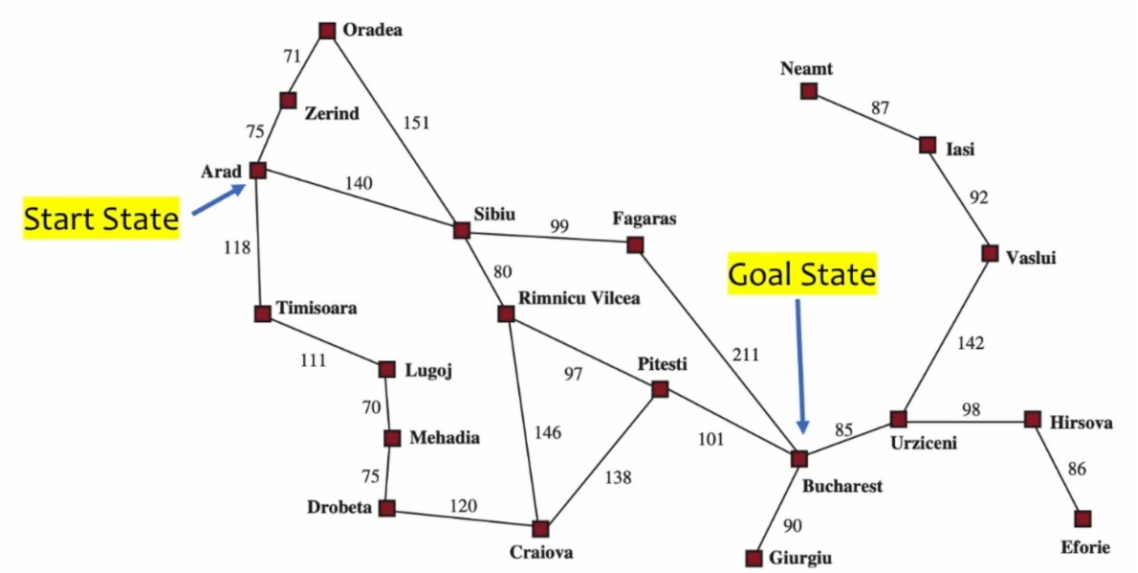
Lecture 2 - Uniform Search Strategies

Problem Formulation: Given a map, pick the best route from *start state* to the *goal state*. For example, check the below:



We start somewhere like Arad, and we want to get to the *goal state*, which, in this case, is Bucharest.

Let's try to look at some other problems that we can solve with the same kind of algorithm. See the below:

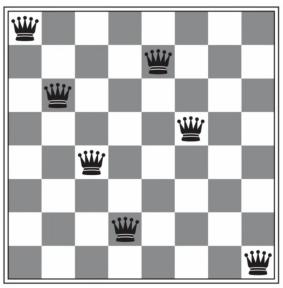
$$\begin{bmatrix} 7 & 2 & 4 \\ 5 & & 6 \\ 8 & 3 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

(start state) (goal state)

This is the Sokoban puzzle. We want to push the boxes to designated locations, with each box moving only to an adjacent empty cell. For example:

In $\begin{bmatrix} 7 & 2 & 4 \\ 5 & & 6 \\ 8 & 3 & 1 \end{bmatrix}$, there are only four possible actions: $2\downarrow, 5\rightarrow, 3\uparrow,$ and $6\leftarrow$. This is almost an equivalent problem to the big map we have above.

Let's look at another problem:



This is the 8-queens problem. We want to place eight queens on a chessboard such that no queen attacks any others. Note that queens can attack each other if they are on the same row, diagonal or column. The solution to this is relatively simple. It's just the last queen that needs to move from the bottom right corner.

There's also another problem involving a vacuum and cleaning some rooms. There are some actions that are available for the agent to do, putting it in a *new* state. Ultimately, we want to reach the final state such that all the constraints are satisfied. Let's try to come up with a standardized problem so that we can represent all of them.

