Uninformed search

Reflex vs. planning agents.

- Reflex agents
  - Choose an action based on current observations (and maybe memory)
  - Do not consider the future consequences of actions
- Can a reflex agent be rational?
  - Well chosen reflex agents can actually go very far in implementing useful behaviors
  - Many animals might be reflex agents, humans have many reflexive behaviors.
- How do you implement it?
  - Simplest: lookup table $a \leftarrow lookup[obs]$
  - Function approximation: $a \leftarrow f(obs)$

Planning agents

- Plan a certain set of actions $plan = \{a_1, a_2, \ldots\}$
- During execution time, just execute the actions, as listed in the plan
- Planning:
  - Ask "what if" a certain action is done, make decisions based on the (hypothesised consequences)
  - Must have a world model $T(s, a) \rightarrow s'$ which tells how the world evolves in response to actions

Partial, complete and shortest paths

- **Partial plan**: it does not reach the goal
- **Complete plan**: goal is achieve at the end
- **Optimal plan**: some kind of additional optimization criteria
    - Lowest number of actions
    - Lowest cost (cost associated with actions) - eg time, energy
    - Preferred states visited along the plan

Challenges of uncertainty

- Even if the plan is perfect, it might not succeed
  - Uncertainty in actions (T probabilistic)
  - Other agents acting in the world
- A possible solution: **replanning**
  - Redo the planning whenever situations diverge from what was expected
  - **Contingency plans**: create plans ahead of time for possible negative events
  - **Model predictive control**: Make a complete plan, but only perform first action, replan at every step afterwards

The problem of searching for a plan

- A **problem of searching for a plan** consists of
  - State space $S = \{s_1, s_2, ...\}$
  - Successor function $T(s, a) \rightarrow s'$
  - Start state s_0
  - Goal test $G(s) \rightarrow \{true, false\}$
- Together, they *imply* a **state space graph**
- **Solution**: a plan that transforms the start state to goal state
  - a list of actions $\{a_{p1}, \ldots, a_{pm}\}$

Model state vs world state

- The search problem is a given only in AI class homework and exam problems.
- Otherwise: setting up the problem correctly is critical.
- The search problem is a **model**: a mathematical object that captures those aspects of the world that are useful for the solution or the problem and *ignores the rest*
- We need to distinguish between the **world state** which is always very large and complex and the **model state** which we try to tailor to the problem.

Modeling exercise 1 (Goat-wolf-cabbage)

- Representation of states, count
- Representation of actions
- Human understandable vs computer efficient
  - state: {GC | HW}, action

Modeling exercise 2 (Harry Potter)

- Harry Potter (HP), Albus Dumbledore (AD), Horcrux HX1, HX2...
- Map: Hogwards (hw), Hogsmeade (hm), Gringotts (gr) and London (ln)
- hw-hm, hm-gr, hm-ln, ln-gr
- P1: path planning
- P2: one horcrux HX in total
- P3: each location might have a horcrux

State space considerations

- **Exponential explosion** of number of states
    - Every time a **feature** might or might not be present, it doubles the state space
- Building the state space graph explicitly is often impossible
- In some cases, states and transitions are only revealed *during* the search
    - e.g. fog of war in games
- In other times, we generate them as we go

Planning with a search tree

- Root node: *labeled* with the start state s_0
- Downward edges from nodes: actions
- Nodes: *labeled* with the state
  - A state can appear multiple times in the search tree!
- As this is a tree, for each node, there is a unique path from the root
  - The edges of that path is the **plan** that gets us to this state!

Search tree considerations

- Can get very large, unlikely that we can build it completely.
- It can get infinitely large, if there is a loop in the state graph
- For the Harry Potter example: hw - hm - gr - ln - gr - ln ...

Tree search algorithm

- Consider nodes as **partial plans**
- Start from the root
- Moving from a note to its children is called **expanding** a node
- Maintain a collection datastructure called the **fringe**: nodes that we know that we need to expand
- Stop when we found a **complete plan**: the node we are expanding is in the goal set.

