PLANNING IN THE REAL WORLD

Real world planning

- Last time we looked at the STRIPS approach to plannning, which works for simple examples.
- We also looked at how partial-order planning solved some problems with the STRIPS approach.
- This time we'll look at further extensions that get us closer to being able to deal with the real world.
  - Dealing with abstract actions
  - Learning the effects of actions
  - A non-deterministic world.

Hierachical planning

- State-of-the-art planning algorithms can build plans with a few thousand actions.
  - Sounds like a lot
- BUT small compared to a lifetime of actions:

$$10^3 \times 10 \times 10^9 = 10^{13}$$

  (muscles $\times$ frequency $\times$ lifetime)
- Even a few weeks will contain $\approx 10^{10}$ actions.
- We deal with this by *abstraction*.

Vacation planning

Vacation planning

- We plan:

  Go to aiport
  Catch plane
  Got to hotel
  ⋮

  not at the level of:

  Go to door
  Exit room
  Go to end of corridor
  ⋮

- However, we can refine any of the abstract actions as required.

Hierarchical task networks

- In HTNs we still use the standard STRIPS rules.
  – We will now call these *primitive actions*.
- In addition we have *high level action*s (HLA).
- Each HLA has a *refinement*.
- A refinement is a sequence of actions
  – either HLAs or primitive actions.

  thus to get to primitive actions from an HLA might take several refinements.

- Consider the high level action:

$$Go(Home, EWR)$$

this has refinements:

$$\begin{aligned}&Refinement(Go(Home, EWR)\\&\quad STEPS : [Taxi(Home, EWR)])\end{aligned}$$

$$\begin{aligned}&Refinement(Go(Home, EWR)\\&\quad STEPS : [Drive(Home, EWREconomyLotH),\\&\qquad\qquad Shuttle(EWREconomyLotH, EWR)])\end{aligned}$$

$$\begin{aligned}&Refinement(Go(Home, EWR)\\&\quad STEPS : [Walk(Home, GrandStSubway),\\&\qquad\qquad Subway(GrandSt, PennStation),\\&\qquad\qquad NJT(PennStation, EWR)])\end{aligned}$$

- Navigation for a grid-based robot:

$$Navigate([a, b], [x, y])$$

this has refinements:

$$\begin{aligned}&Navigate([a, b], [x, y])\\&\quad PRECOND : a = x, b = y\\&\quad STEPS : [\,])\end{aligned}$$

$$\begin{aligned}&Navigate([a, b], [x, y])\\&\quad PRECOND : Connected([a-1, b], [x, y]),\\&\quad STEPS : [Left, Navigate([a-1, b], [x, y])])\end{aligned}$$

$$\begin{aligned}&Navigate([a, b], [x, y])\\&\quad PRECOND : Connected([a+1, b], [x, y]),\\&\quad STEPS : [Left, Navigate([a+1, b], [x, y])])\end{aligned}$$

- Here the refinements are *recursive*

- Why will this help?

- Can first plan at an abstract level and only later worry about which refinements to use.
- So don't have to think about all the primitive actions that are available.
  - Not when doing the abstract plan.
  - Not when refining.
- Massively reduces the search space.

Implementations

- An *implementation* of of an HLA is a refinement that only contains primitive actions.
  - It is a refinement that could be executed.
- An implementation of:

$$\mathit{Navigate}([1,3],[3,2])$$

is

$$[\mathit{Right}, \mathit{Right}, \mathit{Down}]$$

- A high level plan is a sequence of HLAs:

$$\mathit{Navigate}([1,3],[3,2]), \mathit{CollectDirt}, \mathit{Navigate}([3,2],[1,3])$$

- An implementation of a high level plan is just the concatenation of implementations of each HLA in the plan:

$$[\mathit{Right}, \mathit{Right}, \mathit{Down}, \mathit{SuckDirt}, \mathit{Left}, \mathit{Up}, \mathit{Left}]$$

- A high level plan *achieves a goal* from a given state if at least one of its implementations achieves the goal from that state.
  - Only need one implementation to work for the high level plan to be good.