The core concept here is **searching** through the state space. A search problem is defined by 5 items:

1. Initial state
2. Actions: A description of possible actions. Given a state s, Actions(s) returns the set of actions that can be executed in s.
3. Transition model: A description of what each action does, specified by Results(s, a). This returns the state that results from doing an action a in a state s.
4. Goal test: Determines whether a given state is a goal state. GoalTest(s) $\longrightarrow T|F$.
5. Path cost: $c(s, a, s')$ is the cost of a step, assumed to be $\geqslant 0$.

A **solution** is a sequence of actions leading from the intial state to a goal state. The set of all states is call the **state space**. We can formulate the above mentioned problems based on these 5 items. I will not include those, but it's very easy to do. In many cases, the transition model is straight-froward and trivial. But if we are in a non-deterministic environment, then it wouldn't be as easy. The transition model would actually be useful. The function may return probabilities in non-deterministic settings.

Let's demonstrate with the 8-puzzle:

1. States? We have $\frac{9!}{2}$ possible states, because if we start at any state, half of the other states are not reachable. So it's $9 \times \cdots \times 3 \times 2 \times 1$ possibilities over 2.
2. Actions? Actions are based on moving cells to adjacent empty cells. These are actions that are *possible* given the current state
3. Goal test? Only true for the goal state.
4. Path cost? All the actions have a cost of 1.

The 8-queens problem: We have $\frac{64!}{(64-8)!8!}$ possible states. Now, we have a data structure that contains all of this information. What do we do with this?

```
Problem p:
    p.initial_state()
    p.actions()
```

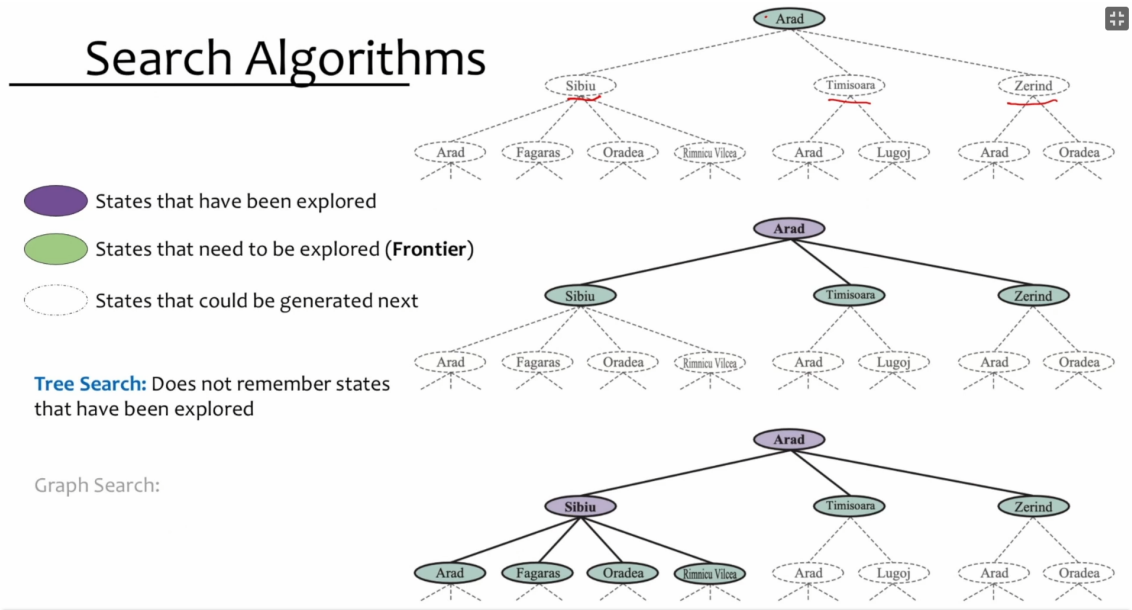
```
p.result()
p.goal_test()
p.cost()
```

Search Algorithms:

A search algorithm takes the search problem as input and returns a solution. The basic idea is:

- Explore the state space **offline** by simulating and expanding the next possible states from already explored ones.
- Offline: We will not actually go and play the game, we will not actually visit cities, etc... We just simulate it.