General tree search algorithm for planning

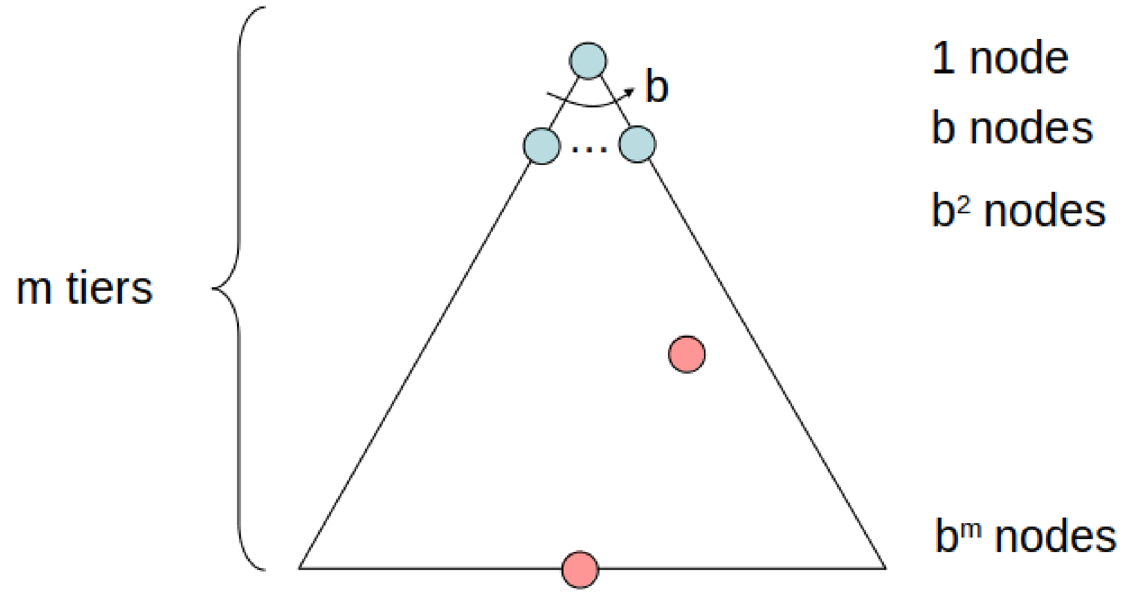
```
function TREE_SEARCH({S, T, s_0, G}, strategy):
    fringe = {s_0}
    loop
        if fringe == {} return failure
        choose node n from fringe according to strategy
        if G(n) return solution
        remove n from fringe
        create successor nodes of n based on T(n) and add them to fringe
```

Understanding the general tree search algorithm

- Amazing algorithm, works for any problem!
- Critical part: **strategy**
  - How to pick the next node from the fringe
  - The fringe, as a datastructure, should support the strategy
- Determines:
  - Whether we find a solution
  - Whether we find the optimal solution
  - How long do we search until we find a solution
  - Which solution we find first

Properties of a search algorithm

- **Completeness**: guaranteed to find a solution if one exists?
- **Optimal**: least cost plan?
- **Time complexity**?
- **Space complexity**?
- b branching factor
- m maximum depth
- Total nodes?
  $1 + b + b^2 + \ldots + b^m = O(b^m)$

