2 Uninformed Search

2.1 Goal-Based Agents

In this case, the agent has a particular goal to achieve. Their environment is fully observable and deterministic. Let us re-consider MopBot example, and we want MopBot to clean such that it satisfies all performance criteria. Let us try to model MopBot example as a goal-based agent and abstract the problem using the following components:

1. State (of Environment): $< Location, Status(A), Status(B) >$
2. Actions: $\{Left, Right, Clean, Idle\}$
3. Transition Model: $g : State \times Action \mapsto State$
4. Performance Measure: $h : State \times Action \mapsto R$
5. Goal State(s): $\{State_1, State_2, \ldots\}$
6. Start State: e.g $< Room\, A, Clean, Dirty >$

Let us assume a static environment, and two location A,B and status *clean,dirty.* Clean is represented as C and dirty as D. Figure 3 shows transition model for MopBot example.

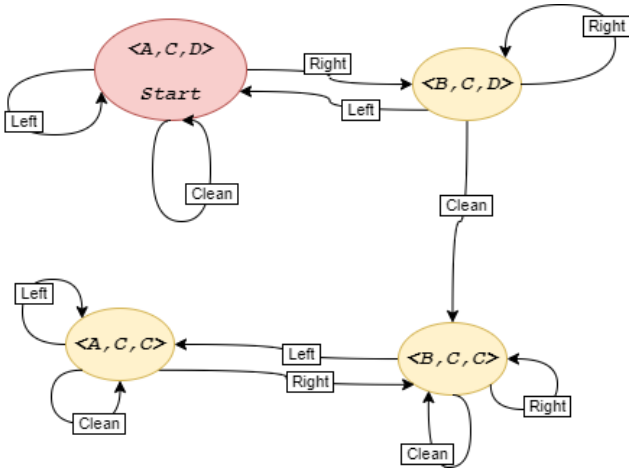

Figure 3: Transition Model for MopBot; A,B,C,D stands for location A, B, Clean and Dirty respectively

Figure 3 represents a graph. The graph is specified implicitly and not explicitly because the explicit graph is often large. Hence, the problem of the goal-based agent can be abstracted as a minimum path graph problem. Let us analyze an example to frame a goal-based agent as a minimum path search in graph problem.

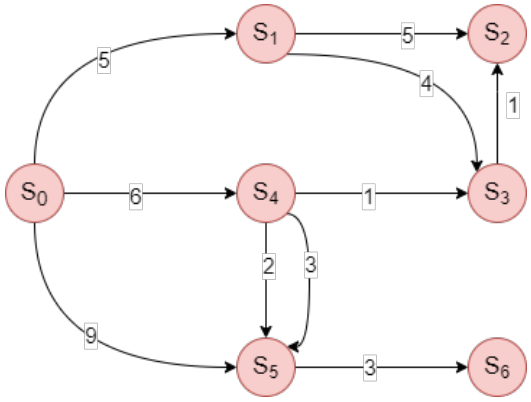

Figure 4: Graph of transition model. Vertex represents state, edge represents an action with the associated cost.

Figure 4 is a transition model in which each vertex represents a state, and an edge represents an action with the associated cost. Let us assume that the start state is S_0 and the goal state is S_2. Note that the goal state need not necessarily be one state. Now, the question is *how to reach from state* S_0 *to* S_2.

One possible way to find the action sequence that would reach the goal state would be to convert all the possible action sequences starting from the initial state into a **search tree** with the initial state at the root. The branch (edge) between parent S_i and child S_j in the search tree represents that there exists an action such that we can reach S_j from S_i. A search tree based on transition model (Figure 4) is as shown in Figure 5.

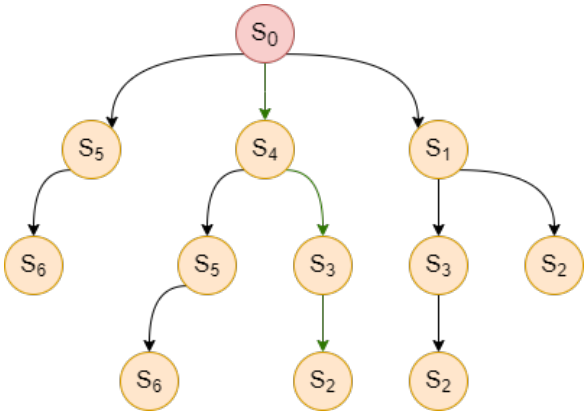

Figure 5: Search Tree corresponding to Figure 4

Now, we can traverse the entire tree from left to right to find a path to reach the goal state (S_2). The algorithm can terminates upon reaching S_2, returning the path taken from S_0 to reach S_2. The algorithm would explore the search tree (Figure 5) in the following order until it reaches S_2:

$$S_0, S_5, S_6, S_4, S_5, S_6, S_3, S_2$$

The solution path it finds would be $< S_0, S_4, S_3, S_2 >$, represented by the green arrows in Figure 5

We want to find an optimal path from S_0 to S_2 in time, memory, and performance measure. However, one of the significant drawbacks of this algorithm is that it is not time efficient. The search path had to explore S_5 and S_6 twice, even though it had already established that those two nodes are a dead end, which will not reach S_2. We somehow want to remember the nodes that the algorithm has already explored to improve in time and memory.

