

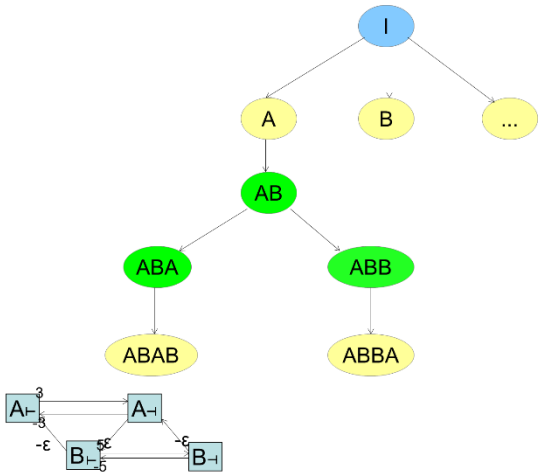
Example: We start of from the initial state and can apply any action available in the plan (or the end snap-action of an action that is already executing). We can keep selecting a nodes to expand (using a heuristic) and build the STN for each state. Of course, we can only select those actions whose preconditions are satisfied and those which are not deleting the invariants of any other actions.



As we can see alongside, the STN for the state $ABBA$ results in a negative cycle. Thus, we can prune this state and not expand it anymore as each expanded node of this state will also contain the negative cycle.

The same actions, but in a different order, can be seen elsewhere in the plan. In this case, the state $A_{\vdash}, B_{\vdash}, A_{\dashv}, B_{\dashv}$ is the final plan.

We can use a STN to ensure **temporal consistency**



MEMOISATION IN TEMPORAL PLANNING

Temporal planning ruins the closed list used in classical planning to reduce the state space (States already seen were discarded via the closed list). But with temporal planning the same facts can occur but at different times meaning it cannot be discarded.

THE TEMPORAL RELAXED PLANNING GRAPH HEURISTIC

With increasing complexity and expressiveness of planners, we need increasingly complex and expressive heuristics in order to guide search. Thus, in order to guide search for temporal planners, there are 3 options for heuristics:

1. Ignore time and use a **non-temporal heuristic**
    a. This was used in first version of CRIKEY (FF's RPG heuristic, with snap actions), but this resulted in the planner not telling us how long it will take to achieve goal, and favoring fewer longer actions rather than many shorter ones
2. Use a **temporal reachability analysis**
    a. Sapa and CRIKEY3 use an RPG with timestamped layers
3. **Approximate time** as action costs and use a **numeric heuristic**
    a. $Cost(A) = dur(A)$

TEMPORAL RPG

- Snap actions are used instead of actions.
- Fact and Action Layers are now timestamped
    - Fact and actions layers don't alternate as its possible for the start snap action to not have any effects
- All actions satisfied by fact layer 0 (initial state) appear in action layer 0
    - Facts from action layer 1 appear in layer 1.01 or $1 + \varepsilon$
- If $A_{\vdash}$ appeared at t, then $A_{\dashv}$ cannot appear in the RPG until layer $t + dur(A)$
- Preconditions need to be satisfied before $A_{\vdash}$ and $A_{\dashv}$ can be applied
- Also need to look at invariant conditions
- Since its an RPG, no delete effects will be considered

SOLUTION EXTRACTION

- If we add $A_{\vdash}$ to the Relaxed Plan we also add $A_{\dashv}$.
    - And achieve its preconditions.
    - Similarly, for adding $A_{\dashv}$ need $A_{\vdash}$ (most common case)
    - Since actions cannot be in the middle of execution during a goal state.
- Also need to achieve invariants in order to be able to apply $A_{\vdash}$.
    - No need to maintain them as this is relaxed planning.

PROPERTIES OF TRPG HEURISTIC

- The number of actions in the relaxed plan is the estimate of the distance to the goal, i.e., the heuristic value
- TRPG solutions will guide towards a plan which is shorter in time and not shorter in number of actions
- TRPG heuristic is **not admissible**, as there is no guarantee it is an underestimate.
- The layer at which a fact appears, is an admissible heuristic for when we can achieve that fact in the RPG
- The makespan of TRPG is optimal, i.e., the **makespan** of the TRPG is an **admissible** estimate of the makespan of a plan to solve the problem

THE POPF (PARTIAL ORDER PLANNING FORWARDS) PLANNER

In partial order planning, ordering constraints are only added to the plan when necessary, i.e., it's a least commitment approach. The problem with partial order planning is that it was performed in plan space and not state space, i.e., it considered partial plans. POPF aims to take the principals of partial order planning and apply them to forward search, as seen in CRIKEY 3.

- CRIKEY 3 = Forward Search + STN
  - Order each action after the previous action in the plan (ordering constraint)
- POPF:
  - Extends CRIKEY 3
  - Only put ordering constraints between actions when they are needed.

FORWARD CHAINING TEMPORAL PLANNING

A state S is a tuple $\langle F, V, P, T \rangle$, where

- $P \rightarrow$ propositional facts
- $V \rightarrow$ values of numeric task variables
- $P \rightarrow$ plan to reach S
  - Consists of the starts $A_{\vdash}$ and ends $A_{\dashv}$ of actions
- $T \rightarrow$ temporal constraints on the steps in P

TOTAL ORDERING IN SEARCH

Consider the scenario where there are 2 actions A and B, where B is longer than A. There is also no interaction between $A_{\vdash}$ and $B_{\vdash}$, but $B_{\dashv}$ must precede $A_{\dashv}$. In this case, the planner will choose the partial plan shown alongside. But there is no way that $A_{\dashv}$ can come after $B_{\dashv}$ since B is longer than A.

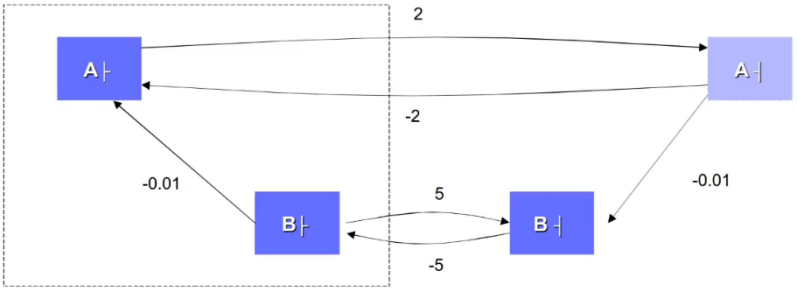
Thus, the aim is to identify these temporal inconsistencies and identify that in this scenario, $A_{\vdash}$ should be placed after $B_{\vdash}$, so that $B_{\dashv}$ can precede $A_{\dashv}$.

REDUCING COMMITMENT

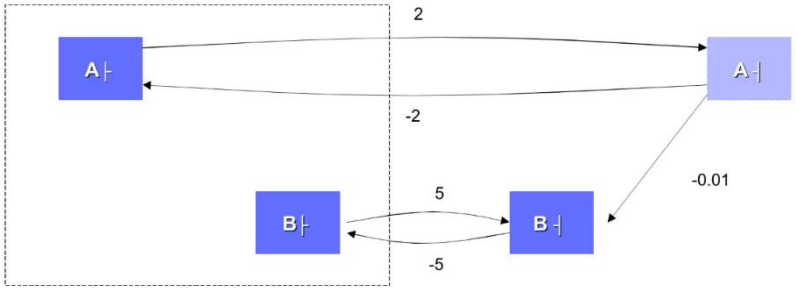
Looks to reduce commitment (from total to partial order) to allow for better ordering that is not just epsilon apart.

- Needs to now record additional information at each state concerning which steps achieve, delete, depend on each fact
- Because we are using forward chain planning, can still resolve threats, opting to always do promotion (order action after any actions whose preconditions threaten)
  - Traditionally you have demotion (ordering the action before) or promotion (ordering the action afterwards)

For example, we start off with $A_{\vdash}$. Then, when we add $B_{\vdash}$, we have this ordering constraint that $B_{\vdash}$ comes at least $\varepsilon = 0.01$ time units after $A_{\vdash}$.



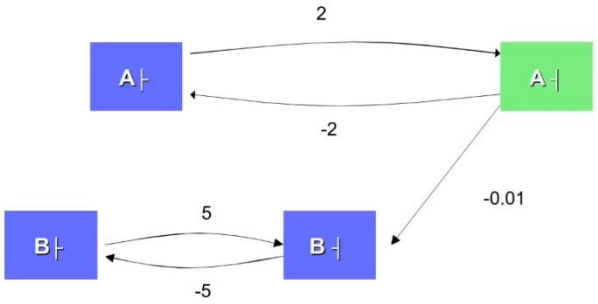
However, since we know that $B_{\vdash}$ does not interfere with $A_{\vdash}$ in anyways, i.e., $B_{\vdash}$ is not dependent on any effects of $A_{\vdash}$ and $B_{\vdash}$ does not delete any preconditions of $A_{\vdash}$, then we can simply remove the ordering constraint between $A_{\vdash}$ and $B_{\vdash}$. Thus, when we add $B_{\vdash}$, we can do so without any ordering constraints between $B_{\vdash}$ and $A_{\vdash}$.



When we continue to search and apply $A_{\dashv}$, the scheduler will simply pull the action $A_{\vdash}$ (as there are no negative cycles), and place $A_{\vdash}$ after $B_{\vdash}$.