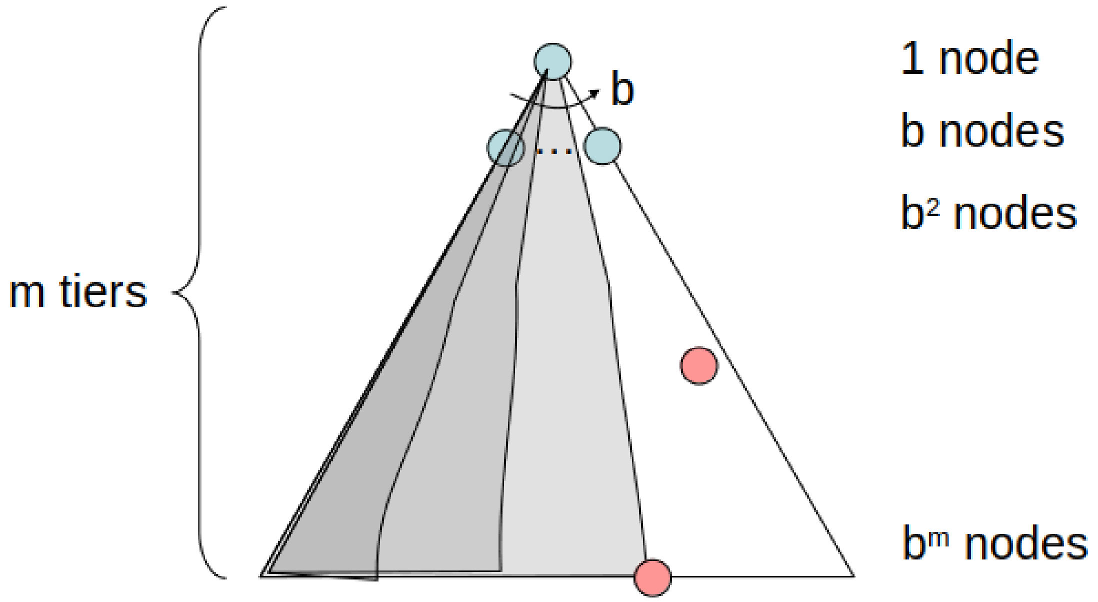


Depth-first tree search

- Strategy: expand a **deepest** node first
  - Practically, this means expand the nodes you just put in
  - Last in first out
- Fringe: stack

Properties of DFS

- What does DFT expand?
  - Some left prefix of the tree
  - Could process the whole tree $O(b_m)$
- Space complexity: fringe only has the siblings of the current path to root $O(bm)$
- Complete: **no**, if m is infinite!
- Optimal: **no**, it finds the *leftmost* solution

