I am gonna try to actually learn Deep Learning

Starting with CS50's AI Python course by Harvard.

SEARCH

Mazes, Google maps...

This is used to solve a rubik cube

- Agent → entity that perceives its environment and acts upon that environment
- State → a configuration of an agent in an environment
- Initial state
- Actions → choices that can be made in a state
  - ↳ Actions(s) returns the set of actions that can be executed in ~~an~~ a state s

- Transition model → a description of what state results from performing any applicable action in any state.
  - ↳ Result(s, a) returns the state resulting from performing action a in state s

- State space -> the set of all states we can get from the initial state by any sequence of actions

- Goal state -> way to determine whether a given state is a goal state

- Path cost -> numerical cost associated with a given path

- Optimal solution -> the solution with the lowest path cost

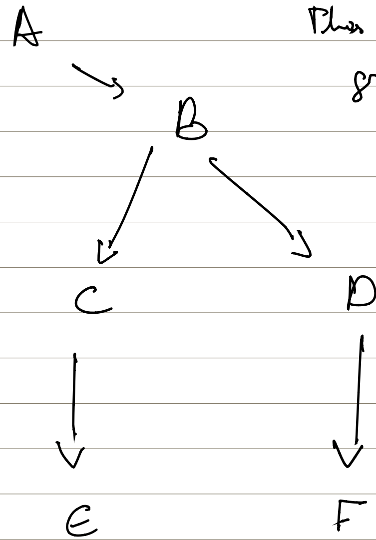
NODE

A data that keeps track of:
- a state
- a parent (node that generated this node)
- an action (action applied to parent to get node)
- a path cost from initial state

- frontier (What we can explore next)

Approach
- Start with a frontier containing only the initial state
- Repeat:
  - If the frontier is empty (no solution)
  - Remove a node from the frontier

- if node contains goal state → solution
- expand node, add resulting nodes to the frontier



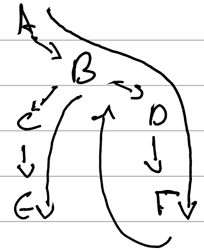
This is a simple node graph

To avoid entering an infinite loop, store the explored states ~~and~~ in a explored set and every time we explore a state first check if it was explored before

- Stack
  Last-in first-out data type (This is important to know ~~data~~ how to store the data in the frontier.

HNB203-57

The version of the algorithm using a stack is called
DEPTH-FIRST SEARCH

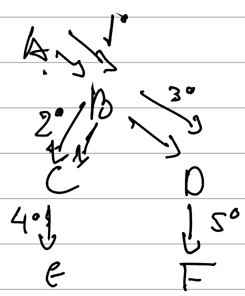


A → E

Basically, look in depth every branch until the end and if no solution move to the last breakpoint

BREADTH-FIRST SEARCH
search algorithm that always expands the shallowest node in the frontier.

Use a queue → first-in first-out



Search branches in parallel, jumping between branches

Code:

- Create a class that defines the node

    This class defines parent, action and state

- Create class StackFrontier

    It has an array with the frontier nodes

    add -> node as parameter and append it.

    contains_state -> checks if the ~~node of a~~ frontier has a particular state. It is a basic for loop but it's written fancily

    return any(node.state == state for node in self.frontier)

    empty -> true if the frontier is empty

    remove -> if it's ... do nothing

    as we're using stack, remove the last one. remove the node from the frontier and return the last node

    to remove
    self.frontier[:-1]

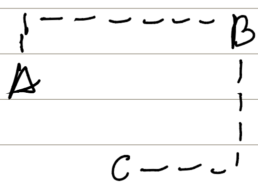
HNB203-57

greedy best-first search

↳ Estimates the closest to the node using a ostic function $h(n)$

for this problem (ignoring the walls) ~~it's the geographically closer~~ the heuristic function is which node is closest

Using Manhattan distance (x and y distance)



$\boxed{\text{Greedy may not make the best solution}}$

A* Search

Picks ~~lower~~ node with the min value of $g(n)+h(n)$

$g(n)$ = cost ~~of~~ to reach node

$h(n)$ = estimated cost of goal (same as before)

- Create QueueFrontier (StackFrontier)

Inherit

only thing that changes is the remove, we remove the first one and we return the first one

to remove self.frontier [1:]

- Create class Maze

#→ Walls
A → initial state
B → goal state

gets the filename

→ solve functions

start = Node (self.start, None parent, no action)

Uninformed search algorithms

No information of the problem

Informed search

→ Strategy that uses knowledge of the problem

HNB203-57

Optimal if

- h(n) is admissable (never overestimates the true cost),
- h(n) is consistent (for every node n and successor n' with step cost c, $h(n) \leq h(n') + c$

> Write the algorithms ~~are~~ easy.. The problem comes when ~~[illegible]~~ choosing the heuristic (h(n))

Adversarial Search

(There's smth opposing the goal of the agent)

for Tic Tac Toe there would be 2 agents playing against each other.

Minimax

Algorithm for games 1v1 like Tic Tac Toe

Lost	Draw	Win
-1	0	1

One agent is the Max (X) tries to win
The other agent is the Min (O) tries to make the other one to lose.

