Lecture 26

Heuristics and AI development Part 2

Introduction to Software Development
Pablo Frank-Bolton

Recall: Formalizing Heuristic Problem-Solving

For a single player game, we may define the problem as a sequence of decisions from a given set. We may define the set of **candidate actions** as:

$$A = \{a_1, \ldots, a_N\}$$

We may call a **Partial Solution**, a sequence of decisions taken from A:

$$\hat{S}: \{s_1, s_2, \ldots, s_k\}$$

We want to define the approximate function **$f(\hat{S})$** that estimates the cost of reaching a goal from the current **state**.

What do we mean by “State”

A state is a possible **configuration of parts and characteristics** that a system might find itself in.

For our purposes, it is the configuration the game is in before your next decision. They can have different complexities:

- In 2048, it is the arrangement of the tiles in the board.
- In Chess, it is the configuration of the board before you move.
- In Space Invaders, it is the position and shape of obstructions, and the positions, velocities, and directions of all the ships and projectiles.

Why so we think about States?

States allow us to think about sets of values that represent something in our system, like a “checkmate” in chess.

It is also the first step in the process of attempting to rethink the system as a state-machine, with states, transitions, start and end states, inputs and outputs.

One crucial concept is the idea of **neighboring states**: those reachable from our current state in one transition.

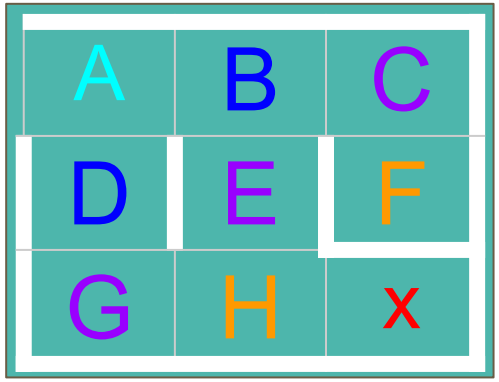
Problem-Solving as a State-Search Problem

It turns out that the strategy of choosing options from a set of candidates in order to find a goal state can be viewed as traversing a **graph of all possible decisions** until you find the correct end state or node.

In this Graph, the root is the starting state (step 0) with $\hat{S}[0]=\{\emptyset\}$. All other nodes are possible intermediate states that can be reached when picking each of the available candidates from A. e.g. A={Down, Right, Up, Left}.

After making one choice, we'd have several partial solutions $\hat{S}_i[1]$;

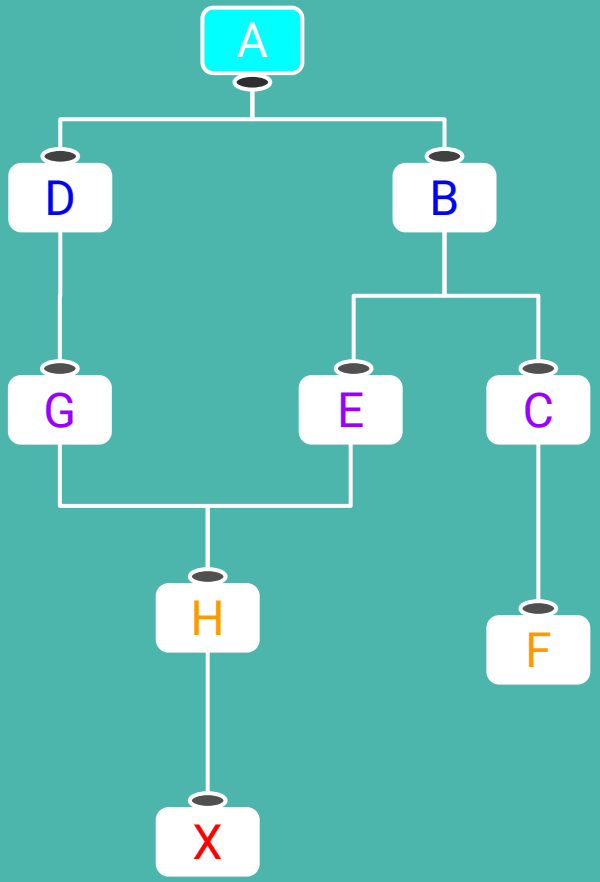
After making two choices, we'd have a set of intermediate solutions $\hat{S}_j[2]$;