- Hierarchical planning often starts with a single top level action *Act*.
  - Find an implementation that satisifes the goal.
- Simple algorithm,
- Repeatedly picks an HLA in the current plan and replace it with a refinement until the plan reaches the goal.

```
function H-SEARCH(problem, hierarchy) returns plan or fail
frontier ← a FIFO queue with [Act] as only element
  loop do
     if EMPTY?(frontier) then return fail
     plan ← POP(frontier)
     hla ← the first HLA in plan, or null if none
     prefix,suffix ← action sequences before and after hla in plan
     outcome ← RESULT(problem.INITIAL-STATE,prefix)
     if hla is null then
        if outcome satisfies problem.GOAL then return plan
     else
     for each sequence in REFINEMENTS(hla, outcome, hierarchy)
        do
        frontier ←
          INSERT(APPEND(prefix, sequence, suffix), frontier).
```

- The algorithm on the previous slide is basically breadth-first.
- There is a set of plans in *frontier*.
- The algorithm takes each plan and expands the first HLA.
- One new plan is generated for each refinement of that HLA.
  - Each new plan is the old plan with an HLA replaced by its refinement.

  and each new plan is put back on *frontier*
- Can easily develop depth-first or interative deepening versions.

- The key to hierachical planning is the set of refinements
  - Plan library
- Encoding knowledge in both actions sequences and preconditions.
  - Want HLAs with small number of refinements, but each refinement having lots of actions.
- O-PLAN used by Hitachai to develop production plans.
  - 350 products
  - 35 machines
  - 2000 operations

  can generate 30-day schedules.

Reinforcement learning

- We have been considering techniques for planning.
- The view we have taken is that an agent knows all about its environment and plans by thinking hard about what it wants to do.
- Instead we can think of planning as a process of exploring the environment around the agent.
- We'll look at a number of approaches based on this idea.
- These are all types of *reinforcement learning*.
- They also link back to search (again).

- We'll start by considering that we have a state-space in which the agent is carrying out actions.
- We want to come up with a plan.
- Well, in fact what we end up with is a *policy*.
- That is matrix that tells us which action to carry out in which state.
- This is a *conditional plan*, which is much more robust than a linear plan as produced by STRIPS.
- One way to get this is by determining a value for each state — the for each state helps to tell us which action we should pick.

Learning values for states

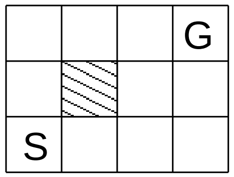
- We will start by assuming that the agent knows the results and costs of each operation.
- We will also assume that it can build the whole search tree.
- This is just what we did when we dealt with search before.
- We then set $h(n) = 0$ for all n and run an A* search.
- When the agent has expanded node n_i to give a set of children $\delta(n_i)$, it updates its $h(n_i)$ to be:

$$h(n_i) := \min_{n_j \in \delta(n_i)} [h(n_j) + c(n_i, n_j)]$$

  where $c(n_i, n_j)$ is the cost of moving from n_i to n_j.
- We further assume that the agent can recognise the goal state and knows that $h(goal)$ is 0.

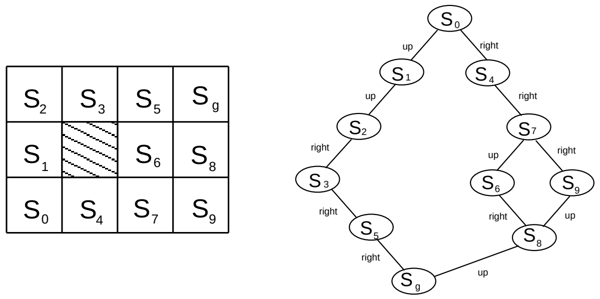
- Let's look at how this works.

- Robot starts at S and wants to get to G.



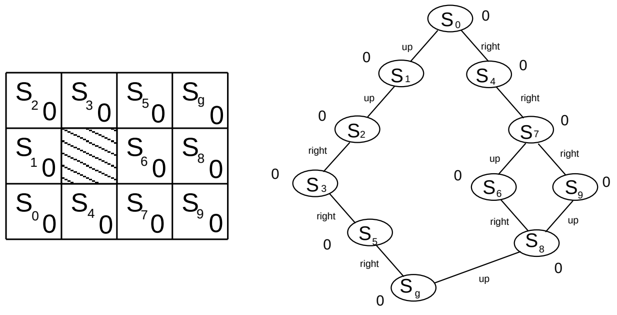
- Robot can move up, down, left, right. All moves have the same cost = 1.