Let's take a look at this from the perspective of the navigation problem.

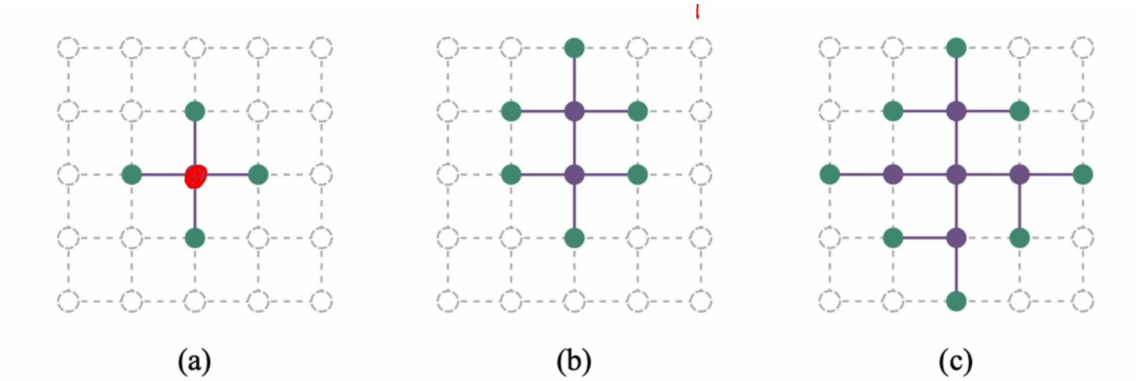


After we go to the top of the tree, there are three possible states that we can explore. This is the **frontier**. Now, we have a data structure called frontier, that contains: `frontier = {Sibiu, Timisoara, Zerind}`. We keep expanding down the tree. The other data structure is the **explored** data structure. At time-step 3, `explored = {Arad, Sibiu}`.

You may notice that we can visit the same state more than once, even though it has already been explored. This is unnecessary, but it is a valid action. If we do not keep track of what states have been explored, i.e. the explored data structure does not exist, then we will keep going back to it.

- **Tree search**: Does not rememebr states that have been previously explored.
- **Graph search**: Remembers states that have been explored, so that it does not visit them again.

Look at the below example of a graph search:



In terms of implementation, let's go back to the tree. Is the Arad at the root the same as the Arad at time-step 3? We can keep track of them in terms of the parents of a node.

```
class Node():
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent # parent of the current node
        self.action = action # action that was taken to reach this node
```

This way, it is very easy to keep track of what each node is / what it represents. For example, `node_Arad = Node(Arad, None, None)`. Keep in mind that the state here is Arad, because that is where we currently are. This is not the same as the node. On the other hand, `node_Sibiu = Node(Sibiu, Arad, go_to_Sibiu)`. The parent here is Arad. Now, let's implement the algorithm according to these.

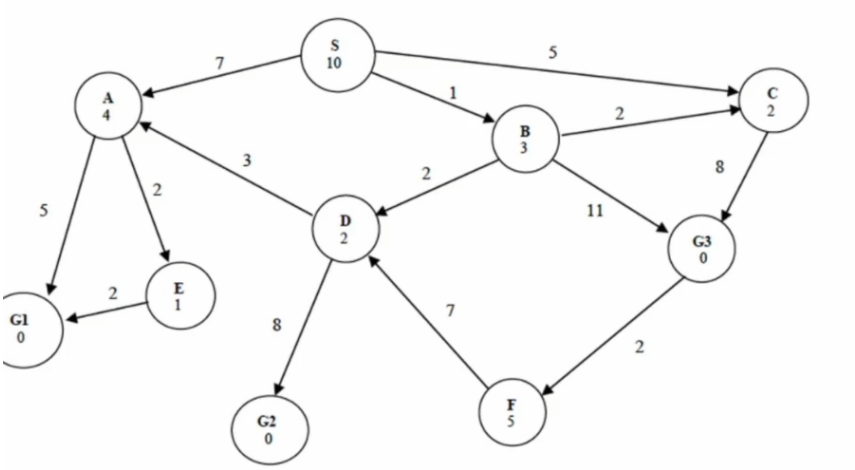
```
Tree_Search(Problem p) returns path:
    frontier = [Node(p.initial_state(), None, None)]
    # At the starting time, we are at the initial node. In tree search, we do
not have an explored. We just remove elements from the frontier once they have
been "explored."

    loop:
        if frontier is empty: return fail

        node = frontier.pop() # Remove an element from the frontier
        state = node.state
        if p.goaltest(state): return node
        for a in p.action(state):
         frontier.push(Node(p.result(state, a), node, a)) # Add the state we
are going to.
```