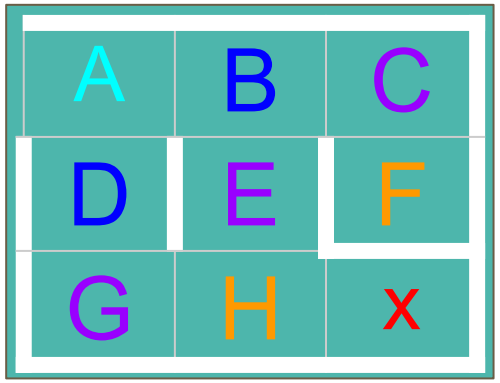


Position Graph

First Visualization

In this example, the Graph shows cell neighborhood.



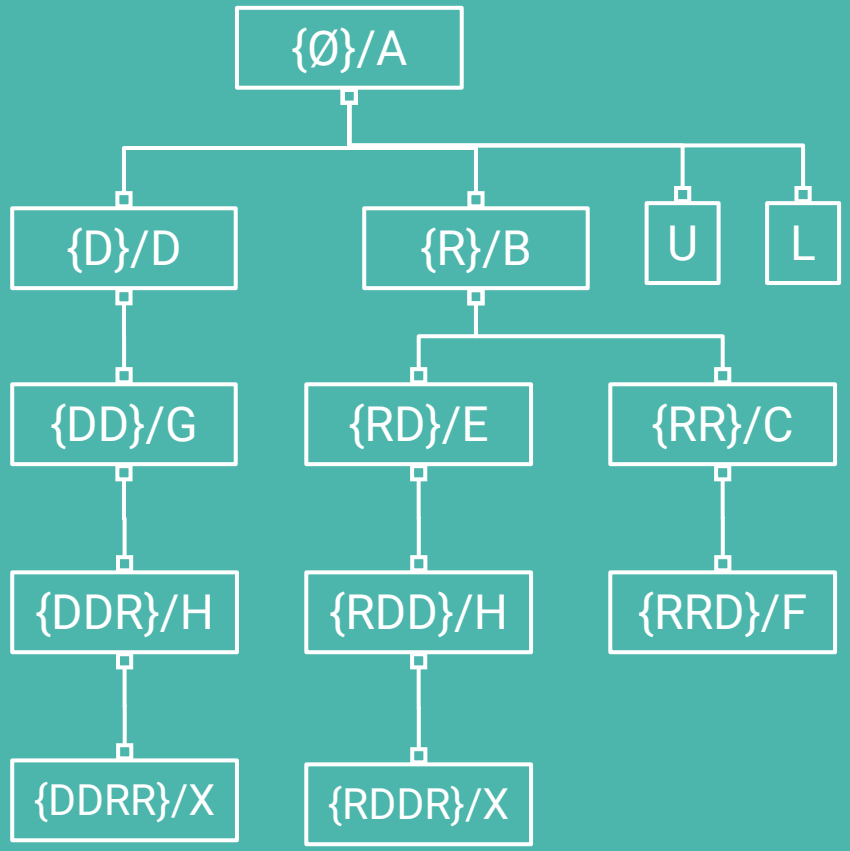


Decision Tree

Tree of decision states

Every Level has 4 options
For Clarity: Only shown in first level

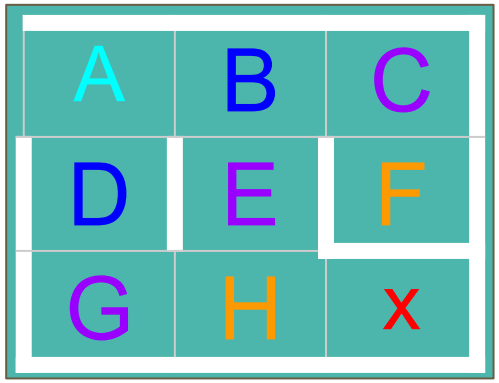
A Solution is a Path from the root to the goal X



$S_1[4] = \{DDRR\}$
$S_2[4] = \{RDDR\}$

Remember the Greedy Method?

1. Design a decent approximation of $f(\hat{S})$
2. Sort the set of allowable next actions according to $f(\hat{S})$
3. Pick the best one (minimum cost or maximum value).