Thus, the final plan will look like the following

- $t = 0.00 \rightarrow B_{\vdash}$
- $t = 3.01 \rightarrow A_{\vdash}$
- $t = 5.00 \rightarrow B_{\dashv}$
- $t = 5.01 \rightarrow A_{\dashv}$



EXTENDING THE STATE: PROPOSITIONAL

To capture ordering information POPF uses the following:

- F_+ and F_-, where $F_+(p)$ is the index of step that most recently added p, and $F_-(p)$ is the index of the step that most recently deleted p.
- **Active Conditions**: FP where $FP(p)$ is a set of pairs $\langle j, d \rangle$
  - **Instantaneous Conditions**: start or end conditions on snap-actions
    - $\langle j, \varepsilon \rangle$ denotes that step j has an instantaneous condition on p (at start or at end)
  - **Invariant Conditions**: Must hold true if an action has started in the plan but not yet finished.
    - $\langle j, 0 \rangle$ denotes that step j marks the end of an action with an overall condition on p

Actions can achieve and delete their own invariants. Thus, we only need to maintain the invariant until exactly when the action ends, rather than ε after the action ends. Therefore there is a difference between the instantaneous conditions and the invariant conditions.

Example: The Light Match PDDL given alongside demonstrates this. The invariant condition of the action is that the match must be lit. This condition is added by the start effect of the actions and deleted by the end effect of the action.

```
:durative-action LIGHT_MATCH
:parameters (?m - match)
:duration (= ?duration 8)
:condition
   (and (at start (unused ?m))
   (over all (light ?m)))
:effect (and
   (at start (not (unused ?m)))
   (at start (light ?m))
   (at end (not (light ?m))))
```

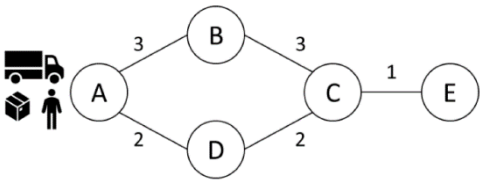
For each `at start` condition p:

- $t(F_+(p)) + \varepsilon \leq t(i)$
  - This action comes after the most recent action in the plan to add p $(+\varepsilon)$

For each `over all` condition p:

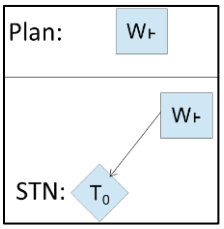
- $if\ F_+(p) \neq i, t\big(F_+(p)\big) \leq t(i)$
  - No ε here as it is not needed, as action can achieve invariant condition at the same time as start
  - If action adds itself, we do not need to worry about ordering, hence the check $\neq i$

Example: **Driver Log Shift domain**. The goal is to deliver packages from A to E.

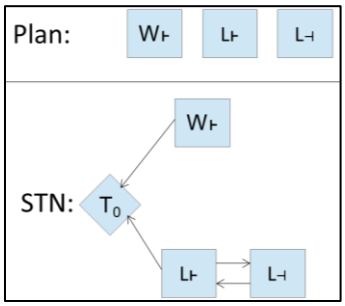


- Driver works a 6 hour shift
- Durative action `work`
  - Duration 6
  - Adds (`working driver`) at start, and deletes it at end
- `Drive`, `Board` and `Alight` actions have `over all` condition: (`working driver`)
- `Load` and `Unload` do not, i.e., do not require the driver to be working

The first action to be added to the plan is $\boldsymbol{W}_\vdash$ $(Work_{start})$. There are no dependencies to consider, since it is the first action to be added to the plan, thus we can order it after $t = 0$.

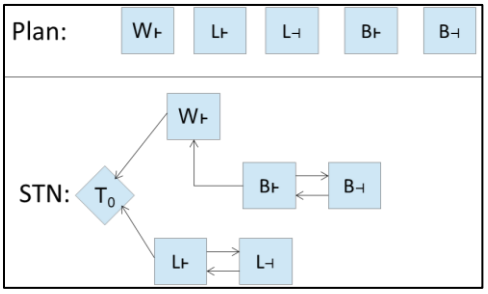


Next, let's say we add the action $\boldsymbol{L}_\vdash$ $(Load_{start})$. In CRIKEY 3, we would order $L_\vdash$, ε after $W_\vdash$, since it was added to the plan after $W_\vdash$. But since $L_\vdash$ does not rely on the effects of $W_\vdash$ and it doesn't delete any preconditions of $W_\vdash$, it can simply be ordered after $t = 0$.



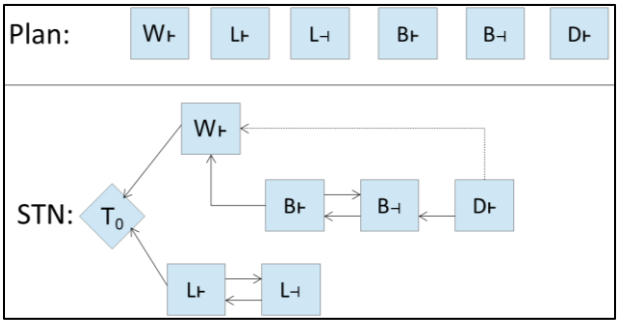
We can now add the action $\boldsymbol{L}_\dashv$ $(Load_{end})$. This will occur after $L_\vdash$, as it is the corresponding end snap action. The constraints between them are the duration constraints (not shown in the image). Again, since $L_\dashv$ does not interfere with $W_\vdash$ in anyway, it is not going to be ordered with respect to $W_\vdash$.

Let's say that the next action we add is $\boldsymbol{B}_\vdash$ $(Board_{start})$. This action is dependent on the effect of $W_\vdash$, i.e., the driver is working. Thus we need to add it after the last action that added its precondition, i.e., after $W_\vdash$.

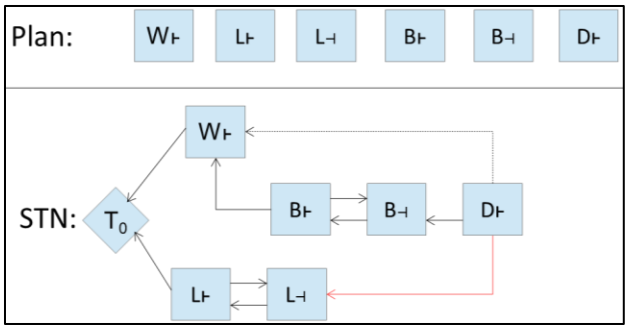


We can now add the action $\boldsymbol{B}_\dashv$ $(Board_{end})$ after $B_\vdash$, with duration constrains between the two of them as shown alongside.

Now if we add the action $D_\vdash$ $(Drive_{start})$, we can see that the it has a precondition that the driver must be in the truck, which was added by the action `board`. Thus, need to order $D_\vdash$ after the last action that added its precondition, i.e., $B_\dashv$. But $D_\vdash$ has an additional precondition that the driver needs to be working, which was added by $W_\vdash$. Thus, $D_\vdash$ has an additional ordering constraint, that $D_\vdash$ comes after $W_\vdash$. This is the difference between COLIN and POPF.



In this case, the edge is a dotted line, since $D_{\vdash}$ is ordered to come after board, which is ordered after $W_{\vdash}$, so $D_{\vdash}$ transitively implied to be ordered after $W_{\vdash}$ without having to add another constraint. POPF adds **transitively implied edges** anyways, to improve efficiency.



The **problem** here is that $D_{\vdash}$ is not ensured to come after $L_{\dashv}$ and $D_{\vdash}$ deletes an invariant of load, i.e., the truck remains at the location throughout. While adding actions, we need to ensure that we are not breaking the preconditions or invariants of preexisting actions. Thus we need to add an ordering constraint, that $D_{\vdash}$, comes after $L_{\dashv}$.

AT START: DELETE EFFECTS

For each `at start` delete effect p, it must come after every step with a precondition in p.

- $\forall \langle j, d \rangle \in FP(p), t(j) + d \leq t(i)$
  - Any action with a precondition or invariant on p must come before step i that we are adding to the plan.
  - d is either 0 or ε depending on if its an invariant constraint or precondition

We also need to make sure that before deleting an effect we are ordered after the last action to add that effect because we want to know in the current state if the fact is true or false.

$$t\big(F_{+}(p)\big) + \varepsilon \leq t(i)$$

- Then update $F_{-}(p) = i$ to set the new action as the most recent delete effect of p
- Clear $FP(p)$ as we have ordered after all actions in the list and have deleted p

AT START: ADD EFFECTS

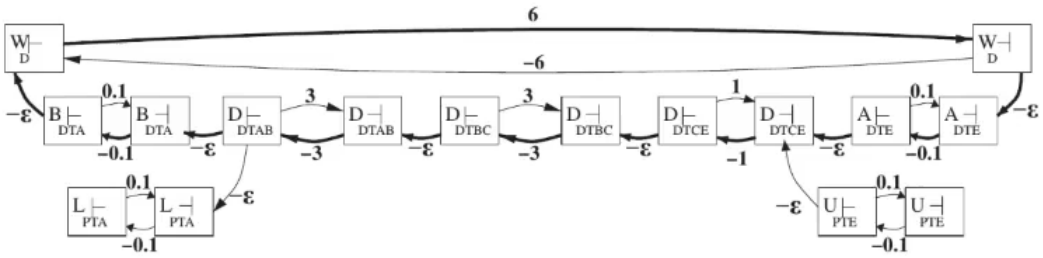
For each `at start` add effect p:

- $if\ F_{-}(p) \neq i, t\big(F_{-}(p)\big) + \varepsilon \leq t(i)$
  - A check to see if we are the last action to delete p (invariant). If so, then this is not applicable. Otherwise, make sure we come ε time units after the last action to delete p.
- Then update $F_{+}(p) = i$, as we are now the last action to add p.

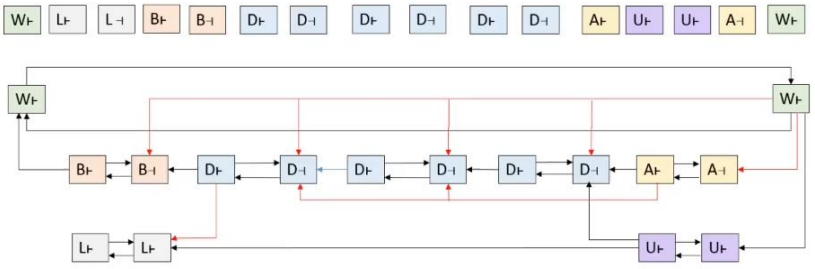
POPF STN FOR THIS PROBLEM

Transitively implied edges are omitted.