Here, we would like to take the time to think *do we really need a search tree, or we can continue with graph model shown in Figure 4?*.

We can traverse the graph to find a path to S_2 while storing the nodes already been explored to avoid traverse through those nodes again. This would improve time complexity but also requires more memory to implement, thus illustrating a **space-time trade-off** between algorithms.

2.2 Breadth-First Search

One possible way to traverse the graph would be to use Breadth-First Search(BFS). Algorithm 2 represents BFS.

Algorithm 2 Breadth First Search: FindPathToGoal(u)

```
1:  F(Frontier)← Queue(u)                                   ▷ FIFO
2:  E(Explored)← {u}
3:  while F is not empty  do
4:      u ← F.pop()
5:      for all children v of u do
6:          if GoalTest(v) then return path(v)
7:          else
8:              if v not in E then
9:                  E.add(v)
10:                 F.push(v)
11: return Failure
```

Let us try to find an optimal path to reach S_2 in Figure 4 through Algorithm 2. The initial state is S_0. The F and E values during each iteration would be as per Table 4.

Iteration (i)	Internal State of Data Structure
$i = 0$	• $F = [S_0]$ • $E = \{S_0\}$
$i = 1$	• $F = [S_1, S_4, S_5]$ • $E = \{S_0, S_1, S_4, S_5\}$
$i = 2$	GoalTest(S_2) is true, thus return the path to S_2

Table 4: States of Data Structure F and E

Now, as we know an algorithm to find a path from start state to goal state. Let us try to analyze the algorithm through following four criteria:

Completeness An algorithm is called complete if and only if it is guaranteed to find a path from the initial state to the final state whenever it is connected.

Claim 1 *BFS is complete.*

Proof: Let us assume there exists a path from the start state S_0 to goal state S_g, called π of length $k+1$.

$$\Pi : S_0, S_{i_1}, S_{i_2}, \ldots, S_{i_k}, S_g$$

Note that Π is not necessary shortest path from S_0 and S_g.
Let us proof by induction:

Base Case: Path of zero length, i.e $k+1=0$ and $S_0 = S_g$.

Induction Hypothesis: The algorithm will find path to all nodes with path length $\leq k+1$.

Induction Step: Let us assume, we are at S_{i_k}. Either S_g is already explored or S_g will be explored at time of exploring S_{i_k}'s children.

■

Optimality The algorithm will find the path of minimum cost.
BFS is not always optimal. To understand this, let us consider the example based on Table 4. As per Table 4, the path solution returned by BFS would be $< S_0, S_1, S_2 >$ with a path cost of 10. However, there exists a lower cost path $< S_0, S_4, S_3, S_2 >$ with a path cost of 8. Thus, the path returned by BFS is not optimal.

Alternatively, if the goal state was S_6, BFS would choose the following path $< S_0, S_4, S_5, S_6 >$ with a path cost 12. But, another path $< S_0, S_5, S_6 >$ has a lower path cost of 11.

Time Complexity BFS will traverse all nodes in the graph in the worst-case scenario. We can determine the time complexity based on the number of states in the transition model. For a general problem, the number of nodes increases by d for every edge that is travelled away from the initial state node S_0. This occurs for the full depth d of the graph as seen from Fig. 6 below.

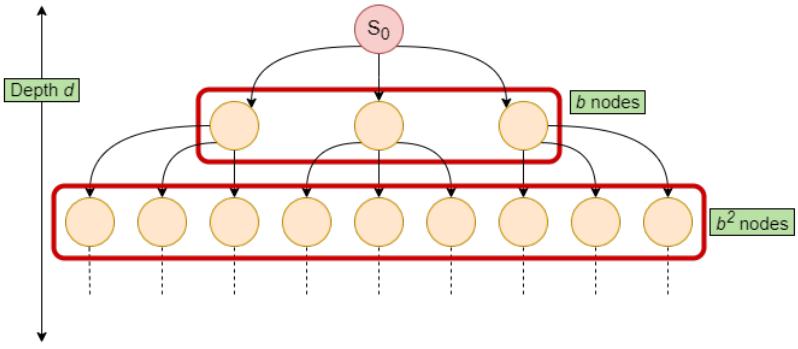

Figure 6: Number of nodes for a graph of depth d and branching factor $b = 3$

Therefore, the time complexity is:

$$O(b + b^2 + b^3 + \cdots + b^d) \leq b^{d+1} \implies O(b^{d+1})$$

Space Complexity The memory used for data structure tracking which node has been explored keeps track of all nodes in the worst case. Therefore the space complexity for E is $O(b^{d+1})$.