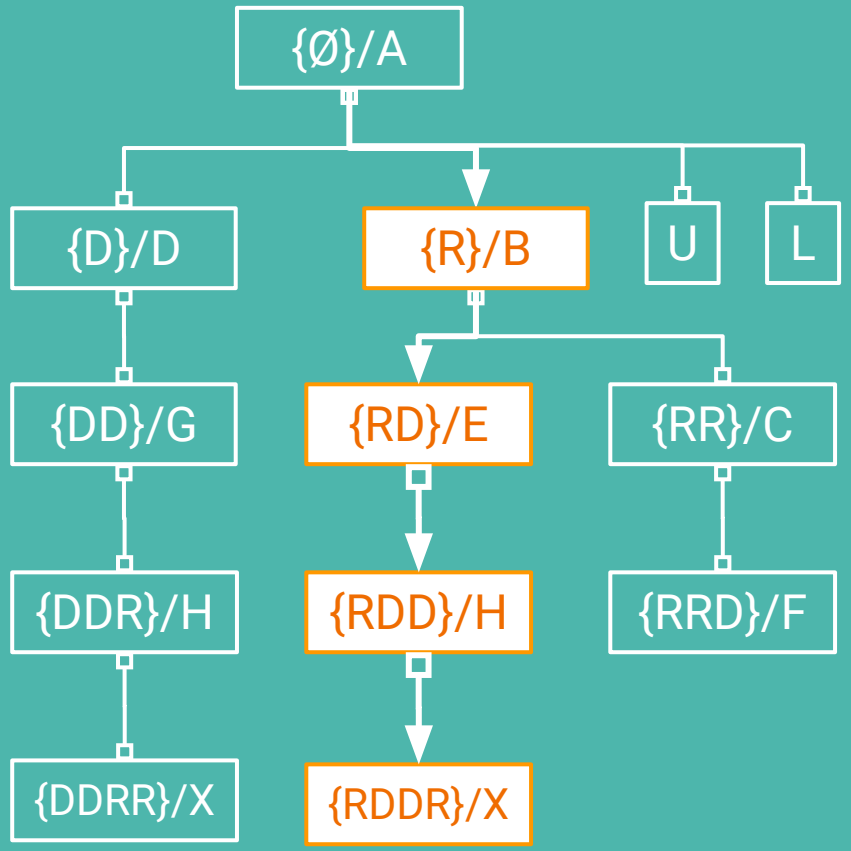


Decision Tree

Greedy Method

Every Level has 4 options
For Clarity: Only shown in first level

Best Greedy uses
$f(\hat{S}) = \text{Sqrt}(\Delta x + \Delta y)$



$S_1[4] = \{DDRR\}$
$S_2[4] = \{RDDR\}$

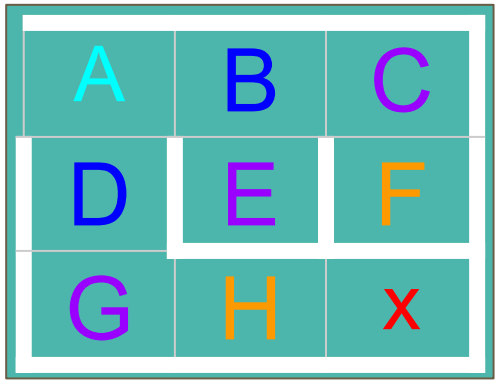
Remember the Greedy Method?

1. Design a decent approximation of f(Ŝ)
2. Sort the set of allowable next actions according to f(Ŝ)
3. Pick the best one (minimum cost or maximum value).

Also, remember that we sometimes might get stuck in a Local Minimum.

The cool part about the search tree is that we **could** figure out how to backtrack from a dead end.

You just need to follow the decision tree back to where a node or state branches into another a promising child option.