All the actions which are not dependent on the driver working are not ordered with respect to `work`, and all the actions dependent on the driver working are order with respect to `work`.



The reason why this is useful is because this single STN, found through a single search, represents a lot of plans. A small selection of possible plans is shown below.



This is so because the `load` and `board` actions can be ordered in different orders, since there are no ordering constraints between them. Similarly, the `unload` and `alight` actions can take place independently any many orders as they have no ordering constraints among them.

The `work` and `load` (and `unload`) actions can also be permuted as there is no ordering constraint between them.

COMPRESSION SAFETY

By splitting actions into instantaneous start and end actions we make the search space **at least twice as big**, as instead of just having to decide when to start an action, now we also have to decide when to end it.

There are different motivations for calling actions; some are called for their end effects and some for their durative effects. Once $C_{\vdash}$ has been applied in a plan, $C_{\dashv}$ will be considered for application in **every state** in which its preconditions are satisfied. If the end of the action does nothing bad, we may as well end it straight away.

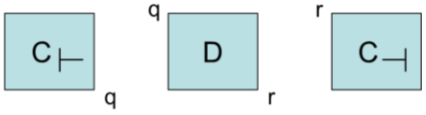
RULES FOR COMPRESSION SAFETY

The End-Effect Rule

- There must be no end-delete effects
- If there are end-delete effects, they might make preconditions of other actions false, e.g., match domain or barrier domain
    - If we immediately add $C_{\dashv}$ after $C_{\vdash}$ in this case, then we might not be able to use a fact that C added, e.g., the match domain.

The End-Precondition Rule

- If an action has end-preconditions that are not invariant conditions, it cannot be ended immediately; maybe some other actions needs to be called first.
    - Example: $C_{\vdash}$ requires p and adds q. D requires q and adds r. $C_{\dashv}$ requires r. Thus, in this case C cannot be ended immediately, as $pre(C_{\dashv})$ might not be satisfied.



- Invariant conditions do not need to be considered because they have already been satisfied for $C_{\vdash}$, and no action can be applied that violates them
- In general once the start of an action $C_{\vdash}$ has been applied, invariants of C will be maintained until $C_{\dashv}$ is applied (no action may be applied that violates them). Thus, if the preconditions of $C_{\dashv}$ are a subset of preconditions of $C_{\leftrightarrow}$, i.e., $pre(C_{\dashv}) \subseteq pre(C_{\leftrightarrow})$, then $C_{\dashv}$ is always applicable so we don't need to worry about some other action deleting the preconditions of $C_{\dashv}$.

COMPRESSION SAFE ACTIONS

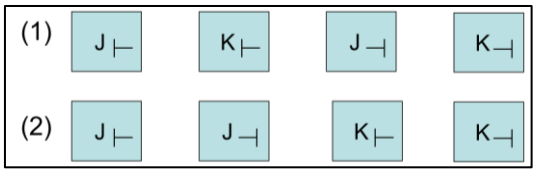
By combining the above concepts, we can say that an action is compression safe if,

- **It has no end delete effects, and**
- **no end preconditions that are not invariant conditions**.

In general, once the start of an action ($C_{\vdash}$) has been applied, its end ($C_{\dashv}$) will be considered in every possible state (in which its end preconditions are satisfied) until it has been applied. But if the action is compression safe, we can just add both the start and end of the action to the plan at the same time.

Thus, if an action is compression safe, can prune the search space by applying the end action immediately after the start action.

Consider the actions J and K, which are independent of each other. The most efficient plan for executing the actions is shown alongside (1). But if we say that they are compression safe, i.e., both the actions can end immediately after starting, then they will have a different ordering (2).



In such cases where there is **compression safe overlapping**, POPF can still achieve the efficient plan due to its partial order nature; only future actions that **use** or **delete** p **must come after** the action, making overlapping still possible.

PLANNING WITH DEADLINES

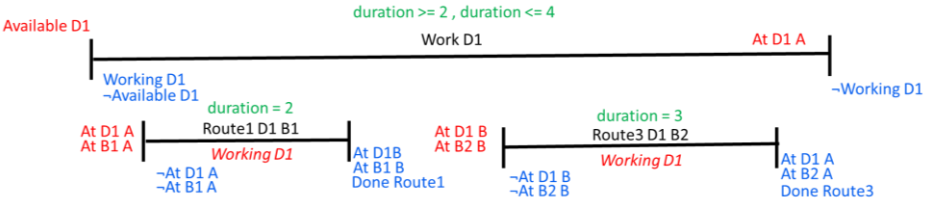
Examples of domains with deadlines:

- Driver log shift – packages need to be delivered before end of shift
- Rovers – rover needs to complete activities before EOD so that it can go into safe mode.
- **Public Transport Domain**
  - Drivers have working hours
  - Bus routes have fixed durations and start and end locations
  - Goal is that all bus routes need to be completed
  - Routes have timetables that must be followed

Consider the public transport example shown alongside.

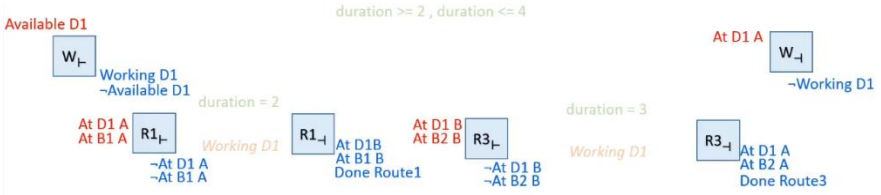
Actions have the following

- Conditions and Effects at the start and at the end
- Invariant conditions
- Duration constraints
  - Fixed
  - Inequality



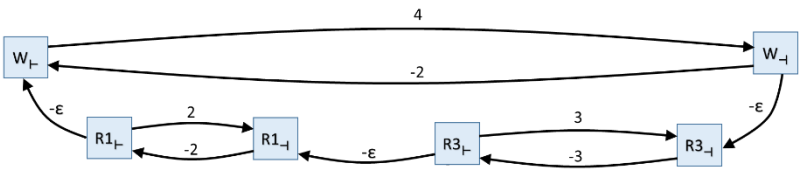
CRIKEY/CRIKEY3 APPROACH

Split the actions into start- and end-snap actions. Tackle following challenges:



- Invariant conditions are not deleted by other actions during execution
- Make sure end snap actions can't be applied unless start snap actions have been applied.
- Duration constraints – build STN to check plan consistency

Given alongside is the STN that would be built by CRIKEY3 for this plan.



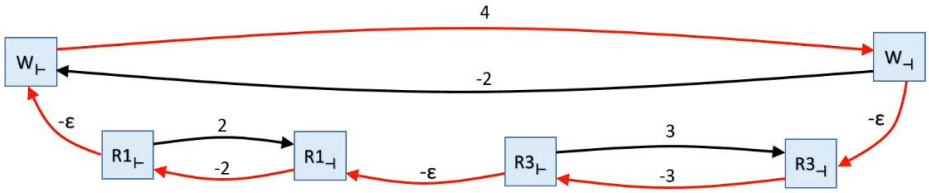
In this case, since all the actions depend on a previous step in the plan, this is also the STN that **POPF** will generate for the plan.

The STN depicts the following Ordering Constraints:

- $R1_{\vdash} \geq W_{\vdash} + \varepsilon$
- $R3_{\vdash} \geq R1_{\vdash} + \varepsilon$
- $W_{\dashv} \geq R3_{\dashv} + \varepsilon$

and the following Duration Constraints:

- $W_{\dashv} - W_{\vdash} \geq 2$ (lower bound = backward arrow),
- $W_{\dashv} - W_{\vdash} \leq 4$ (upper bound = forward arrow),
- $R1_{\dashv} - R1_{\vdash} = 2$,
- $R3_{\dashv} - R3_{\vdash} = 3$



But CRIKEY3 won't be able to solve the problem since there is a negative cycle, i.e., the driver is only working for 4 hours, but is driving for $5 + 3\varepsilon$ hours

TIMED INITIAL LITERALS

- Introduced in PDDL 2.2 (IP 2004)
- Allow us to model facts that become true or false at specified time
- Can be used to model deadlines or time windows
    - Cannot be done directly, but can be achieved by adding more facts to the domain
        - Can be used to both **delete facts** (used for deadlines) and **add facts** at certain times

MODELLING DEADLINES USING TIMED INITIAL LITERALS

Consider the example of the driver log domain given alongside. The timed initial literals are included in the `init` (example given alongside is a small part of the entire problem) part of the problem file.

At time `9`, `can-deliver package1` will be deleted and at time `11`, `can-deliver package2` will be deleted. Both were initially true.

We are trying to make sure that `package1` is delivered before $t = 9$ and that `package2` is delivered before $t = 11$.

```
(:durative-action unload-truck
  :parameters (?p - obj ?t- truck ?l- location)
  :duration (= ?duration 2)
  :condition  (and
    (over all (at ?t ?l))
    (at start (in ?p ?t))
    (at end (can-deliver ?p)))
  :effect   (and
    (at start (not (in ?p ?t)))
    (at end (at ?p ?l)))
)
init:
  (can-deliver package1)
  (at 9 (not (can-deliver package1)))
  (can-deliver package2)
  (at 11 (not (can-deliver package2)))
```

It is important to note that the `(can-deliver ?p)` precondition is a `at end` condition and not a `at start` precondition. This is because, if the fact only needed to be true at the beginning of the action, then the `package1` could be delivered at $t = 11$, since the duration of `unload-truck` is 2. Thus, the plan would miss the deadline.

- An `at start` condition, with an adjusted deadline (considering the duration of the action) could also be used, but this causes problems when dealing with actions with varying durations. Thus, its best to use deadlines with `at end` preconditions.
- An `over all` condition could also be used, but the difference in the deadline in that case would be of ε due to the semantics of these specifiers.