What about graph search?

```
Graph_Search(Problem p) returns path:
    frontier = [Node(p.initial_state())] # Same as tree search
    explored = {} # At first, we have not explored anything

    loop:
        if frontier is empty: return fail
        node = frontier.pop()
        state = node.state()
        explored.add(state)
        if p.goaltest(state): return node
        for a in p.action(state):
```

```
            if p.result(state, a) not in explored: # Only if that state has not
been in the frontier before
                frontier.push(Node(p.result(state, a), node, a))
```

Let's do this for an example problem:



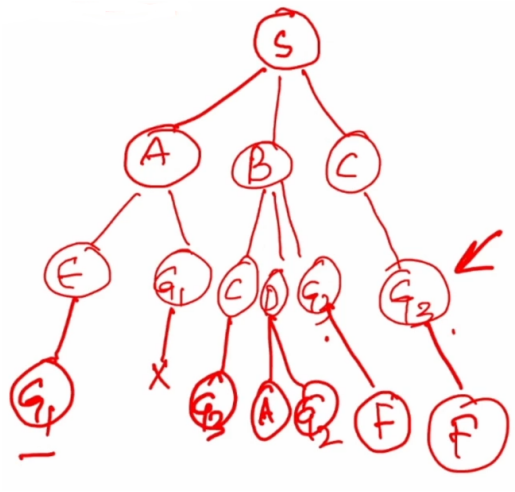
For now, ignore the numbers in the cells and the numbers on the vertices. Let's apply tree search first.

The initial state is S, the goal state is $G2$. We will have a list called frontier.

```
# Time step 1 (Assume alphabetical, always):
frontier = {S} # Then, we remove S and add the three things we can see:

# Time step 2:
frontier = {A, B, C} # Remove S, add A, B and C

# Time step 3:
frontier = {B, C, E, G1} # Remove A, add G1 and E

# Time step 4:
frontier = {C, E, G1, C, D, G3} # Remove B (explore it), add C (repeated), G3,
and D

# Time step 5:
frontier = {E, G1, C, D, G3, G3} # Remove C, explore and add G3 (again)

# Time step 6:
frontier = {G1, C, D, G3, G3, G1} # Explore E

# ...
frontier = {G1, G3, G2, F, F}
```

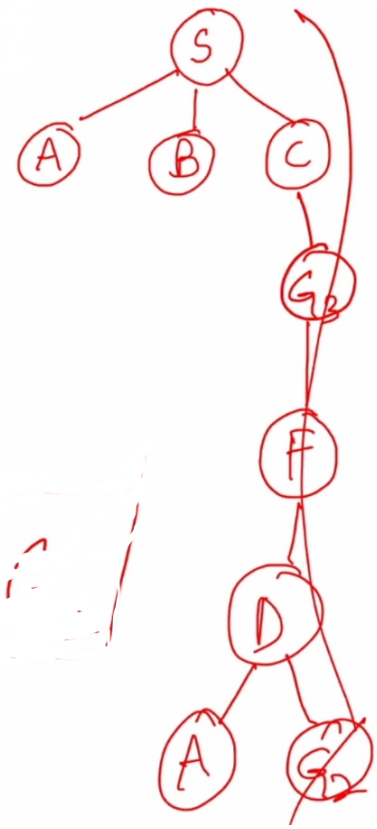
Once we get to $G2$, then we can stop, because it is the goal (in other words, goaltest() would return true).