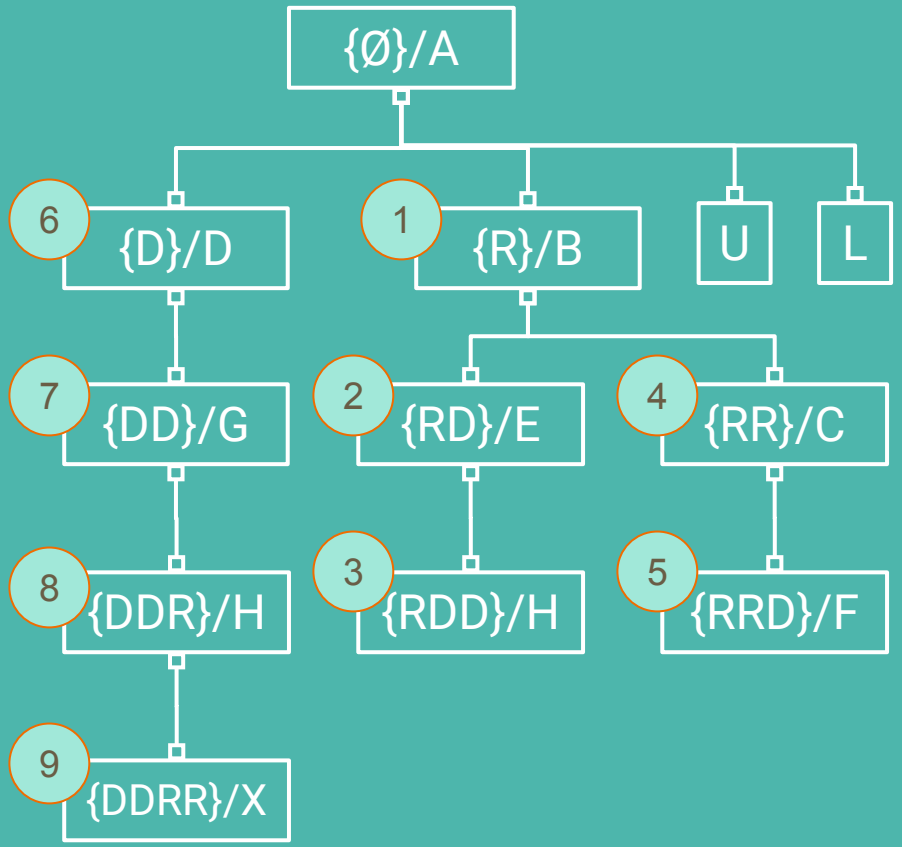
Decision Tree

Backtracking

Every Level has 4 options

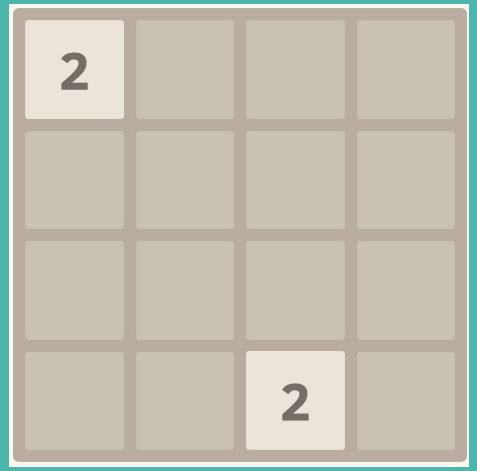
For Clarity: Only shown in first level

Note that there may be many $S_i[1]$, $S_i[2]$, etc.



$S_1[4] = \{DDRR\}$

How about 2048?