- The grid has a corresponding search space.

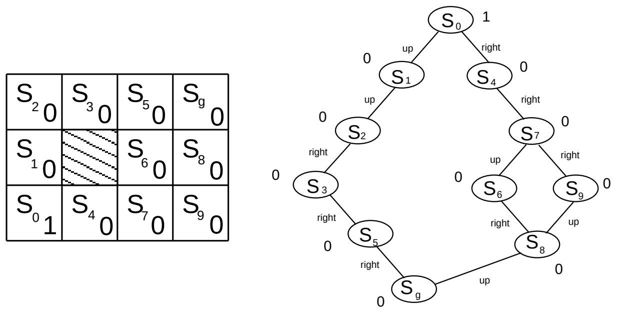


- The state space assumes breadth-first search and doesn't visit the same state twice.

- Initially all h values are zero.

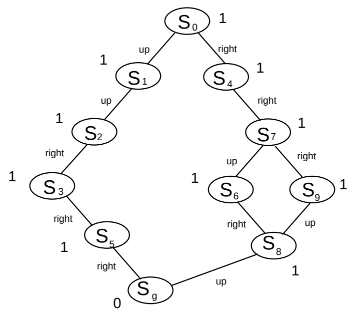

S_2 0 S_3 0 S_5 0 S_g 0
S_1 0 S_6 0 S_8 0
S_0 0 S_4 0 S_7 0 S_9 0
up
right

- Then the agent updates the value for S_0. Both its children have value 0 and it costs 1 to get to them, so:


S_2 0 S_3 0 S_5 0 S_g 0
S_1 0 S_6 0 S_8 0
S_0 1 S_4 0 S_7 0 S_9 0
up
right

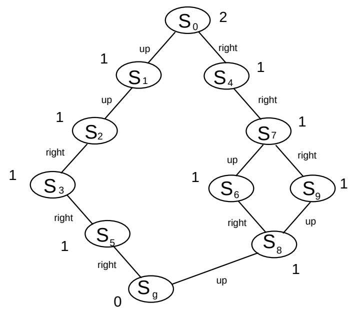
- The same update will apply to each node in turn, except S_g which has its value fixed at 0 since it is the goal.

S_2 1	S_3 1	S_5 1	S_g 0
S_1 1	▨	S_6 1	S_8 1
S_0 1	S_4 1	S_7 1	S_9 1



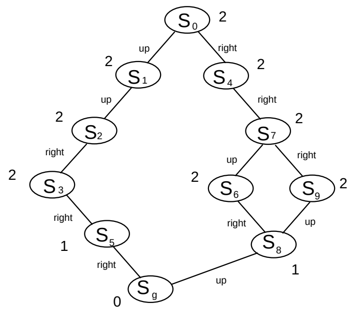
- The agent repeats the process. Now S_0 has value 2 since its children have value 1

S_2 1	S_3 1	S_5 1	S_g 0
S_1 1	▨	S_6 1	S_8 1
S_0 2	S_4 1	S_7 1	S_9 1



- All the remaining nodes will get value 2, except S_5 and S_8 since their value is fixed by S_g

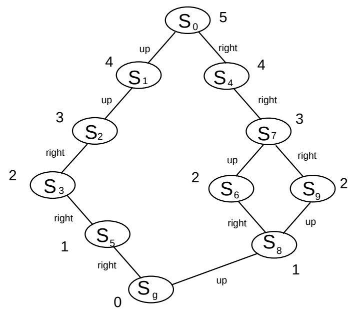
S_2 2	S_3 2	S_5 1	S_g 0
S_1 2	▨	S_6 2	S_8 1
S_0 2	S_4 2	S_7 2	S_9 2



- Each successive update will increase the value of nodes whose value does not reflect their distance from the goal.

- Eventually we end up with this:

S_2 3	S_3 2	S_5 1	S_g 0
S_1 4	▨	S_6 2	S_8 1
S_0 5	S_4 4	S_7 3	S_9 2



- If you don't see why, go back and run through all the updates.