MODELLING TIME WINDOWS USING TIMED INITIAL LITERALS

Consider the public transport domain example given alongside. In this case we have used TIILs to create a time window using the variable `route3`. Thus, the earliest the driver can arrive at `route3` is at $t = 3.75$, and the latest is at $t = 4$. This forces the planner to end the action within the time window.

```
(: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)))
  :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))
)
init:
  (at 3.75 (due route3))
  (at 4 (not (due route3)))
```

It is important to note that `due route3` is not true in the initial state.

Other uses of time windows with TILs:

- If we want to say that the action can only start during a certain time window, then we can add a `at start` precondition instead of the `at end` shown alongside.
- If we wanted to say that the entire action should fit within a time window, then we can use an invariant condition.
- Time windows **can be recurring**, i.e., we can create another time window at a later stage which will give the planner multiple opportunities for the execution of an action.

REASONING WITH TILS IN FORWARD SEARCH

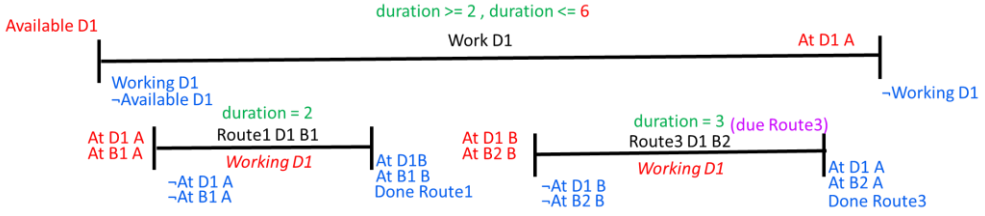
- Order the TILs chronologically.
  - This is possible as each TIL gives us a fixed time at which a literal becomes either True or False.
- At each state we have an extra choice:
  - Apply an action that is applicable in that state (as always), or
  - Apply the next available TIL
    - i.e., the next one which is going to happen in the chronological order

This allows us to leave the choice to search about whether the TIL will appear before or after a give action. This gives the planner a choice of ordering between TILs and actions.

POPF has an advantage over CRIKEY3:

- If CRIKEY3 chooses to apply a TIL, then every action it chooses to apply to the plan will be ordered after the TIL.
- If POPF chooses to apply a TIL, and then apply an action which does not depend on the TIL (i.e., it does not use the TIL as a precondition and the TIL does not delete its preconditions), then that action will not be ordered with respect to the TIL as that ordering constraint is not necessary and POPF only enforces necessary orderings. This gives more flexibility in planning.

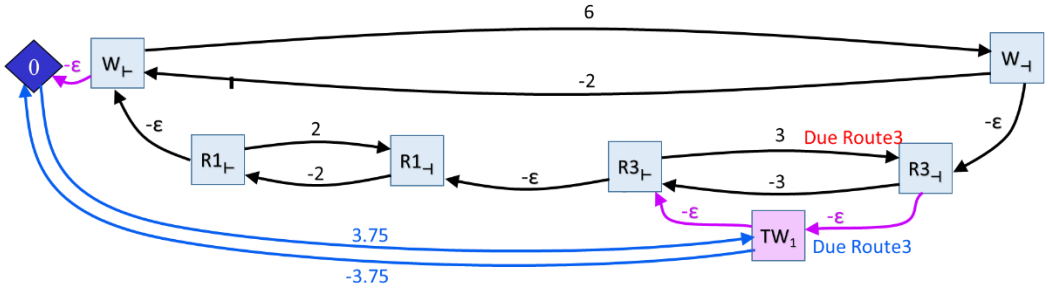
Consider the public transport domain problem used earlier. This time the duration constraint on the work action has been changed to remove the negative cycle.



As mentioned earlier, CRIKEY3 will add ordering constraints with the time window as shown below. We also have to add T_0, i.e., $t = 0$ to our plan when using timed initial literals. In this case, since $R3_\vdash$ was added before TW_1, and $R3_\dashv$ was added after TW_1, there is an ordering constraint between them.

Constraints:

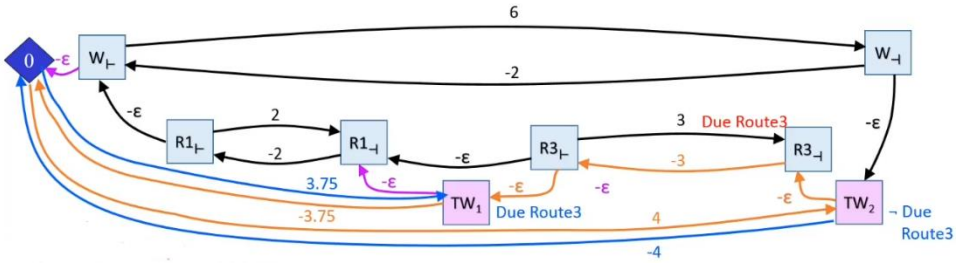
$W_\dashv - W_\vdash >= 2$ $W_\vdash >= T_0 + \varepsilon$
$W_\dashv - W_\vdash <= 6$ $TW_1 = T_0 + 3.75$
$R1_\vdash >= W_\vdash + \varepsilon$ $TW1_1 >= R3_\vdash + \varepsilon$
$R1_\dashv - R1_\vdash = 2$ $R3_\dashv >= TW_1 + \varepsilon$
$R3_\vdash >= R1_\vdash + \varepsilon$
$R3_\dashv - R3_\vdash = 3$
$W_\dashv >= R3_\dashv + \varepsilon$



Consider a different scenario; what if CRIKEY 3 applied TW_1 before $R3_\vdash$ in its total ordering? In this case we will get the following ordering constraints. In this case when we add TW_2 to the plan, it creates a negative cycle, thus this plan is not valid.

Constraints:

$W_\dashv - W_\vdash >= 2$ $W_\vdash >= T_0 + \varepsilon$
$W_\dashv - W_\vdash <= 6$ $TW_1 = T_0 + 3.75$ $TW_2 = T_0 + 4$
$R1_\vdash >= W_\vdash + \varepsilon$ $TW_1 >= R1_\dashv + \varepsilon$ $TW_2 >= R3_\dashv +$
$R1_\dashv - R1_\vdash = 2$ $R3_\vdash >= TW_1 + \varepsilon$ $W_\dashv >= TW_1$
$R3_\vdash >= R1_\vdash + \varepsilon$
$R3_\dashv - R3_\vdash = 3$



In the case of POPF the ordering constraint between $R3_\vdash$ and TW_1 would not be added as $R3_\vdash$ does not depend on `due Route3`. This will break the negative cycle, and POPF will be able to generate a valid plan, as it will be able to start $R3_\vdash$ before TW_1.

So CRIKEY3 will still be able to solve the order, it will have to explore more spaces. Thus POPF's least commitment strategy can help solve plans with deadlines quicker.

PLANNING WITH CONTINUOUS LINEAR CHANGE: COLIN

COLIN – **Co**ntinuous **Lin**ear Change

Planning with Continuous Linear Change – planning with numeric quantities that do not change instantaneously but change continuously with respect to time. Numeric values often change continuously, rather than discretely. Example:

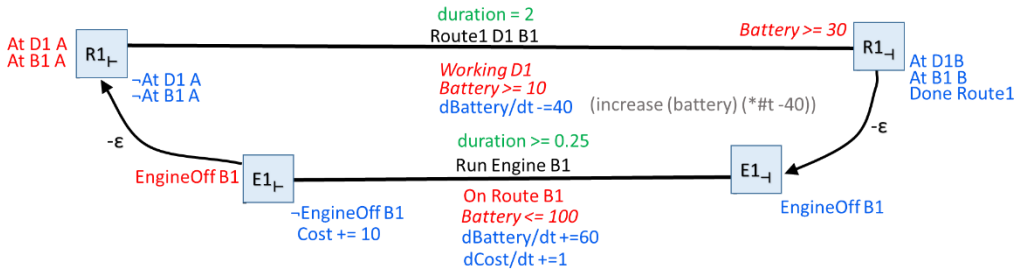
- **Instantaneous Change**
  - `(at end (decrease (battery) 1))`
  - `(at end (decrease (battery) (+ (* 0.5 (temperature)) (* 3 (?duration)))))`
- **Continuous Linear Change**
  - While bus is running $\frac{dbattery}{dt} = -40$
  - `(increase (battery) (* #t -40))` or `(decrease (battery) (* #t 40))`
    - Where #t represents the derivative of the effected variable (i.e., battery) with respect to time
  - But change is not always linear

At an action start ($A_\vdash$) a numeric value will be set. But instantaneously after that action ($A_\vdash'$) a continuous value could affect the numeric value so it will be different. To calculate the change of a value from one point to another ($A_\vdash \rightarrow B_\vdash$) we use the formula $\boldsymbol{V'} = (\boldsymbol{W} \times \boldsymbol{V}) + \boldsymbol{C}$.

In the example of a battery being decreased by 40 every time unit we have $B_{\vdash} = A_{\vdash}' + (B_{\vdash} - A_{\vdash}) \times -40$, where $(B_{\vdash} - A_{\vdash})$ is the time difference between the two states.

When adding constraints (E.g., battery cannot drop below 10 units) we add these immediately after the starting point $(A_{\vdash}', A_{\dashv}', B_{\vdash}', B_{\dashv}')$. And as a condition for all future actions X and X'.

Consider the example given alongside. The battery of the bus is decreasing continuously by 40. In order to charge the battery, we can turn the engine on. By doing so, the battery is increasing continuously by 60, but the cost is also increasing continuously by 1.