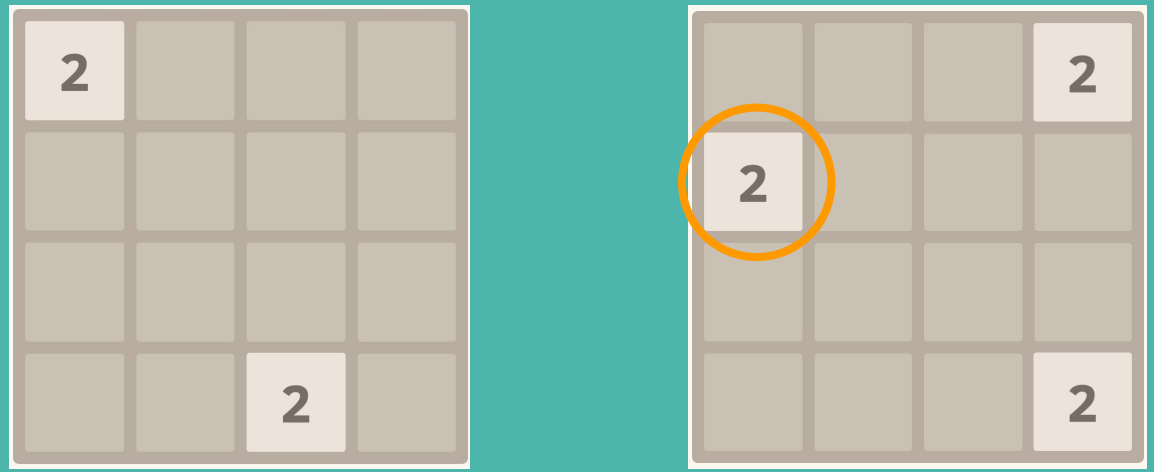
Recall: Types of Games

- Type I: try to maximize or minimize an approximate value
  - 0/1 Knapsack
- Type II: try to reach a specific state by the best possible path
  - Maze
- Type III: try to beat an opponent by reaching a final state where we win
  - Chess

For our purposes, 2048 is a Single-Player Maximization Problem. The only slight difference is that the game also has stochasticity: some part of the game state depends on a random event, in this case, the location and value of the added tile.

Partial Stochastic State

State after a “move Right”



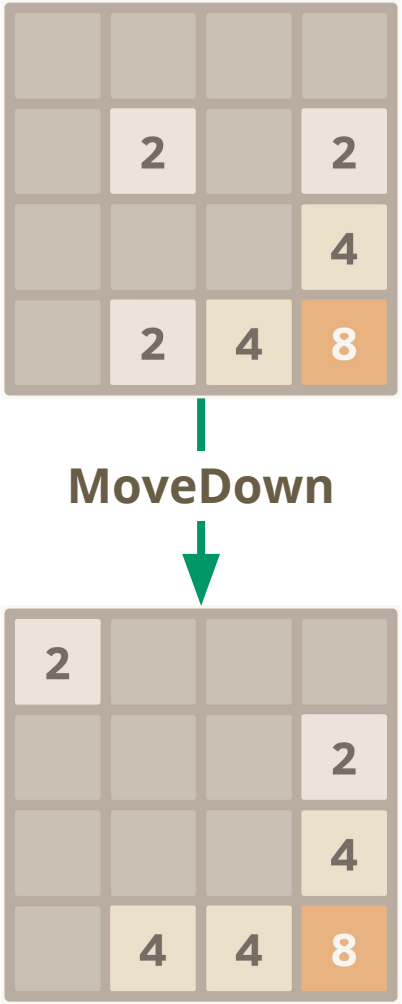
Consequences of the stochastic part:

Note the two intermediate states to the right.

It is important to note that in this game, there is **no backtracking** since we can't undo the stochastic part.

This means we're limited to the basic Greedy Method. You can only go forward.

It is therefore VERY important to take care to define your value function **f(Ŝ)**.



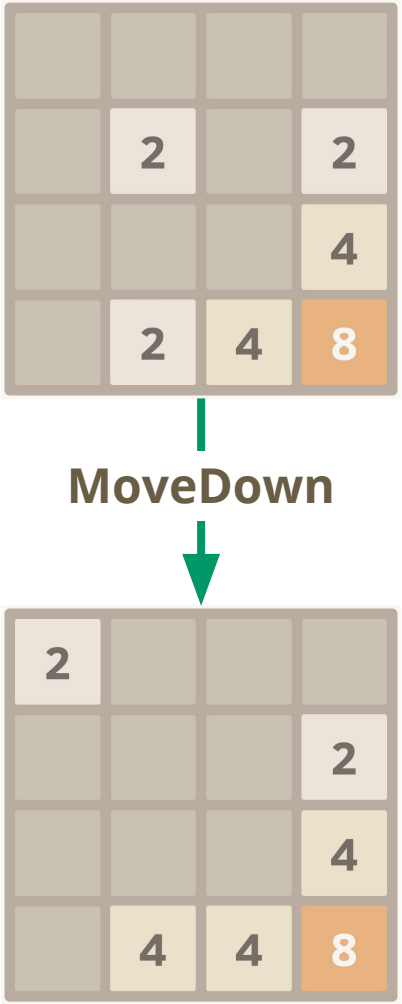
Let's Proceed Formally

- Define a game **state.**
  - Some part of it may be predicted from the previous state
  - Some part may be the random new element (tile location and value)
- Define **A**: the set of **candidate actions** for each round.
- Define **f(Ŝ)**: the **value function** that we want to maximize.