- The approach doesn't do much for the agent the first time — it is just path cost search.
- However, subsequent searches "zoom in" on the right solution faster and faster.
- This happens as the $h_T(n)$ values propagate back from the goal.
- (All of this assumes there are few enough values that they can be stored in a table.)
- Each run propagates the true cost of getting to the goal further back through the search.
- We can tell that learning is complete by looking at whether values change — if no value changes after another iteration, then the values are correct.

- Eventually, the minimal cost path can just be read off the tree and whatever the state the agent is in, it knows what to do.
- It makes its choice by

$$a = \text{argmin}_a \left[h(\textsc{Result}(n_i, a)) + c(n_i, \textsc{Result}(n_i, a)) \right]$$

  where, as before, $\textsc{Result}(n_i, a)$ is the state reached from n_i after carrying out a.
- In other words, the agent looks at all the states it can get to from its current state, computes the sum of the heuristic value of each of those states and cost of getting to that state, and then picks the action which minimizes the result.
- When it has a correct value for each state, this procedure will take it straight to the goal.

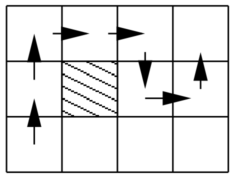
- Even when the agent doesn't have the correct value for each state, it can still use it to find the goal.
- It just uses the value as a heuristic in A^* search.
- Since the value always underestimates the distance to the goal, the hueristic is admissable.
- However, the search will typically take longer to find the goal than when the learning is complete.

Learning without a model of action

- The search we described above is *off-line*.
  - The search would typically be run before the agent does anything.
  - The final values are then used to act.
- If the agent is a robot, this would all happen before the robot moved at all.
- An alternative is to do the learning *online*.
  - The agent to actually carries out the actions to see what happens.

- This is rather similar to the way in which we learn how to do unfamiliar things.
- In the case of the robot it could move through the grid randomly at first, working out over a number of runs what the outcomes of actions were, and which were most useful at which point.
- To do this, the agent will have to build a model of the state space in its "head" as it moves.

- What we assume is that:
  - The agent can distinguish the states it visits (and name them).
  - The agent knows how much actions cost once it has taken them.
- The process starts at the start state s_0.
- The agent then takes an action (maybe at random), and moves to another state. And repeats. So for our previous state space we might have:

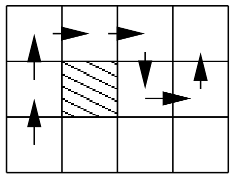


- As the agent visits each state, it names it and updates the heuristic value of this state as:

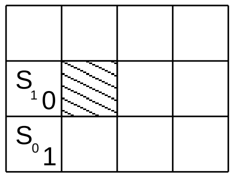
$$h(n_i) := [h(n_j) + c(n_i, n_j)]$$

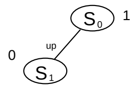
  where n_i is the node in which an action is taken, n_j is the node the action takes the agent to, and $c(n_i, n_j)$ is the cost of the action.
- $h(n_j)$ is zero if the node hasn't been reached before.
- As before, the estimated minimum cost path to the goal is built up over repeated runs.

- Let's look at how this works for the run:



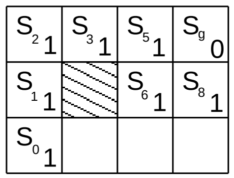
- After the first move, the agent knows that executing up in S_io takes it to S_1.

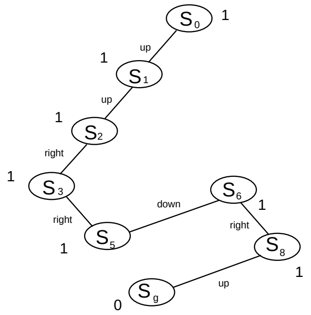




- And having made the move and found that $h(S_1) = 0$ (since it is a new state) the $h(\cdot)$ value of S_o. Can be updated.

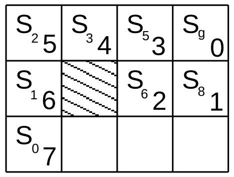
- The same expansion will take place, along the path until the robot gets to the goal state.

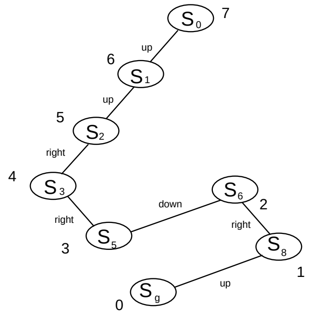




- Note that the tree is different from in the previous example.