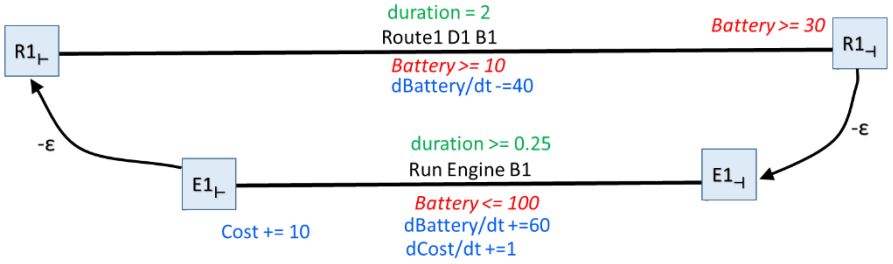


In order to model this problem in **COLIN**, we perform the following steps

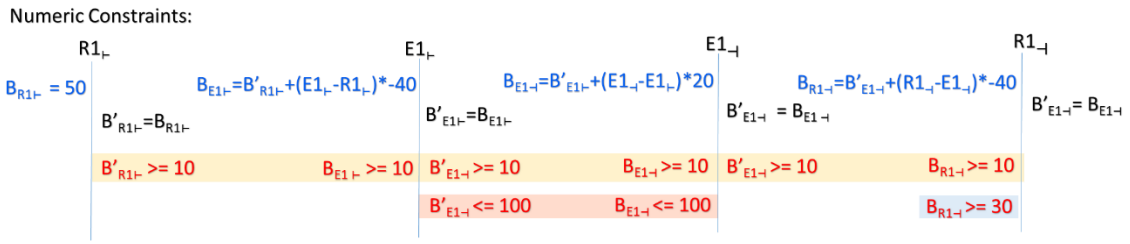
1. **Model temporal and numeric constraints as sets of linear equations and inequalities**.
2. **Solve the linear equations and inequalities**

Thus in this problem, we have the following constraints

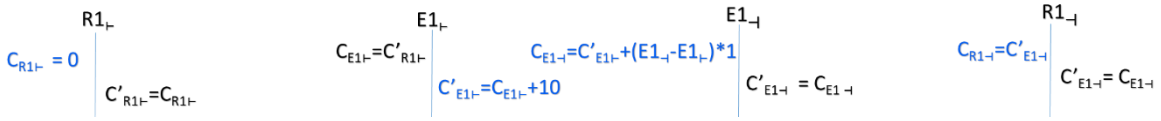
- **Temporal Constraints**
  - $R1_{\dashv} - R1_{\vdash} = 2$
  - $E1_{\dashv} - E1_{\vdash} \geq 0.25$
- **Ordering Constraints**
  - $E1_{\vdash} \geq R1_{\vdash} + \varepsilon$
  - $R1_{\dashv} \geq E1_{\dashv} + \varepsilon$
- **Numeric Constraints**:
  - These constrains are ordered by index of the action of the plan the effect correspond to (shown below)



First, we consider the constraints for the battery. In the initial state, the battery is 50. This is shown alongside. The values change from A to A' after an action, and A' in each case can be calculated as shown alongside. The numeric constraints are shown in the 3rd and 4th row of the diagram. All of these are **invariant numeric constraints** except for $B_{R1_{\dashv}} \geq 30$, which is an **instantaneous numeric constraint**.



The encoding of the cost is independent of the encoding of the battery. The difference here is the discrete increase in cost after $E1_{\vdash}$, i.e., $C'_{E1_{\vdash}} = C_{E1_{\vdash}} + 10$.



Thus, we now have a complete set of equations and inequalities, the solution of which will give us a scheduled plan.

LINEAR PROGRAMMING

The following is the **general form** of linear programming:

- Objective Function → maximize (or minimize): $z = w_z \cdot v$
  - Example: minimize cost
- Subject to a set of constraints:
  - $w_1 \cdot v \{\leq, \geq, =\} c_1$
  - $w_2 \cdot v \{\leq, \geq, =\} c_2$
  - …
  - $w_n \cdot v \{\leq, \geq, =\} c_n$

These equations can be fed into existing linear programming solvers in order to get an assignment for the variables.

COLIN: GENERAL APPROACH

For each (snap) action A_i in the (partial) plan create the following **linear programming (LP) variables** in the problem:

- $\boldsymbol{v_i}$: the value of v immediately before A_i is executed
- $\boldsymbol{v_i'}$: the value of v immediately after A_i is executed
- $\boldsymbol{\delta v_i}$: the rate of change active on v after A_i is executed

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

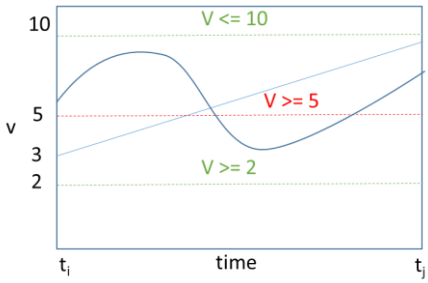
Constraints:

- Initial Values
  - $\boldsymbol{v_0}$: initial state value of v
- Temporal Constraints
  - $t_i \geq t_{i-1} + \varepsilon$
  - $t_j - t_i \leq \text{max\_dur(A)}$, where t_j is the end of the action starting at t_i
  - $t_j - t_i \geq \text{min\_dur(A)}$, where t_j is the end of the action starting at t_i
- Continuous Change
  - $v_{i+1} = v_i' + \delta v_i (t_{i+1} - t_i)$
- Discrete Change
  - $v_i' = v_i + w \cdot v_i$
    - Example: $v_i' = v_i + 2u_i - 3w_i$ (Can be dependent on other variables as well)
- Preconditions: Constraints over v_i:
  - $w \cdot v_i \{\geq, =, \leq\} c$
    - Example: $2w_i - 3u_i \leq 4$
- Invariants of A must be checked before and after every step between the start t_i and end t_j of A:
  - $w_1 \cdot v_i' \{\leq, \geq, =\} c_1$
  - $w_2 \cdot v_{i+1} \{\leq, \geq, =\} c_2$
  - $w_2 \cdot v_{i+1}' \{\leq, \geq, =\} c_2$
  - …
  - $w_n \cdot v_j \{\leq, \geq, =\} c_n$

$\boldsymbol{\delta v_i}$ is a constant that we can calculate whilst making the LP by looking at the continuous numeric effects of actions:

- All in the form $(\frac{dv}{dt} \pm c)$ or $\delta v_i += c$ or $\delta v_i -= c$
- What if δv_i was a function of the variables: e.g., $\delta v_i = 2w - u$?
- $v_{i+1} = v_i' + \delta v_i \, (t_{i+1} - t_i)$

Invariant Checking



- We only check the condition at the start and end of each interval (i.e., after one action is applied, and before the next is).
- Checking becomes very hard with continuous values as it is not enough to check beginning and end anymore as it could be violated at any point in time (consider the wavy line in the graph alongside).

Thus linearity guarantees that

- if we check invariant conditions at snap actions points (specific time when the action is applied in the plan), we are guaranteed correctness
- the equations we write to calculate the new values of v at new action steps are linear equations, and thus we can use an LP solver

Objective Function

- LPs have an objective function, with an aim to minimize (or maximize) a value
  - Example: Aim is to minimize makespan
    - Make a variable t_{now} and order it after all other steps in the plan:
      - $t_{now} - t_i \geq \varepsilon$
    - Now set the LP objective to minimize t_{now}
  - Example: aim is to minimize some cost function other than makespan
    - Write the objective as a function of t_{now} for the final action in the plan so far
    - E.g.: minimize $3v_{now} + 2w_{now} - u_{now}$
- LP solvers optimize function (e.g., minimize cost). LP will **find an optimal schedule** (optimal set of timestamps) for this plan but will **not create an optimal plan**. Thus, this is a way to optimize the objective. The only way to get an optimal plan is by exhausting other state spaces.

Action Applicability

- In general in discrete numeric planning, we know the values of the variables:
  - Value in the initial state specified:
  - Effects update value by a known amount: $v = v + 2u - 3w$
  - Compute the new value in the current state and check whether preconditions are satisfied.
- What if there is a continuous numeric change active in a state?
  - The value of the A variables now depend on how much time we allow to elapse
  - In our example, if we start the route the value of the battery is: $50 - 40 \times time_{elapsed}$
  - Since we don't know how much time has passed, we don't know the exact value of the battery, but we know a rough range, i.e., $[50,0]$ (can't be higher than 50 since it started at 50 and its decreasing, can't be lower than 0 since there is a constraint on the lower bound) depending on what time we apply the next action.

USING THE LP TO FIND BOUNDS

In general, its not easy once a lot of change has happened to know the bounds on a given variable in a state. We can however, use the same LP to calculate this with a small modification:

- A variable t_{now} represents 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 set the objective function to:

- **Maximize v_{now}** to give the **upper bound on v**
- **Minimize v_{now}** to give the **lower bound on v**

Now take the upper or lower bound to satisfy all $\geq$ or $\leq$ conditions respectively.

ISNT SOLVING LPS EXPENSIVE?

No. Most algorithms are polynomial (Simplex isnt, but its efficient). These LPs are easy ones.

- Heuristic computation is notoriously expensive
  - An analysis showed that FF spends ~80% of its time evaluating the heuristic
- COLIN:
  - Empirically uses an STP scheduler scheduling accounts for an average less than 5% of state evaluation time
  - For CLP and CPLEX (LP solvers), the figures are 13% and 18% respectively.
  - So it is better than calculating the heuristic.

