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, we want MopBot to clean such that it satisfies all performance criteria. Let us try to model MopBot example as goal based agent, and abstract the problem using 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 static environment, and two location A,B and status *clean,dirty*. Clean is represented as C and dirty as D. Figure 3[3] shows transition model for MopBot example.

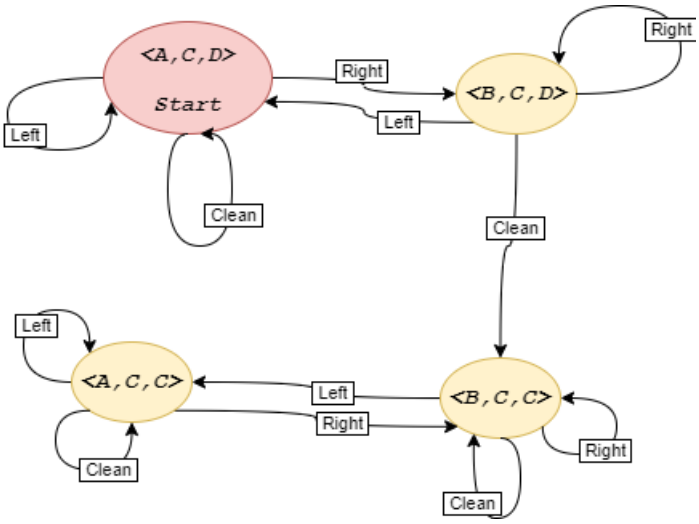

Figure 3: Transition Model for MopBot; A,B,C,D stands for location A, B, Clean and Dirty respectively

Figure 3 represents a graph. The graph is specified implicitly, and not explicitly, because the explicit graph is very often large. In general, we specify the complexity of a graph in terms of:

- *b:* Branching Factor: Maximum number of successors of any node.
- *d:* Minimum length of path from start to goal.
- *m:* Max length of path from start to any state in the graph.

[3]Special thanks to Khiew Zhi Kai's class scribe.

- Performance Measure: Minimum cost of the path.

Hence, the problem of goal-based agent can be abstracted as a minimum path graph problem. Let us try to analysis an example to frame goal bases agent as minimum path search graph problem.

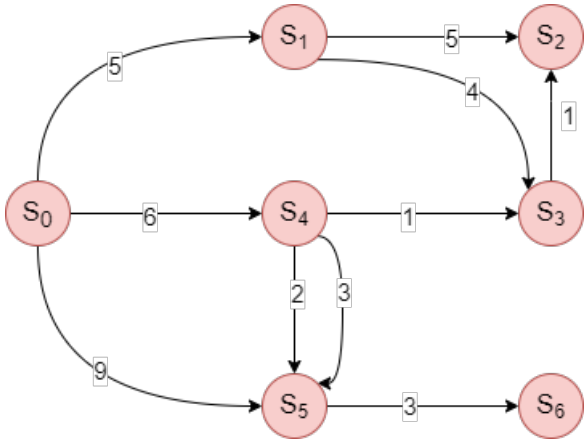

Figure 4: Graph of transition model. Vertex represents state, edge represents an action with associated cost.

Figure 4[4] graph of transition model, each vertex represents state and an edge represents an action with associated cost. Let us assume that start state is S_0 and goal state is S_2. Note that goal state need not necessarily to 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 sequence starting from the initial state into a **search tree** with the initial state at the root; the branches as the possible actions taken at that state and the subsequent child nodes corresponding to states after that action is taken in the state space. A tree based on transition model in Figure 4 is as shown in Figure 5. Each node of graph contains the information of its state, parent, action to generate the node, path cost.

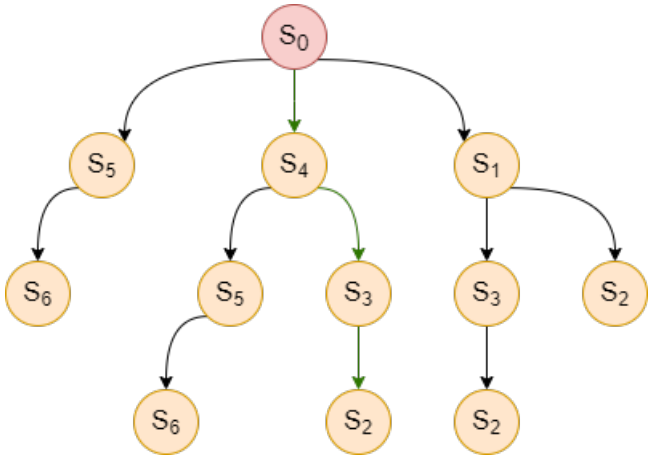

Figure 5: Search Tree corresponding to Figure 4

[4]Taken from Khiew Zhi Kai's class scribe.

To find the solution path to S_2, a tree search algorithm could be implemented to traverse the entire tree from the left to the right, and terminates upon reaching S_2, returning the path taken from S_0 to reach S_2. The algorithm would explore 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 terms of time, memory and performance measure. However, on of the major drawbacks of this algorithm is that it is not time efficient. Based on the search path, it had to explore S_5 and S_6 twice, even though it had already established that those two nodes are a dead end that will not reach S_2. Indeed, if there was a way to remember the nodes that the algorithm had already explored before, there would not be a need to explore S_6 a second time, thus improving the time and memory complexity.

