Search - Lecture 0

Search, knowledge, uncertainty, optimalization, learning, neural network

agent: entity that perceives its environment and acts upon that environment

state: a configuration (initial state)

actions: choices that can be made in a state, Actions(s) (s: state)

transition model: a description of what state results from performing any applicable action in any state
Result(s, a)

state space: the set of all states reachable from the initial state by any sequence of actions

goal test: way to determine whether a given state is a goal state

path cost: numerical cost associated with a given path

frontier: all the possible paths, we haven't taken yet

A Search problem has 5 things

initial state, actions, transition model, goal test, path cost funct

We would like to find the optimal solution

Node - data structure:
- a state
- a parent (a node that generated this node)
- an action (action applied to parent to get this)
- path cost (from initial state to node)

Approach

- Start with frontier = initial state
- Repeat
  - if frontier is empty then not possible ↩ if/else
  - remove a node from the frontier
  - if node contains goal state, return the solut
  - Expand node, add resulting nodes to the frontier ↩ if/else

Drawbacks: you can lose time on repeating states

Revised Approach

- Start with a **frontier** that contains the initial state.
- Start with an empty **explored set**. ←
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove a node from the frontier. ← it is important how we choose
  - If node contains goal state, return the solution.
  - Add the node to the explored set.
  - **Expand** node, add resulting nodes to the frontier if they aren't already in the frontier or the explored set.

Depth first search

DFS

we use a stack

not always optimal

Breadth-first search - BFS

finds the shortest

search algorithm that always expands the shallowest node in the frontier

look at maze.py

Uninformed Search vs. Informed Search

DFS, BFS

DFS no, BFS yes

search strategy that uses problem specific knowledge to find solutions more efficiently

Greedy Best-First Search

search algorithm that expands the node that is closest to the goal, as estimated by a heuristic function h(n)

we need to define it

Manhattan distance

x-y distance

doesn't allways find best route

A* Search

search algorithm that expands node with lowest value of $g(n) + h(n)$

$g(n)$ = cost to reach node
$h(n)$ = estimated cost to goal - Manhattan, etc.

this is optimal if

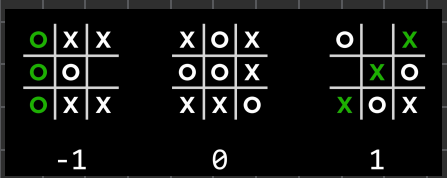
- $h(n)$ is admissible
- $h(n)$ is consistent
  $h(n) \leq h(n') + c$

uses more memory

Adversarial Search

the agent has to work against something eg: tic tac toe

Minimax



- Max (x) aims to maximise score
- Min (x) aims to minimise score

– score

Min Player

Max Player

Recursive

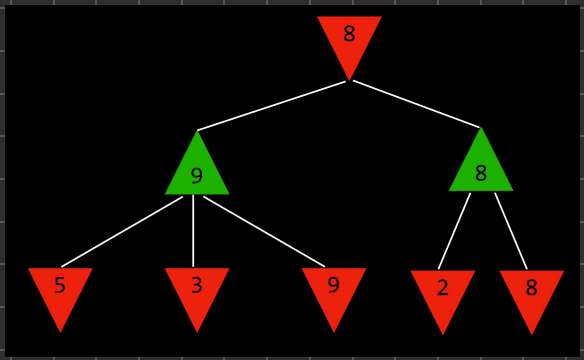
Game

- S_0 : initial state
- PLAYER(s) : returns which player to move in state s
- ACTIONS(s) : returns legal moves in state s
- RESULT(s, a) : returns state after action a taken in state s
- TERMINAL(s) : checks if state s is a terminal state
- UTILITY(s) : final numerical value for terminal state s

empty game array

– this is for the AI

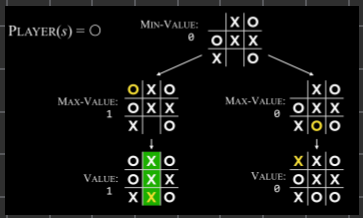
– score



we put ourselves in the opposite players shoes

min - max - min - max ....

recursively



Pseudo Code Minimax

- Given a state *s*:
  - MAX picks action *a* in ACTIONS(*s*) that produces highest value of MIN-VALUE(RESULT(*s*, *a*))
  - MIN picks action *a* in ACTIONS(*s*) that produces smallest value of MAX-VALUE(RESULT(*s*, *a*))

```
function MAX-VALUE(state):
  if TERMINAL(state):
    return UTILITY(state)
  v = -∞
  for action in ACTIONS(state):
    v = MAX(v, MIN-VALUE(RESULT(state, action)))
  return v
```

Min-Value is the opposite

Optimalization – Alpha Beta Pruning

this is inefficient, you need to skip branches, that are worse, than what we found so far.

There are too many states to → Depth-limited Minimax ← this needs an Evaluation Function