OPTIC: COMBINING TEMPORAL PLANNING AND PREFERENCES

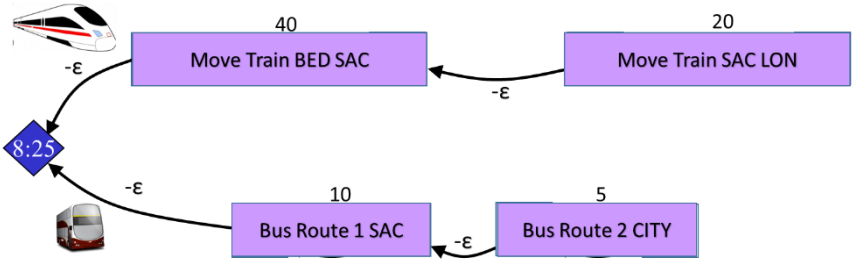
RELATIONSHIPS BETWEEN PLANNERS

The following planners are all based on the same code base and are an extension of the previous one (FF was originally a STRIPS planner):

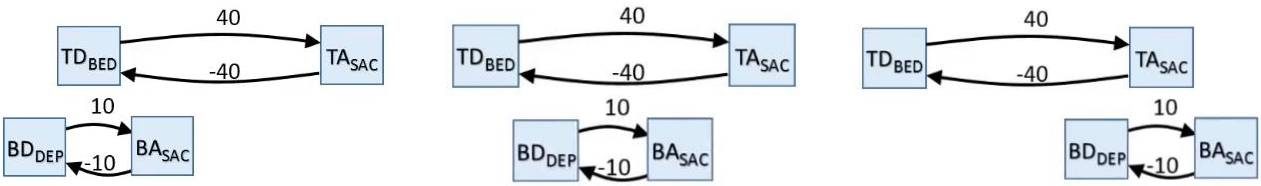
- CRIKEY → FF + STN
- COLIN → CRIKEY s/STN/LP/;
- POPF → COLIN + fewer ordering constraints
- OPTIC → POPF + Preferences.

PARTIAL ORDER PLANNING FORWARDS: POPF

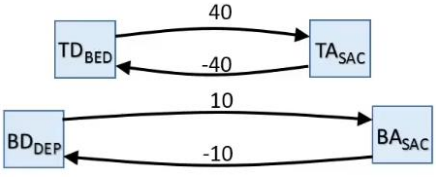
Consider the public transport domain given alongside. There is a train which goes from Bedford to St Albans, and them St Albans to London. There is a bus, which drives to the station, and then from the station to the city.



Given below are 3 partial plans that POPF could come up with

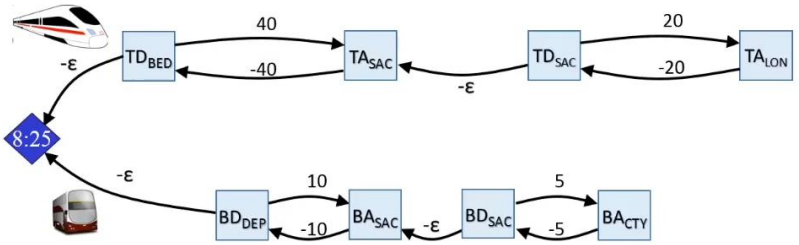


We did not include the scenario shown alongside, as that is a temporally inconsistent plan.



Thus, using POPF, the plan STN for this planning problem would look like this.

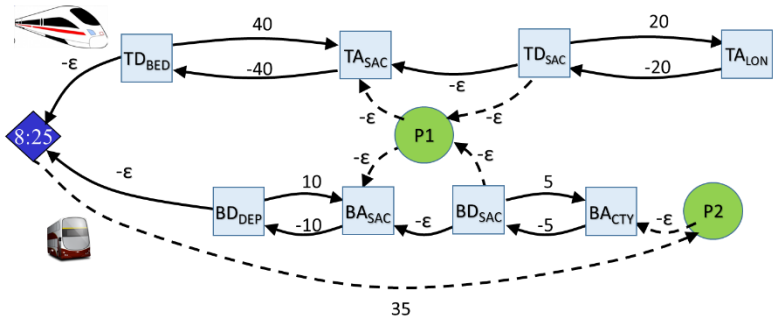
Note: there are no preferences here



OPTIMIZING PREFERENCES: OPTIC

With OPTIC we can add preferences to the problem. Thus, in this case, to make the transport system more efficient (for the travelers), we would like the bus and the train to arrive at St Albans station at the same time. We can represent this preference in the STN as `P1` and add some ordering constraints, stating the following

- The bus must arrive before the train departs
- The train must arrive before the bus departs



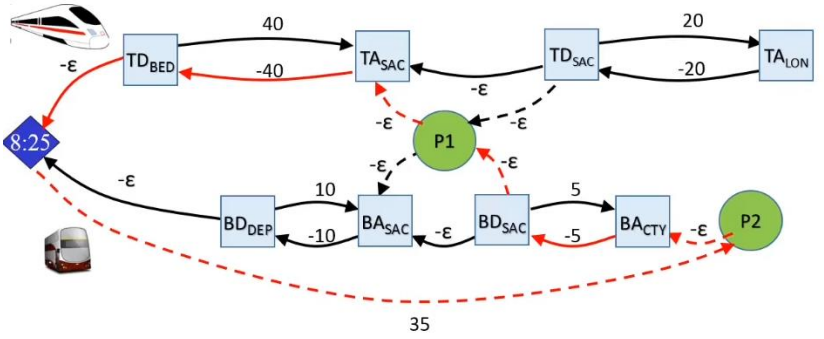
We would also like the bus to arrive in the CTY before 9am. We can model this as the second preference `P2`. It must happen ε after the bus arrives at the CTY and must happen no more than 35 time units after the beginning of the plan (assuming the plan starts at 8:25am).

ENCODING PREFERENCES IN LP

- Train arrives before bus departs: $BD_{SAC} - TA_{SAC} \geq \varepsilon$
- Bus arrives before train departs: $TD_{SAC} - BA_{SAC} \geq \varepsilon$
- Bus arrives at CTY by 9am ($t = 35$): $BA_{CTY} \leq 35$

As any preference, these are **not hard constraints**. Thus if we put them in as hard preferences then the LP could become unsolvable and the plan can be reported as invalid when it is not.

Example: In the plan, if we say that the bus must arrive at CTY at $t = 35$, it will result in a negative cycle, as it must wait at SAC for the train to arrive at SAC, which takes $t = 40$.



Thus the preferences need to be added to the plan as **optional constraints**, i.e. meet them when you can, otherwise violate them at some cost. In order to do this we need to use a **Mixed Integer Program (MIP)** and allow the option to veto the constraints. An MIP is just a LP where some variables are constrained to hold integer values (in our case 0 or 1).

BIG M CONSTRAINTS

We need:

- A 0 or 1 integer variable per preference p_1, p_2
- A very large constant M (arbitrary big number)

Now, we can rewrite the optional constraints as follows:

- Train arrives before the bus departs: $BD_{SAC} - TA_{SAC} + \boldsymbol{M} \cdot \boldsymbol{p_1} \geq \varepsilon$
  - Same as before, but we have added $M \cdot p_1$. Since M is a large number, if p_1 is 1, then the equality is satisfied, otherwise it is violated. Thus, if we set $p_1 = 0$, then $BD_{SAC} - TA_{SAC} \geq \varepsilon$ must hold, i.e., satisfy temporal constraint.
- Bus arrives before train departs: $TD_{SAC} - BA_{SAC} + \boldsymbol{M} \cdot \boldsymbol{p_1} \geq \varepsilon$
- Bus arrives at CTY by 9am ($t = 35$): $BA_{CTY} - \boldsymbol{M} \cdot \boldsymbol{p_2} \leq 35$
  - $-$ is used instead of $+$, since the inequality is $\leq$ and not $\geq$
  - Same logic; if $p_2 = 0$, then the temporal constraint must be satisfied.

Now, the **objective function** can **minimize**: $(other\ values) + c_1 p_1 + c_2 p_2$, where c_1 and c_2 are constants, in order to satisfy whatever constraints are possible, and violate the others. Now, the MIP solver can choose either to make $p_1 = 0$ ($p_2 = 0$) and satisfy the constraints or make $p_1 = 1$ ($p_2 = 1$) and violate the constraint paying the objective cost.
In the example we have been using, it is not possible to satisfy both p_1 and p_2 (as that generates a negative cycle), but we can satisfy one of them and violate the other.

Thus, the STN for OPTIC can represent many plans (as ordering is not enforced) and the MIP can provide the schedule for the plan.

CHALLENGE WITH MIP

LP can be solved in polynomial time, but MIP is an **NP-hard** problem. Even though we need to now perform a MIP solver at every stage of the plan to optimize it, it is empirically determined that its not generally time consuming as they are simple enough.

PDDL+: PROCESSES, EVENTS AND NON-LINEAR EFFECTS

So far, the only continuous actions we have seen are the ones with linear continuous numeric effects. Example: `(increase (battery) (* #t 1))`, i.e., increase the battery at a constant rate per unit time.
Now we are going to look at more advanced features of PDDL+,

- **Continuous non-linear effects**
- **Processes** (model continuous change in the environment) and **events** (model instantaneous change in the environment)

CONTINUOUS NON-LINEAR EFFECTS

PDDL+ can be very expressive, as we can write expressions of the following form:

- $\frac{dv}{dt} = \frac{P(t)}{Q(t)}$ where P and Q are polynomials over the variables defined in the planning problem.

This allows us to write (amongst other things)

- Polynomials
- Exponential Functions
- Logarithmic Functions

PDDL2.1: BALL DROP EXAMPLE

We consider the example of a ball dropping, hitting the ground and as a result, bouncing off the ground.

In the action drop-ball shown alongside, there is a **linear continuous affect**:

- $v = gt$

i.e., the velocity is increased by gt per time unit.

We also have a **non-linear continuous effect**:

- $h = vt = gt^2$

```
(: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)))))
```

Since, PDDL syntax limits us from making nonlinear relations, i.e., can't directly represent a quadratic function with continuous numeric change, we need to use an existing linear change (`velocity`), and apply the derivative to that.

We also have the invariant condition that the height of the ball must be ≥ 0 during its fall, as it cant continue falling once its reached the floor.