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

We can traverse the graph to find a path to S_2, while storing the nodes already explored so as to not traverse through those nodes again. This would improve time complexity, but also requires more memory to implement, thus illustrating a **space-time tradeoff** 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 see the Figure 4 through Algorithm 2. The initial state is S_0. Let us see F and E values during each iteration would be as follows:

Now, as we know an algorithm to find a path from start state to goal state. Let us try to analysis the algorithm through standard 4 criteria.

Completeness An algorithm is called complete if and only if it is guaranteed to find a path from initial state to final state, whenever initial and final state 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 $k+1$.

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

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

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 where 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 path of minimum cost.
BFS is not always optimal. To understand this, let us consider the example based on Table 4, the path solution returned by BFS would be $< S_0, S_1, S_2 >$ with a path cost of 10. But a lower cost path would be $< S_0, S_4, S_3, S_2 >$ with a path cost of 8. Thus the returned path 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 worse 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 increase by a factor of d for every edge 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.

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 has to keep track of all nodes in the worse case. Therefore the space complexity for E is $O(b^{d+1})$.

The space complexity for the queue in the worse case 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 $O(b^d)$.

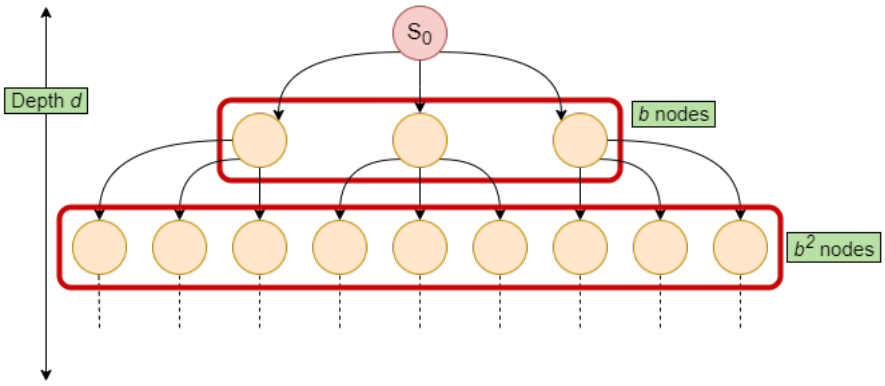

Figure 6: Number of nodes for a graph of depth d and branching factor $b = 3$

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 so as to resolve this problem. Algorithm 3 represents the modified algorithm.

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

```
F(Frontier)← PriorityQueue(u)                ▷ lowest cost node out first
E(Explored)← {u}
while F is not empty  do
    u ← F.pop()
    for all children v of u do
        if GoalTest(v) then return path(v)
        else
            if v not in E then
                E.add(v)
                F.push(v)
return Failure
```

The modified algorithm is such that when the node u is picked from F, then node u is the minimum cost node that is not explored in the queue at that iteration. Does it work in helping to find the optimal solution to the two counter examples raised against BFS for the optimality property?

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 o 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).

Although the modified BFS algorithm is able to get the optimal path for one counter-example, it still fails to do so for the first counter-example, and therefore does not qualify for the optimality property.

This is because it returns the path to the goal state node immediately upon exploring the goal state node, which is too early. Moreover, by adding various nodes into and "explored" data structure and not explore them again, the algorithm rules out any alternative path that may be better than the current path, which is writing them off too early. In particular, there are two main issues with this approach:

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

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

2.4 Uniform Cost Search

Algorithm is described in Algorithm 4. Let us try to analysis the UCS 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
```

Completeness: UCS algorithm is not complete. Let us consider the case of infinite number of zero weight edges deviating away from goal, UCS algorithm will not be able to find the path to goal. On the contrary, if every edge cost $\geq \epsilon$ (small small constant), then even if there are infinite such edges deviating away from goal, there summation will eventually be greater than edge cost between start and goal, and algorithm will find a path to 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.
When we popped $S_k, \hat{g}(u)$ 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 smaller than or equal to the value of actual minimum path cost to u along the path $\{S_0, S_1, \cdots, S_k, u\}$.

$$\hat{g}_{pop}(u) \leq \hat{g}(u) \leq g(u) \tag{3}$$

Thus, by equation 2 and 3

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

■

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 goal is C, The worst case time and space complexity is $O(b^{1+\frac{C^*}{\epsilon}})$. 1 is added because UCS examines all the nodes at the goal's depth to see if one has a lower cost instead of stopping as soon as generating a goal. 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 UCS algorithm, but it is taking too much space. We keep all the nodes in the frontier, can we first try to go into depth? Yes, we can do depth first search, and in this case we don't need priority queue also. We can simply 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 expands 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 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, time complexity for DFS is bounded by state space $O(V)$ where V is set of vertex, which may be infinite. However, for tree search, may generate finite $O(b^m)$ nodes in 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 are main trade-off with 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 algorithm; in all there algorithms, we don't have any information about the goal. These types of search, where we don't have information about the goal is known as un-informed search. The obvious question, we may have now is *Can information about goal help?*. Let us discuss one informed search algorithm called A^* algorithm.