The frontier is a queue data structure. First in, first out. We can also represent this as a tree:



At this stage, all the goals are in the frontier. At the end, we would get the path $S \to B \to D \to G2$. What kind of traversal is this? If we implement the frontier with a queue, then we would get Breadth-First Search (BFS). What if we used a stack instead?

```
# Time step 1:
frontier = {S}

# Time step 2:
frontier = {A, B, C}

# ...
frontier = {A, B, cross(C), cross(G3), cross(F), cross(D)}
```



Thus, the path returned would be $S \to C \to G3 \to F \to D \to G2$. This would be a Depth-First Search (DFS). Both are valid. The only differences in search strategies is just the order of the node expansion.

	Search Strategies	Frontier
Uninformed	Breadth-First Search	Queue
	Uniform Cost Search (aka Dijkstra's algorithm)	Priority Queue
	Depth First Search	Stack
	Depth Limited	Stack
	Iterative Deepening	Stack

What is a priority queue? In a priority queue, we store not only the node, but also a cost for each of the nodes. The priority would be given to the node with the smallest cost. In that case, we would use the actual cost. In BFS and DFS, we just ignore those. We usually use a heap or a min heap to implement the priority queue. We want to see which one is better than the other (across all algorithms). How do we do that?

- Completeness: Does it always find a solution if one exists?
- Time complexity: How long does it take to find a solution?
- Space complexity: How much memory is needed to perform search?
- Optimality: Does it find a least-cost solution?

Time and space complexity is measured in terms of b, d and m, which represent the maximum branching factor, depth of least-cost solution, and maximum depth of the state space, respectively.

For the problem we had earlier, $b = 3$. For the navigation problem, $b = 4$.

For the navigation problem, $d = 2$ (We needed to go down 2 levels to get the solution). m just means we keep going until we explore everything — what is the size of that?

A solution is complete if it always returns some path. It does not need to look at all the possible states. In other words, a solution exists.

BFS:

- Explores level-by-level. At some point in time, it will have to visit the goal state. It is therefore, complete.
- It is **not** optimal. This is because it does not consider the cost of the paths. What if we had uniform cost? Yes, it is optimal only if the cost is 1.
- The time complexity is $O(b^d)$. To be exact, it is going to be $O(b^{d+1})$, because we need to

go one level below the goal state.

- The space complexity is also $O(b^{d+1})$.