A better model is to see releasing the ball as separated from the fate of the ball after it falls

- Release initiates a **process** of falling
- The falling **process** can be terminated by various possible **actions** (catching, hitting, ...) or **events** (bouncing)

PDDL+: PROCESSES AND EVENTS

The planner's executive agents perform **actions** that change the world state in some way. However,

- **Processes** execute continuously under the direction of the world
  - The world decides whether a process is active or not
  - A process will execute on its own whenever its preconditions are satisfied.
  - Represent **continuous change**. Cannot have discrete numeric or propositional effects
- **Events** are the world's version of actions: the world performs them when the conditions are met.
  - Represent **discrete change**. Cannot have continuous effects

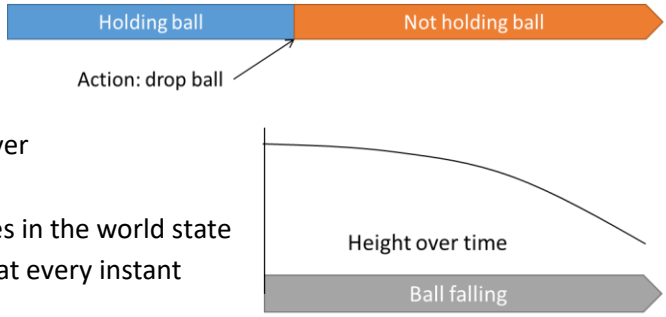
Planning is about deciding which actions the executive agents should perform by anticipating the effects of actions before they are executed.

- Requires a way to predict what will happen when actions are performed (and also when they are not), i.e., their consequences.

HYBRID PLANNING

Also known as mixed discrete continuous planning.

- When **actions** or **events** are performed, they cause **instantaneous changes** in the world
  - These are discrete changes to the world state
  - When an action or an event has happened it is over
- **Processes** are **continuous changes**
  - Once they start they generate continuous updates in the world state
  - A process will run over time, changing the world at every instant



Problem Features:

- We might care about how much of the effect of a process is generated, so we want to choose how long to let it run, for example
  - Mixing chemicals or running reactions
  - Hearing or cooling metals in a foundry
- **Duration Dependent Effects**
- We might need to interact with the process while it runs, exploiting or managing conditions during execution
- **Required concurrency with process effects**: two process can run at the same time, while having an effect on the same variable

PDDL+: BOUNCE

As mentioned earlier, a *better* model is to see releasing the ball as separated from the fate of the ball after it falls

We modify the `drop-ball` action (now called `release`). It is now an **instantaneous action** (not durative).

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

Now we allow the environment to take care of the **process** of the ball falling. The process `fall`, will happen whenever its precondition is met.

```
(: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)))))
```

The continuous effects of the `drop-ball` action have now been placed in the `fall` process. Process cannot have discrete numeric effects and they cannot have propositional effects, as those are handled by events.

Once the ball hits the ground, the ball will `bounce`, which is caused by the environment, and is not a result of something we do. Thus, `bounce` is modelled as an **event**. Again, just like processes, once the preconditions of the event are met, the event will happen (no matter what, we don't have a choice). **Events** can only have **discrete numeric** (and propositional) effects and no continuous effects.

```
(: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)))))
```

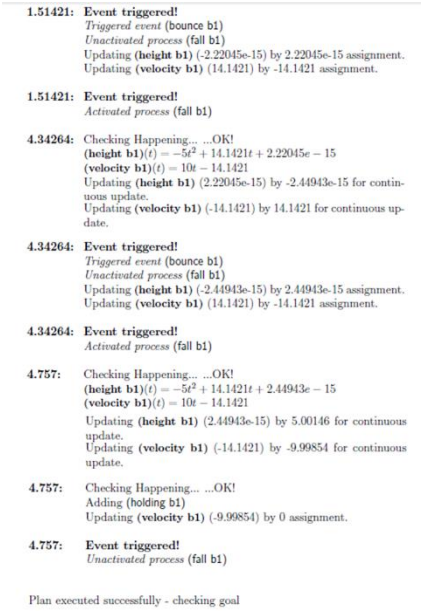
Once the ball bounces, we can attempt to catch it with a robot hand which is at height = 5.
We are using an `assign` effect since at the time of catching the ball, we don't know what its velocity was, but once we have caught it, we know that it will be 0.

```
(:action catch
 :parameters (?b - ball)
 :precondition (and
    (>= (height ?b) 5)
    (<= (height ?b) 5.01))
 :effect (and
    (holding ?b)
    (assign (velocity ?b) 0)))
```

A valid plan for this problem would look like the one shown alongside. The plan only mentions the actions, as the processes and events are happening on their own.

```
0.1: (release b1)
4.757: (catch b1)
```

The planner will consider when the processes and events are executed but will not include them in the plan.



During the plan we see a line: *Unactivated process* `(fall b1)`. This is because at this point the height goes slightly below 0 (due to numeric tolerance), and the precondition of the process is no more satisfied. But right after that, the process is activated again since the height is now ≥ 0.

We can also get some graphs from the **validator**.

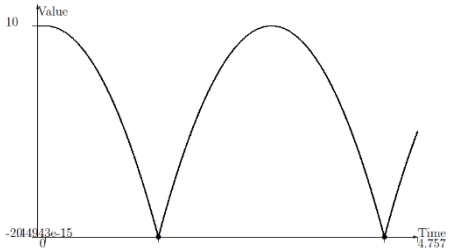


Figure 2.1: Graph of (height b1).

We can see that

- the **height** of the ball is **quadratic**
- and that the ball bounces twice before being caught at $t = 4.757$.

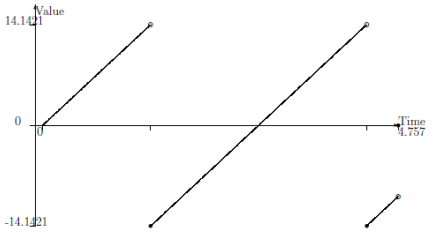


Figure 2.2: Graph of (velocity b1).

We can also see that the **velocity** is **linear** but **discontinuous**.

If we want to model an even more realistic scenario, where the bound is not perfect and the ball looses some energy with each bound, we can add the coefficient of restitution to the `bounce` event.

```
(:event bounce
 :parameters (?b - ball)
 :precondition (and  (>= (velocity ?b) 0)
    (<= (height ?b) 0.00001))
 :effect (and (assign (velocity ?b)
    (* (coeffRest ?b) (velocity ?b)))))
```

Notice further that the precondition on `height` is slightly relaxed to allow bouncing to happen until the ball is too close to bounce any more, i.e., the ball eventually stops bouncing.

Now, we can see that the ball bounces 14 times, before coming to a stop.

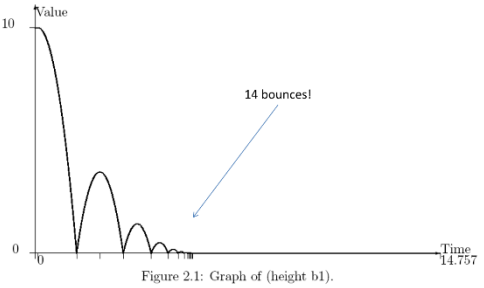


Figure 2.1: Graph of (height b1).

ZENO BEHAVIOR

If we look closely at the bouncing ball, we see that eventually the events are so close (in an idealized view, at least) that infinitely many bounces happen in a finite time. This is **Zeno behavior**, and it is **A Bad Thing**, as we cannot reason about infinite events in finite time in a planner, a validator, or anywhere else.

CASCADING EVENTS

Events (and processes) can be **problematic** in lots of interesting ways. It is easy to set up **events that trigger each other**:

- Imagine a light connected to a light-sensitive switch, so that when the light is on it triggers an event which turns the light off
    - But when the light is off an event is triggered that turns it on!

Thus, we would **require that all events delete their own preconditions**, so that an event does not keep triggering itself.

STRICT INEQUALITIES IN PRECONDITIONS

Events with **strict inequality preconditions** ($<, >$) can create an interesting **problem**.

Consider the example given alongside. The process P will start executing at $t = 0$ (since the precondition is met). As a result, it will increase the value of x, per unit time.

```
(:event E
  :precondition (> 10 (x))
  :effect (and (not (active)) (explode)))
(:process P
  :precondition (active)
  :effect (increase (x) (* #t 1)))

:init (= (x) 0) (active)
```

When does E occur?

- At $t = 10$, $x = 10$, so the precondition of E is not satisfied.
- At $t = 10 + \varepsilon$ (for any $\varepsilon > 0$), the precondition of E is satisfied. But it was also satisfied at $t = 10 + \frac{\varepsilon}{2}$, thus E should trigger between $t = 10$ and $t = 10 + \varepsilon$.

Thus, we don't know when to trigger E.