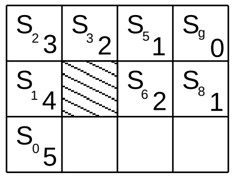
- If the robot keeps executing the same path, over time it will build up a set of values similar to those in the offline case:

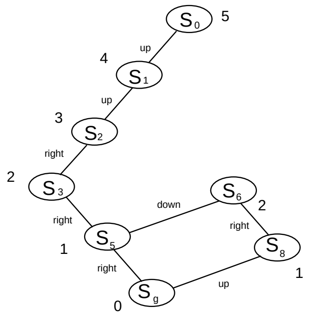




- (Imagine it gets teleported back to the start when it reaches the goal)

- If the robot executes an action that isn't on the path, it will fill in more of the search tree:





- This will then change the values of other nodes in the search graph (though all the changes may not be propagated for some time, depending on how the robot chooses to update the graph).

- As before the agent can use the partially learnt value to decide the best action.
- Whenever the agent has to choose an action a, it can choose using:

$$a = \text{argmin}_a \left[h(\textsc{Result}(n_i, a)) + c(n_i, \textsc{Result}(n_i, a))\right]$$

.

- Assigning a high value to the results of actions that have not been tried will cause the robot to *exploit* paths that it knows.
- Assigning a low value to the results of actions that have not been tried will cause the robot to *explore* paths that it does not know.
- Classic example of the trade-off between *exploitation* and *exploration* in learning.
- Allowing some randomness in the choice of actions is one way to balance this.

- The big difference with the previous case is that the agent combines actions and establishing the value.
- So, over time, it learns the best thing to do by trying out actions.

Planning in non-deterministic domains

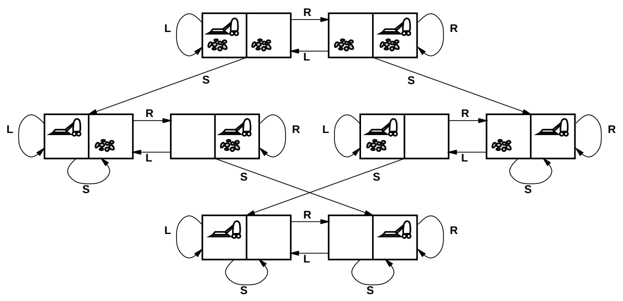
- The major issue with STRIPS, POP and even HTNs is that actions are not deterministic.
- As soon as you start executing a plan, actions may not do what you want then to do.
- To provide a full solution we need to *reason about uncertainty*
  - Will do that in a few weeks after we look at using probability.
- In the meantime we'll look at some other approaches.

Partial solutions

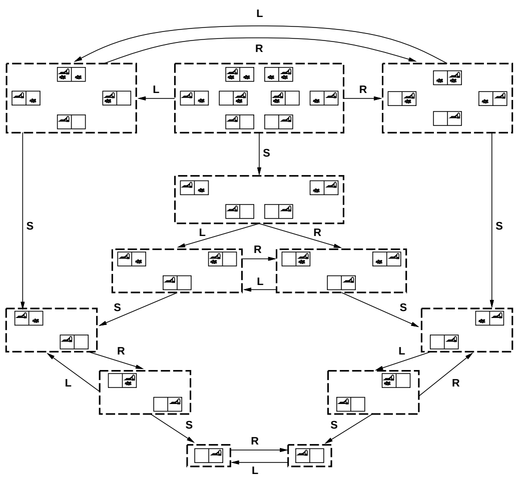
- Conformant/sensorless planning
  - Devise a plan that works from any state
  - May well not exist
- Conditional planning
  - Subplan for each contingency
- Monitoring/replanning
  - Check progress during execution
  - Replan when necessary
- Probably would need a combination of all of these.

Sensorless planning

- Remember the vaccum world:



- What if we had no sensors?
- Wouldn't be able to distinguish states
- Have to think in terms of "belief states"



- Here a plan like:

  Right
  Suck up dirt
  Left
  Suck up dirt

  will work.
- Of course, it may involve additional actions.