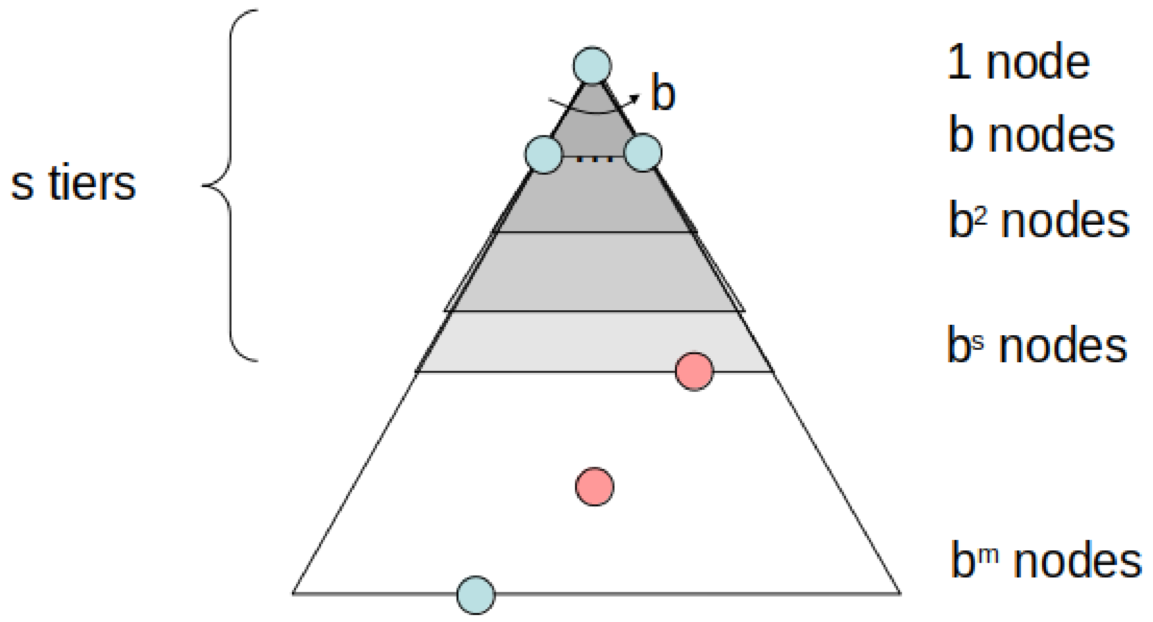


Breadth first search

- Strategy: expand a **shallowest** node first
  - Practically, this means that expand the oldest nodes in the fringe
  - First in, first out
- Fringe: queue

Properties of BFS

- What nodes are expanded?
  - All nodes above the shallowest solution, which is at depth s
  - Search time $O(b^s)$
- Space complexity: fringe can have the last tier, so $O(b^s)$
- Complete: yes, when it reaches the depth s, it will find it
- Optimal: it will find the shallowest solution



Depth first vs. breadth first search

- When will BFS outperform DFS?
  - Solutions are few, but relatively near
  - BFS will find it for sure
  - DFS can get lost, or even stuck in a loop
- When will DFS outperform BFS?
  - Many solutions, but not nearby: finding the ocean from a desert island
  - Important is to keep going, in any direction!

Iterative deepening

- DFS has the advantage of a low spatial complexity. Can we get this advantage with the BFS's shallow-solution advantages?
- Iterative deepening
  - Run a DFS with depth limit 1 - time cost $O(b)$. If no solution...
  - Run a DFS with depth limit 2 - time cost $O(b^2)$. If no solution...
  - ....
- What do we gain: the low space complexity of DFS
- What do we loose: repeated traversal of the upper parts of the tree
  - But for most b, most of the work happens in the last layer.

Cost-based search

- Breadth first search finds the **shortest plan** in terms of number of actions.
- But in many situations different actions have different costs:
  - Road segments have different length - find the **shortest** plan.
  - Some road segments have length + toll - find the **cheapest** plan.
  - Some actions take a different amount of time - find the **fastest** plan.
- Very often we are searching for a plan which has the lowest cost, where the costs are **added up** along the actions in the plan.
  - Other possibilities exist

Uniform cost search (UCS)

- A variant of general tree search
- Assume actions have cost $c(a)$
- For each node n, keep the cumulative cost of actions from the root $g(n)$
- Sort the fringe by $g(\cdot)$
  - Practically: implement the fringe as a priority queue
- Partial plans will be investigated in the order of their cost!

Properties of uniform cost search

- Let us say the cheapest solution has cost C^*. How deep can that solution be?
  - If you have actions with zero cost, it can be infinitely deep!
  - Assume each action has a cost of at least ε
  - Then the deepest it can be is C^*/ε - we call this the **effective depth** of the tree
- Time complexity
  - Process all partial plans with cost less than the cheapest solution
  - Time, exponential like in breadth first search, but this time with effective depth $O(b^{C^*/\varepsilon})$

Properties of uniform cost search (cont'd)

- Space complexity
  - The width of the last tier: $O(b^{C^*/\varepsilon})$
- Is it complete?
  - With some easy assumptions, yes.
  - Assumptions: $\varepsilon > 0$ and C^* finite
- Is it optimal?
  - Yes.

What do we think about UCS?

- Complete and optimal!
- Space complexity problematic
- Can be applied to **anything**, it doesn't use **any** information about the goal.
- Often we know something about the goal:
  - Defeat all the monsters
  - Collect all horcruxes
  - Go to San Francisco with flowers in your hair
- Can we take advantage of what we know about the goal