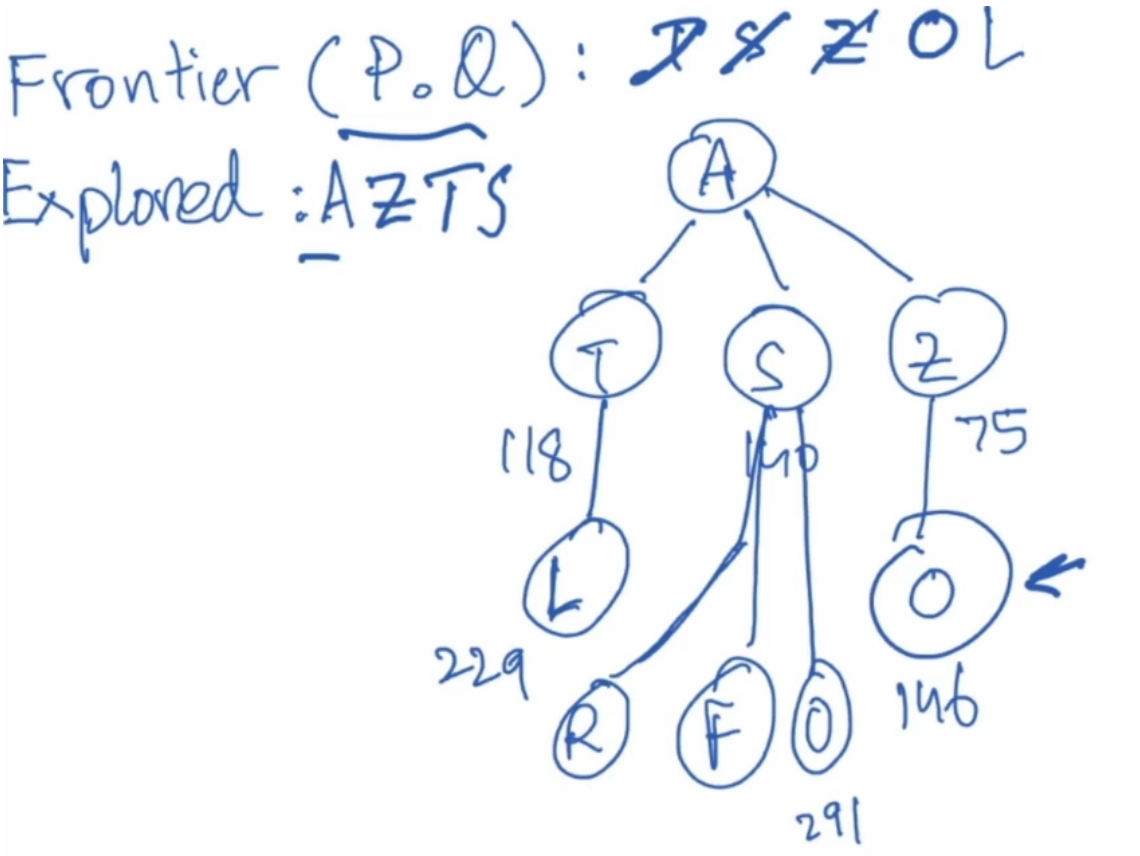
Time and space complexity is exponential. This is a problem. But space is a bigger problem. Why? This is in the slides. Just look at the numbers. We need to modify this algorithm to no longer assume uniform cost. This is where we are introduced to uniform cost search, or **Djikstra's algorithm**.

Dijkstra's Algorithm:

We modify the definition of the node:

```
class Node():
    def __init__(self, state, parent, action, cost):
        self.state = state
        self.parent = parent # parent of the current node
        self.action = action # action that was taken to reach this node
        self.cost = cost # The cost it takes to reach this node
```

Let's run an example, same navigation problem. We will run a graph search here.