In the worse case, the space complexity for the queue is when it has to track the outgoing child nodes of nodes at depth level $d-1$, which means that it has to store all b^d nodes of the last level of the graph in the worse case. Thus, space complexity for F is $O(b^d)$.

2.3 Breadth-First Search with Priority Queue

The BFS algorithm does not always produce the optimal action sequence as a solution; let's modify the algorithm to use a Priority Queue rather than a Queue for data structure F to resolve this problem. Algorithm 3 represents the modified algorithm.

Algorithm 3 Breadth First Search with Priority Queue: FindPathToGoal(u)

```
1:  F(Frontier)← PriorityQueue(u)                    ▷ lowest cost node out first
2:  E(Explored)← {u}
3:  while F is not empty do
4:      u ← F.pop()
5:      for all children v of u do
6:          if GoalTest(v) then return path(v)
7:          else
8:              if v not in E then
9:                  E.add(v)
10:                 F.push(v)
11: return Failure
```

The modified algorithm is such that when the node u is picked from F, node u is the minimum cost node that has not explored in the queue at that iteration. Let us analyze if using a priority queue helps to overcome the optimality issue. We would be considering the two counterexamples discussed earlier:

Counter-example 1 (S_0 to S_2): The BFS with priority queue produces a new solution of $< S_0, S_1, S_3, S_2 >$ with a path cost 10, which is still sub-optimal compared to the optimal path cost of 8 via $< S_0, S_4, S_3, S_2 >$.

Counter-example 2 (S_0 to S_6): The BFS with priority queue was able to produce the improved and optimal solution of $< S_0, S_4, S_5, S_6 >$ with a path cost of 11 (compared to the path cost of 12 for the BFS Algorithm 2).

Although the modified BFS algorithm can get the optimal path for the second counterexample, it still fails for the first counterexample. Therefore, it does not always find an optimal path. This is because it immediately returns the path to the goal state node upon exploring it, which is too early. Moreover, by adding various nodes into the "explored" data structure, the algorithm rules out any alternative path that better than the current path. In particular, there are two main issues with the approach:

1. Checking for goal too early.
2. Putting nodes in the explored set too early.

Let us try to fix the issues mentioned earlier with BFS. The main idea is to track the minimum path cost of reaching goal u discovered so far. The algorithm is known as Uniform Cost Search(UCS).

2.4 Uniform Cost Search

Uniform Cost Search (UCS) Algorithm is described in Algorithm 4.

Algorithm 4 Uniform Cost Search(UCS): FindPathToGoal(u)

```
1:  F(Frontier)← PriorityQueue(u)                    ▷ lowest cost node out first
2:  E(Explored)← {u}
3:  ĝ[u] ← 0
4:  while F is not empty do
5:      u ← F.pop()
6:      if GoalTest(u) then
7:          return path(u)
8:      E.add(u)
9:      for all children v of u do
10:         if v not in E then
11:             if v in F then
12:                 ĝ[v] = min(ĝ[v], ĝ[u] + c(u, v))
13:             else
14:                 F.push(v)
15:                 ĝ[v] = ĝ[u] + c(u, v)
16: return Failure
```

Let us try to analysis the UCS Algorithm 4.

Completeness: UCS algorithm is not complete. Let us consider the case of an infinite number of zero-weight edges deviating away from the goal. The UCS algorithm will not find a path to the goal in such a case. On the contrary, if every edge cost $\geq \epsilon$ (a very small constant), the algorithm will find a path to the goal. As even if infinite such edges deviate from the goal, the summation will eventually be greater than the edge cost between start and goal.

Optimality: UCS algorithm is optimal. The main reason behind this, whenever we are performing *GoalTest* on a particular node, we are always sure that we have found the optimal path to that node.

Claim 2 *When we pop u from F, we have found its optimal path.*

Proof: Let us use the following notations:

- $g(u)$ = minimum path cost from start to u.
- $\hat{g}(u)$ = estimated minimum path cost from start to u by UCS so far.
- $\hat{g}_{pop}(u)$ = estimated minimum path cost from start to u when u is popped from F.