Step 1: Defining “state”

Let's define the state as the configuration of values in a 2D array.

Let’s also define a downstream neighbor as one of the states that we could arrive at after a single move.

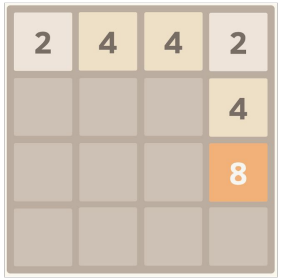


Step 2: Defining Actions

A: {Left, Down, Right, Up}

In the example, the central board is the current state, and the other four boards are their neighbors after each of these movements.

As an example, in all neighbors, the “random” tile is added on the top left. This is NOT predictable.



Step 3: defining f(Ŝ)

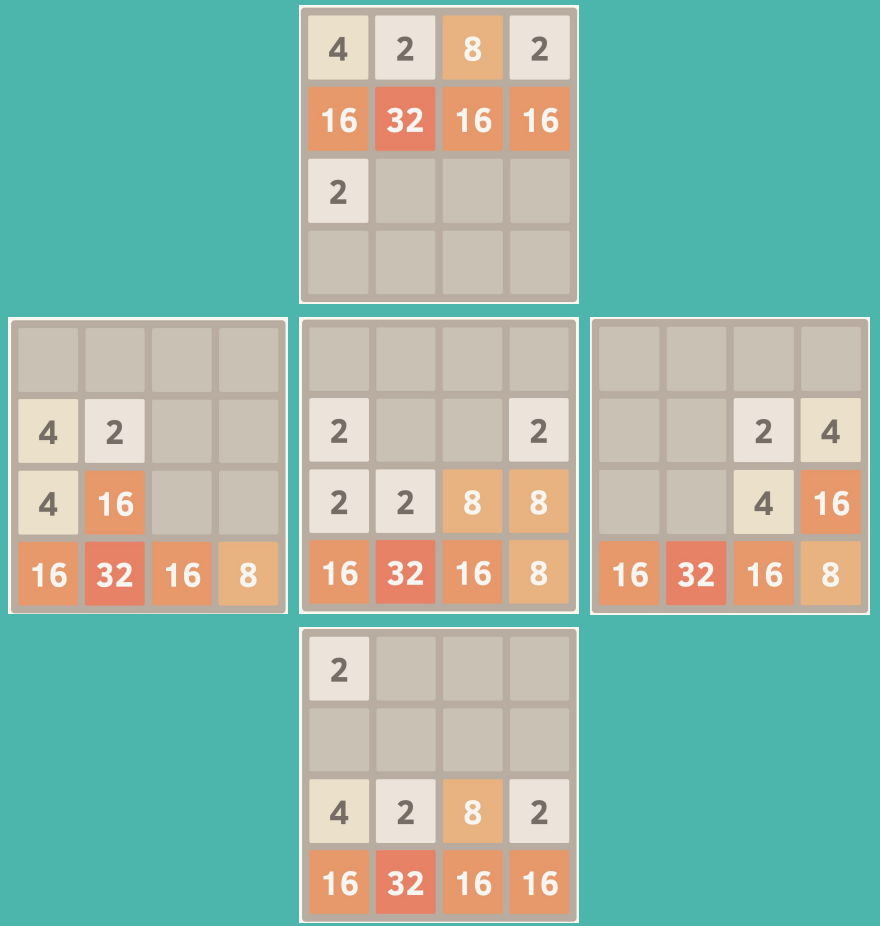
While we cannot just give you a good value function, we'll give you some tips.

- You could maximize the number of points obtained after the next transition.
- You could minimize the number of occupied spaces after the next transition.
- You might want to avoid ever reaching a final state.
- Any move that leaves a single available space COULD lead to a final state.
- Tiles of similar value are closer to being joined than tiles of distant values.

What to do here?

Try different value functions

Choose from:
A: {Left, Down, Right, Up}



Final Notes

In general, when composing a value function to maximize or minimize, you'd do well to construct it in the following way:

$$f(x) = g(x) + h(x)$$

Where:

- g(x) is the cost (or value) so far, and
- h(x) is the proper estimate of the remaining cost (or value) to some goal

h(x) < the actual cost if minimizing, or h(x) > the actual value.

It is always optimistic, otherwise it is NOT an admissible heuristic.