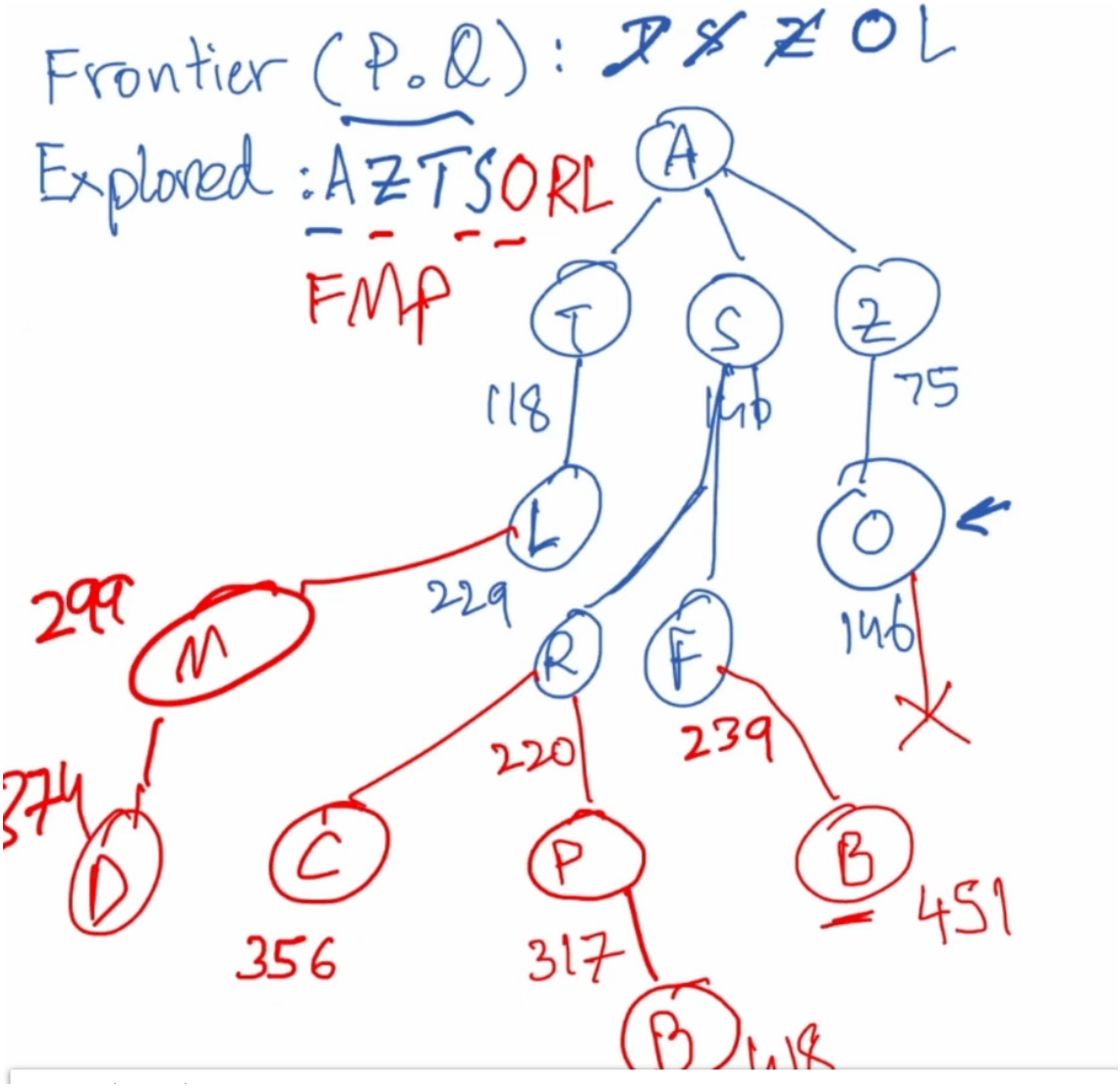


At this stage, we have gotten to the node O through two different paths, but neither have explored them. Therefore, it's valid. Let's compare the costs:

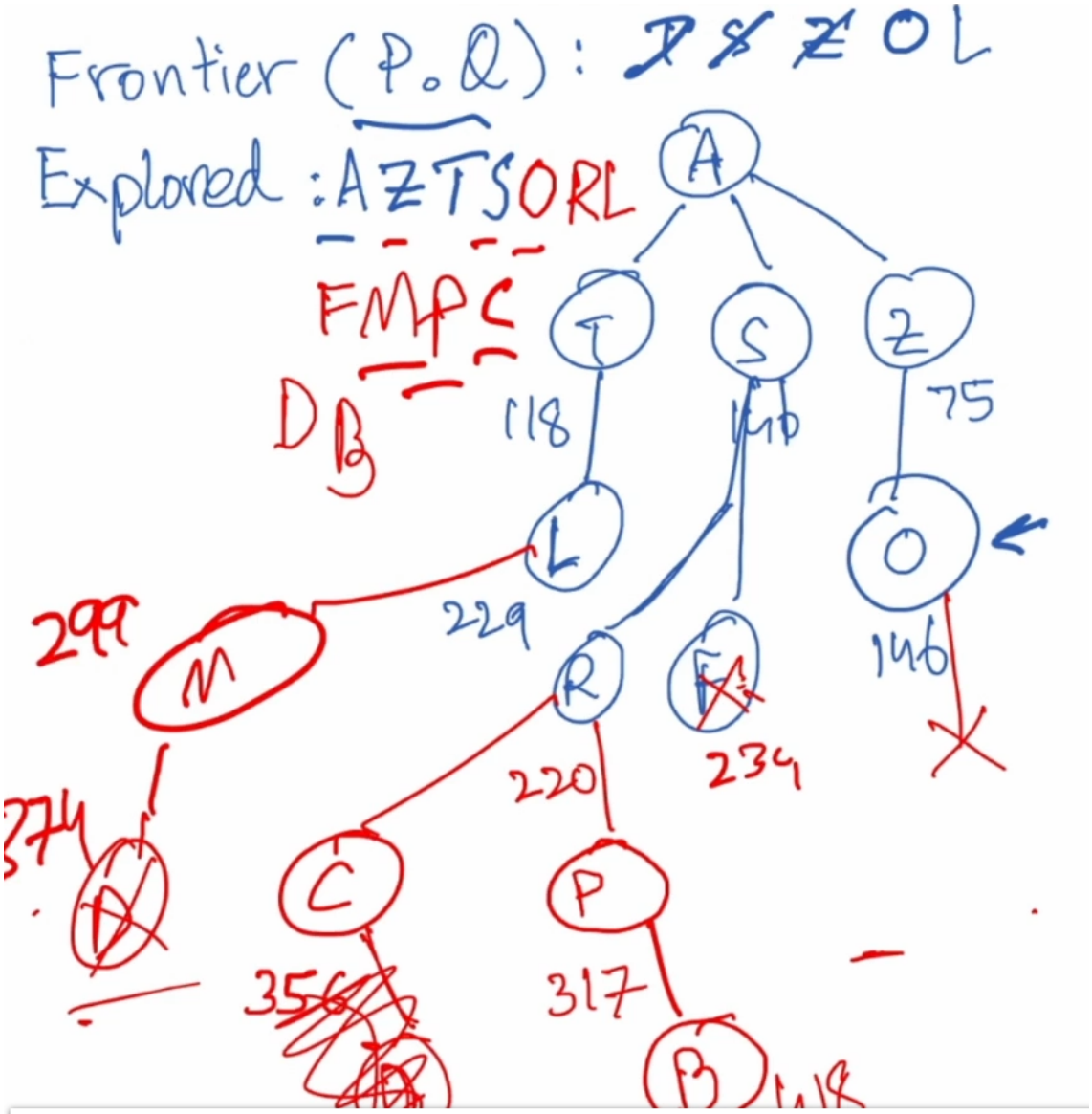
$$A \to S \to O: 291$$

$$A \to Z \to O: 146$$

We need to implement a stack as well. If it is already in the frontier, compare the costs. If it is lower-cost, then add it to the frontier. Otherwise, don't. In this case, we will not add O to the frontier. In the end, the tree, frontier, and explored data structures would look like (remember that we need to go one beyond).



Compare the costs of going from $P \to B$ and from $F \to B$. The former is less, so we remove the latter.



We would have `explored = {A, Z, T, S, O, R, L, F, M, P, C, D, B}`, and the path that would be returned is:

$$A \to S \to R \to P \to B$$

Definitely need to write out some examples for this for it to stick. Let's look at the properties:

- Complete? Yes, if step cost greater than some small positive constant.
- Optimal? Yes, nodes are expanded in the order of increasing path cost.
- Time complexity? $O\left(b^{1+\text{ceil}\left(\frac{C^*}{\epsilon}\right)}\right)$, where C^* is the cost of the optimal solution, and ϵ is the least cost an action can have.
- Space complexity? $O\left(b^{1+\text{ceil}\left(\frac{C^*}{\epsilon}\right)}\right)$

What about DFS?

- Complete? No, it fails in infinite depth spaces or spaces that have loops in them. However, graph search is **complete** in finite spaces.

- Optimal? No.
- Time complexity? $O(b^m)$. This is terrible if m is much larger than d.
- Space complexity? $O(bm)$ for tree search, since it is a linear space. For graph search, the space complexity is the same as BFS, which is $O(b^{d+1})$.

Depth-Limited Search: This is a variant of DFS where a depth limit l is set, and noths at depth l are treated as having no successors (i.e., leaf nodes). However, it is non-trivial to understand what the optimal depth limit l should be. To build on this, we look at **iterative deepening**, which addresses this challenge by systematically exploring all depths, starting from $l = 0$ until we either reach a maximum depth or a solution is found.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Then, there's a bunch of examples that I should probably do. Not now's problem though.