Let the optimal path from start to goal u be:

$$S_0, S_1, S_2, \ldots, S_k, u$$

We will proof the claim using induction.

Base case: $\hat{g}_{pop}(S_0) = g(S_0) = 0$ is trivial.

Induction Hypothesis: $\forall_{i \in \{0,\ldots,k\}} \ \hat{g}_{pop}(S_i) = g(S_i)$.

Induction Step: As we know by definition:

$$g(S_0) \leq g(S_1) \leq \ldots g(S_k) \leq g(u) \tag{1}$$

$$\hat{g}_{pop}(u) \geq g(u) \tag{2}$$

Therefore, by equation 1 and 2, $\hat{g}_{pop}(u) \geq g(u) \geq g(S_k)$
Since, we have $g(u) \geq g(S_k)$, it is guaranteed that S_k is popped before u.
Now, when S_k is popped,$\hat{g}(u)$ is computed as follows:

$$\begin{aligned}
&\hat{g}(u) = min(\hat{g}(u), \hat{g}(S_k) + C(S_k, u)) \\
&\hat{g}(u) \leq \hat{g}(S_k) + C(S_k, u) \\
&\quad \leq g(S_k) + C(S_k, u) && \text{From induction hypothesis} \\
&\quad \leq g(u)
\end{aligned}$$

The estimate of u is always greater than or equal to the value of actual minimum path cost to u along the path $\{S_0, S_1, \cdots, S_k, u\}$.

$$\hat{g}(u) \geq g(u) \tag{3}$$

Thus, by equation 2 and 3

$$\hat{g}(u) = g(u) \tag{4}$$

Furthermore, since $\hat{g}_{pop}(u) \leq \hat{g}(u)$ (note that the value of $\hat{g}(u)$ can only decrease). Also, we know that $\hat{g}_{pop}(u) \geq g(u)$, therefore, we have $\hat{g}_{pop}(u) = g(u)$.

■

Time and Space Complexity : $O(b^{1+d})$

UCS is guided by path cost, assume every action costs at least some small constant ϵ and the optimal cost to reach the goal is C. The worst-case time and space complexity is $O(b^{1+\frac{C^*}{\epsilon}})$ — here 1 is added because UCS examines all the nodes at the goal's depth to see whether they have a lower cost. When all steps are equal, the time and space complexity is just $O(b^{1+d})$.

We have optimality and completeness with every edge cost $\geq \epsilon$ in the UCS algorithm, but it takes too much space. We keep all the nodes in the frontier. However, can we first go into depth? Yes, we can do a depth-first search, and in this case, we also don't need a priority queue. We can use last-in-first-out (Stack). Let us try to understand Depth First Search(DFS) in detail.

2.5 Depth First Search

The key idea in DFS is always expanding the deepest node in the current frontier of the search tree. *GoalTest* is conducted on the node when selected for expansion. If the expanded node has no children, then it is removed from the frontier. Algorithm 5 is for DFS.

Algorithm 5 Depth First Search(DFS): FindPathToGoal(u)

```
1: F(Frontier)← Stack(u)
2: E(Explored)← {}
3: while F is not empty do
4:     u ← F.peek()
5:     if GoalTest(u) then
6:         return path(u)
7:     if HasUnvisitedChildren(u) then
8:         for all children v of u do
9:             if v not in E then
10:                F.push(v)
11:                E.add(v)
12:    else
13:        F.pop()
14:        E.add(u)
15: return Failure
```

Let us discuss DFS algorithm.

Completeness It depends on the search space. If the search space of your algorithm is finite, then DFS is complete. However, if there are infinitely many alternatives, it might not find a solution.

Optimality DFS is not optimal.

Time Complexity For graph search, the time complexity for DFS is bounded by state space $O(V)$ where V is a set of vertex, which may be infinite. However, tree search may generate finite $O(b^m)$ nodes in the search tree, m is the maximum depth of any node, and b is the branching factor. (Still, m might be infinite itself).

Space Complexity $O(bm)$, where b is the branching factor and m is the maximum depth possible.

Hence, it turns out that there is a main trade-off with the DFS algorithm. We are losing in terms of completeness, optimality, and we have significant improvement in terms of space.

So far, we have discussed about BFS 3, UCS 4 and DFS 5. There was one thing common in all three discussed algorithms; we don't have any information about the goal in all these algorithms. The type of search in which we don't have any information about the goal is known as an uninformed search. The obvious question we may have now is *Can the information about goal help?*.
We will discuss *informed search* next.