Thus, **preconditions** of **events** and **processes** must contain **inequalities** ($\leq, \geq$)

COURSEWORK RESEARCH

FAST DOWNWARD PLANNER

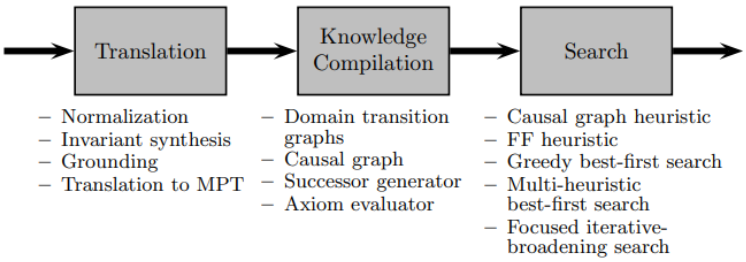
- The FD planner is a planning system based on heuristic search – uses heuristics to solve problems.
- It is a progression planner, searching the space of world stages of a planning task in the forward direction.
- The planning system is based on heuristic state space search and hierarchical problem decomposition.
    - The system creates a causal graph and a domain transition graph (DTG) from a domain that components are used to optimize the preconditions of the planning tasks
- Starting from top level goals – the algorithm recurses further and further down the causal graph until all remaining subproblems are basic graph search tasks
- Search part – Chosen search heuristic combination is run until a solution is found (or not)
    - Search Algorithm: Uses heuristic estimate of a heuristic function of every successor state to find a preferable way to the goal state – gives the function a chance to detect dead end states
        - Thus, heuristic functions are required to detect dead end states in FD planner

PHASES OF EXECUTION

1) Translation

- PDDL input is transformed to a non-binary form which is easier to handle for further hierarchical planning approaches
- Creates a multi-valued planning task as output

2) Knowledge Compilation



- Generates a data structure consisting of the following four
  - *Domain Transition Graph*: Encoded how and under which conditions state variables change their values
  - *Causal Graph*: Represents hierarchical dependence between different state variables
  - *Successor Generator*: Set of applicable operators in a given state
  - *Axiom Evaluator*: data structure for computing values of derived variables

3) Search

- 3 different implementations of search algorithms are used for planning
  - *Greedy Best First Search*
  - *Heuristic Best First Search* (extension of greedy BFS; tries to combine several heuristics)
  - *Focused Iterative-Broadening Search*

EXAMPLE

Consider the example given alongside. There are 2 cities, 3 cars and 1 truck. There are 2 parcels which need to be delivered.

Goal State:

- p_1 needs to be at G
- p_2 needs to be at E

The obvious high level plan for delivering p_1 would involve loading it into a car (c_1 or c_2) in order to get p_1 to D, load p_1 on t and take it to E, load p_1 onto a car (c_3 in this case) and drop it off at G (its destination).

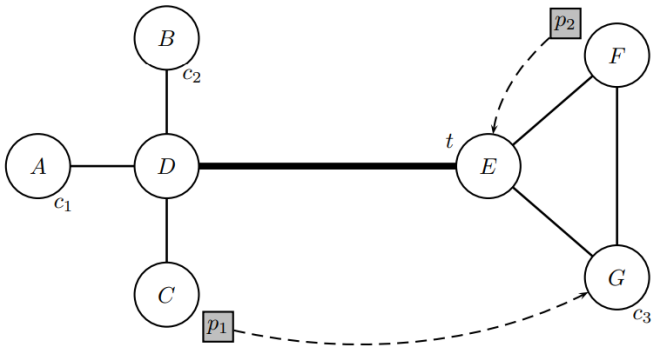
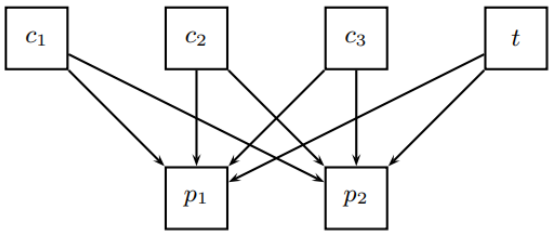


Fig. 9.1. A transportation planning task. Deliver parcel p_1 from C to G and parcel p_2 from F to E, using the cars c_1, c_2, c_3 and truck t. The cars may only use inner-city roads (thin edges), the truck may only use the highway (thick edge).

Similarly, the obvious high level plan for delivering p_2 to E would be to load it into a car (c_3) and drop it off at its destination E.

This gives rise to the ***causal dependencies*** in the transportation planning tasks as shown alongside. We say that the parcels have causal dependencies on the vehicles because they are the operators that change the state of a parcel. Furthermore, parcels do not depend on each other and vehicles do not depend on each other.



The idea is to fill in the high-level plan with movements of the vehicle fleet. Thus, we can decompose the planning tasks into several subtasks:

- *Parcel Scheduling Tasks*: Where, and by which vehicle, a parcel should be loaded and unloaded
- *Path Planning Tasks*: How to move a vehicle from point X to point Y?

Both of these are graph search tasks, and the corresponding graphs are called ***Domain Transition Graphs***.

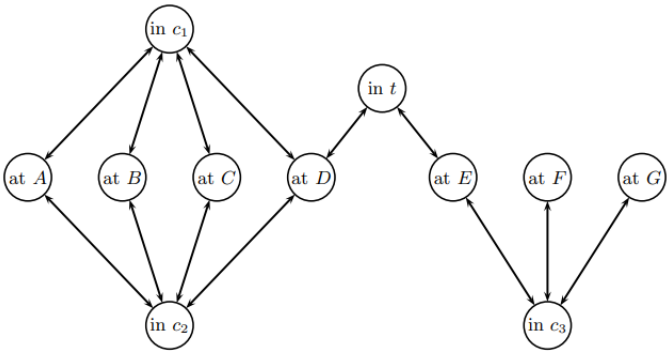


Fig. 9.3. Domain transition graph for the parcels p_1 and p_2. Indicates how a parcel can change its state. For example, the arcs between "at D" and "in t" correspond to the actions of loading/unloading the parcel at location D with the truck t.

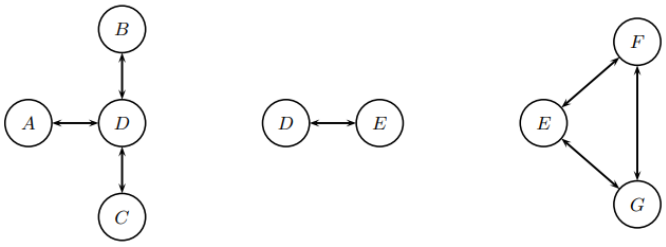


Fig. 9.4. Domain transition graphs for the cars c_1 and c_2 (left), truck t (centre), and car c_3 (right). Note how each graph corresponds to the part of the roadmap that can be traversed by the respective vehicle.

Drawbacks of Cyclic Causal Graphs

If a causal graph exhibits a cycle, hierarchical decomposition is not possible, as the subtasks that must be solved to achieve an operator precondition are not necessarily smaller than the original task.

Thus causal graph heuristic requires acyclicity; it considers relaxed planning tasks in which some operator preconditions are ignored to break causal cycles.

Looking back at causal graphs, in reality they are much more complex than the one shown earlier and look like the one shown alongside.

This is because the earlier causal graph was a high-level dependency graph created using non-binary variables. The STRIPS-level state variables correspond to (binary) object-location propositions like "parcel p_1 is at location A"

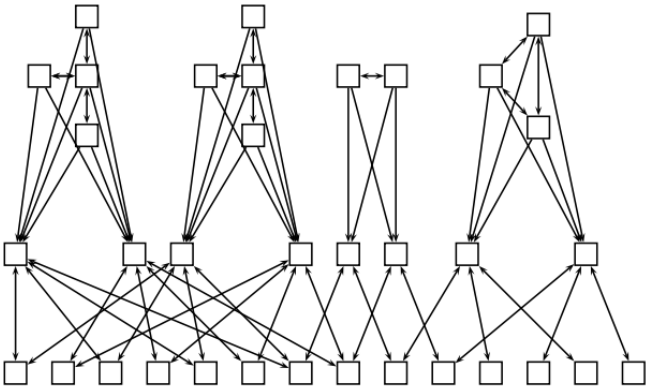


Fig. 9.7. Causal graph for the STRIPS encoding of the transportation planning task.

ADDITIVE HEURISTIC (h^{add})

The h^{add} heuristic estimates the cost of a set of elements as the sum of the costs of this elements.

PROPERTIES OF h^{add}

- Falls under the category of **delete-relaxation domain-independent** heuristic
- It is an **inadmissible heuristic** (it may overestimate the true cost).
- It is **not consistent**
- It assumes that subgoals are independent
    - This is not true in general as the achievement of some subgoals can make the achievement of other subgoals more or less difficult (ties back into h^{add} being inadmissible)

CAUSAL GRAPH HEURISTIC (h^{CG})

The Causal Graph heuristic (h^{CG}) estimates the number of operators that are needed to reach the goal from a state.

PROPERTIES OF h^{CG}

- Falls under the category of **delete-relaxation domain-independent** heuristic
- It is an **inadmissible heuristic** (it may overestimate the true cost).
- It is **not consistent**
- It is **not complete**

h^{CG} PLANNING PROCESS OUTCOMES

- A plan is found - solved task.
- The search space is explored until all frontier sates are assigned a heuristic value of ∞ and verified to be dead ends - task proven unsolvable.
- The search space is explored until all frontier states are assigned a heuristic value of ∞, but some of them are not verified to be dead ends – failure (very rare)
- Planner exhausts time or memory bound – failure

RELEVANT TERMINOLOGY

- **Dead Ends**: formulae that are satisfied in states that make the goal unreachable
- **Invariants**: Formulae that are true in the initial state and in all reachable state
- **Trap**: It is a conditional invariant; once a state is reached that makes the trap true, all the states that are reachable from it will satisfy the trap formula as well.

PLANNING DOMAINS TO CONSIDER

FreeCell

Based on solitaire card game with the same name. The original card game is played with a standard deck of 52 cards, initially arranged into eight tableau piles of 6 or 7 cards each. Cards can be moved between these eight tableau piles, four free cells and four foundation piles according to a set of rules. The objective is to move all cards to *foundations*.

REFERENCES

- *Helmert, M. 2006. Solving Planning Tasks in Theory and Practice.*
- *Helmert, M. 2006. The Fast Downward Planning System. Journal of Artificial Intelligence Research (JAIR).*
- *Hoffmann, J. 2011. Analyzing Search Topology Without Running Any Search: On the Connection Between Causal Graphs and h+. Journal of Artificial Intelligence Research (JAIR).*
- *Helmert, M. 2004. A Planning Heuristic Based on Causal Graph Analysis. ICAPS-04 Proceedings*
- *Lipovetzky, N. Muise, C. Geffner, H. Traps, Invariants and Dead-ends.*
- *Geffner, H. Bonet, B. 2001. Planning as Heuristic Search.*
- *Reißner, C. 2013. Researching Heuristic Functions to Detect and Avoid Dead Ends in Action Planning.*