~~Basically [illegible]~~

Basically every possible ending state of the game has a number, the Max agent tries to increase it, the min agent tries to decrease it.

Needs

S0 : 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) : check if state s is terminal

function MaxValue(s) : (Try to make the value as high as possible

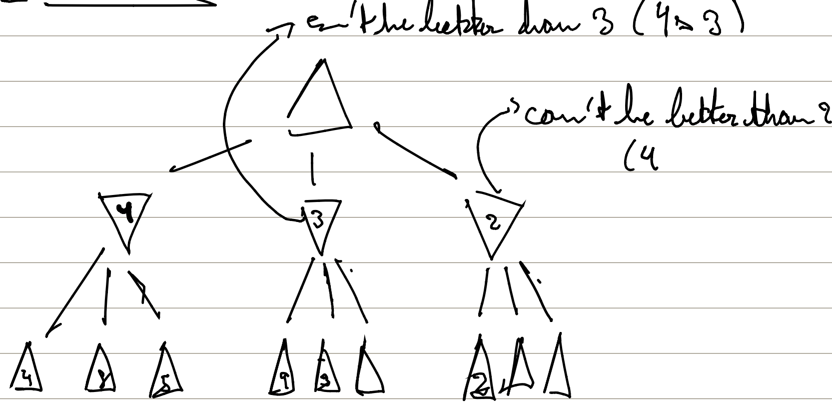
if Terminal(s):
  return Utility(s)

v = -∞
for action in Actions(s)
  v = Max(v, $\underbrace{\text{MinValue(Result(state, action))}}_{\text{Checks the movement of the min player}}$)
return v

function MinValue(s) :

(Opposite of MaxValue)

Optimizations



Instead of checking every possibility, check if ~~there~~ is any other better option, if not change branch

such is called $\boxed{\text{Alpha - Beta Pruning}}$

Depth-Limited Minimax

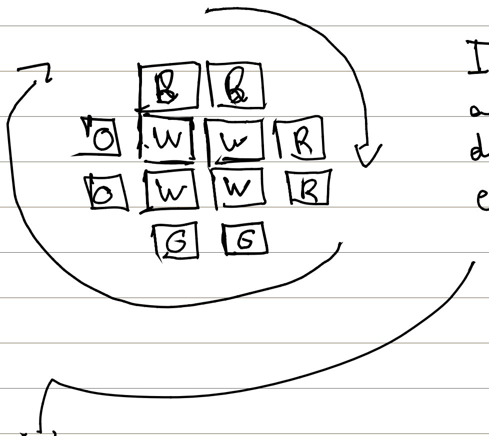
It does not go deep completely into the branches

Evaluation function: estimates the expected utility of the game from a given state

Actions

Let's approach it differently

Actions are gonna be named as the ~~?~~ color of the face ~~???~~ we're turning



I'd like to have a function that does it for every color

For that I need to correlate the faces

W B Y O G B ✗

W R G Y O B ✓

0 1 2 3 4 5

idx(W) = 0

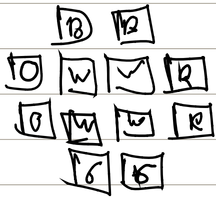
idx(R) = idx(W) + 1 (Right)

for yellow

idx(Y) = 3

idx(O) = idx(Y) + 1

~~This works~~ It does not

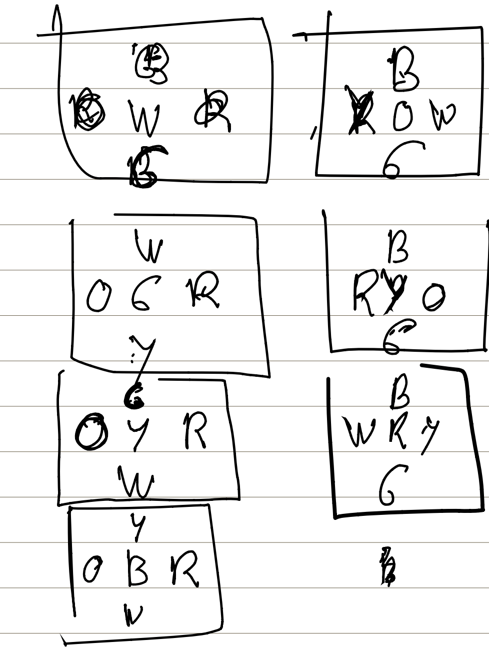


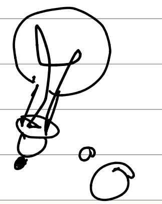
maybe create a dict with all the next faces ~~and~~ to ~~something~~ ~~different~~ if ordered so it's always the same

WRGYOB
0 1 2 3 4 5

0 : [G, B, R, O] 'Front', 'Back', 'Right', 'Orange'
0 : [2, 5, 1, 4]

Rubik solver





The heuristic is the same unless it is in the opposite

H(s) → heuristic

So, if red tile on green face $H(s) = 1$

if red tile in orange face $H(s) = 0$

The H(state) it's the avg of the H(tiles) (every tile)