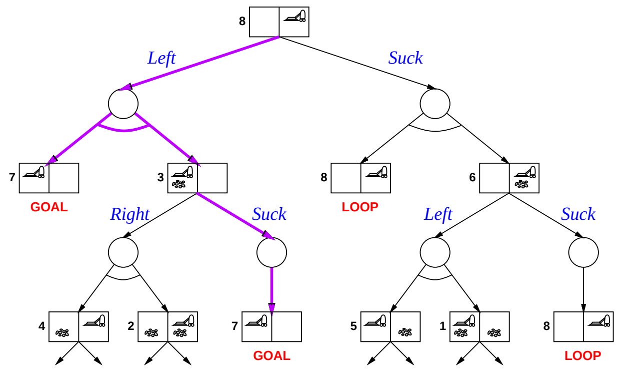
Conditional planning

- Conditional plans check (any consequence of KB +) percept

$$[\ldots, \textbf{if } C \textbf{ then } Plan_A \textbf{ else } Plan_B, \ldots]$$

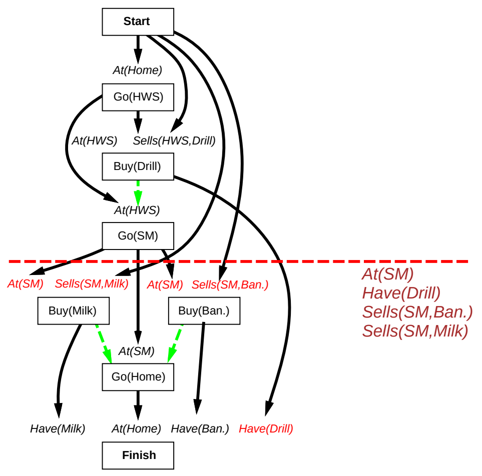
- Execution: check C against current KB, execute "then" or "else"
- Need *some* plan for *every* possible percept
- The learning we looked at before is one way to provide this.

- Double Murphy: sucking or arriving may dirty a clean square



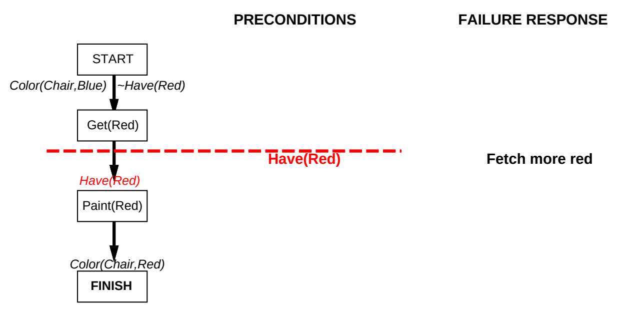
Execution Monitoring

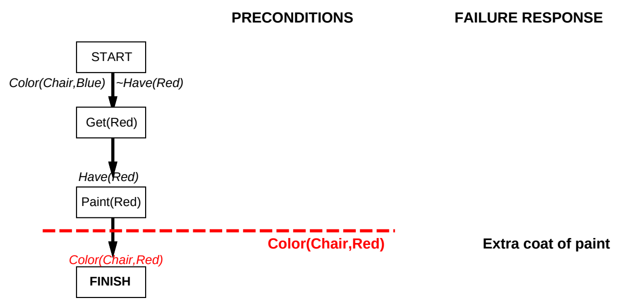
- "Failure" = preconditions of *remaining plan* not met
- For a partial order plan, this means:
  - All preconditions of remaining steps not achieved by remaining steps
  - All causal links *crossing* current time point
- See example on next slide



- On failure, resume POP to achieve open conditions from current state
- IPEM (Integrated Planning, Execution, and Monitoring):
  - Keep updating *Start* to match current state
  - Links from actions replaced by links from *Start* when done

Emergent behavior





- "Loop until success" behavior *emerges* from interaction between monitor/replan agent design and uncooperative environment.

Summary

- This lecture looked at some additional ideas about planning.
- We first looked at making the simple planning techniques from the last lecture more scaleable.
  - Abstraction
  - Hierarchical planning.
- We then looked at using learning to help an agent plan.
- Finally we started looking at the problems caused by non-deterministic action.
- However, the approaches we looked at were limited, and we will need to come back to this topic once we have looked at approaches for handling uncertainty.