CS3243 **August 2020**

Lecture 2: August 19

Lecturer: MEEL, Kuldeep S./Daren Ler Shan Wen *Scribes: Song Qifeng*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

2.1 An analysis of Goal-Based Agents: the mopot

Definition 2.1 *Goal-Based Agents are agents that have specific targets or goals that they need to achieve. A goal-based agent uses searching and planning to act in the most efficient solution to achieve the goal.*
For the sake of simplicity at this stage, we will take a look at the example of mopot and assuming it is Deterministic, Fully-Observable, and Discrete.

Figure 2.1: the environment of the mopot

6 aspects of Goal-Based Agents:

- State: <location, status(A), status(B)>
- Actions: {Left, Right, Clean, Idle}
- Transition model: g: State $\times$ Actions $\rightarrow$ State
- Performance measure (aka cost function): h: State $\times$ Actions $\rightarrow \mathbb{R}$
- Goal state: <A/B, clean, clean >(both room A and room B are clean)
- Start state

2.2 Modeling the problem as a graph

In section 2.1, we have abstracted the state of mopot into the form of
<location, status(A), status(B)>.
Now we can consider each state as a node with 3 attributes-location, status(A), and status(B).

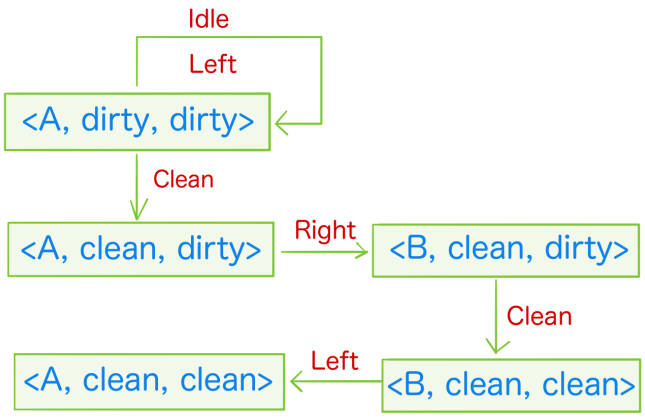

Figure 2.2: a graph of the mopot

As shown in the image above, each state is represented as a node/vertex, and all the nodes form a graph. Now that we have considered states as nodes, we can reconsider the process of the Goal-Based Agent. The task of starting from the [start state] to the [goal state] is actually finding a path from the [start node] to the [goal node] in the graph. This reminds us of what we have covered in CS2040S- searching. We will come into details in the next few sections.

2.3 Searching approaches in the graph of Goal-Based Agent

Before we dive into the specific approaches of searching, let's first get some key concepts:

- Branching factor(b): maximum number of edges out of the state
- Depth(d): maximum number of path length from start state to goal state
- m: maximum path length from start state to any state

We measure the performance of the agent by calculating the cost of the path it takes, our goal is to minimise the cost.

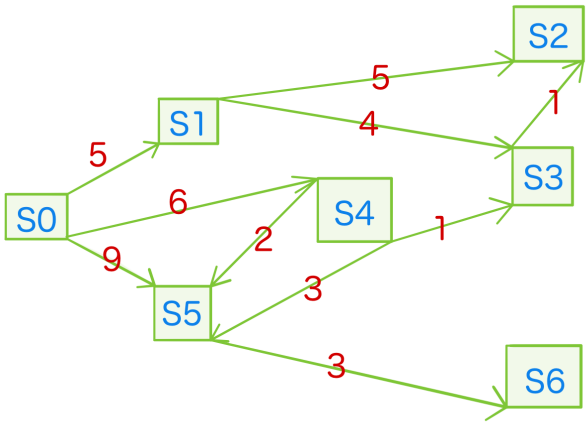

Figure 2.3: weighted graph

The above image is a weighted image. For the convenience of searching, each node will contain the following properties:

- State
- Parent
- Action to generate the node
- Path cost

2.3.1 First approach: Tree search

The image below is what it looks like when we convert the graph in image 2,3 into a tree, meaning that every node can have at most one parent, and as a result of this, same state can appear more than once in the tree.
To explain tree search in one sentence, that is, tree search first treats the graph as a tree, then applies BFS on it. **By converting the graph into a tree, tree search eliminates all cycles in the graph and hence avoids infinite cycles.**

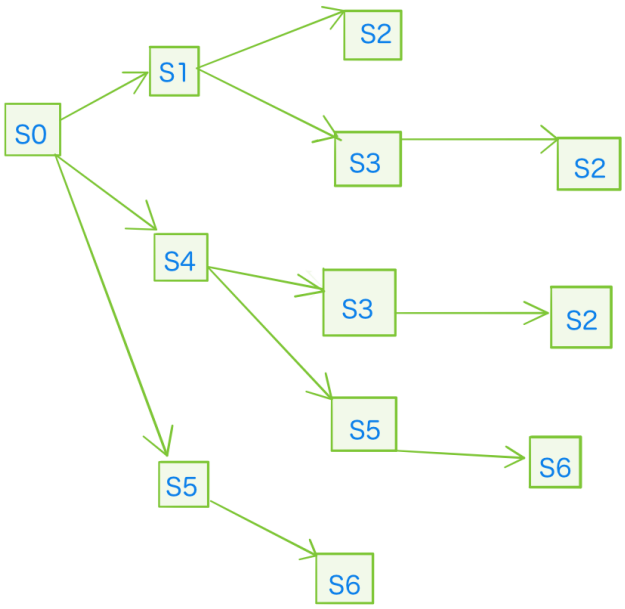

Figure 2.4: converting the graph in image 2.3 into a tree

Here is the pseudocode to of tree search.

Pseudocode:

```
FindPathToGoal(u)
    if (GoalTest(u))
        return path(u)
    else
        F = Queue(u)      //frontier
        while (F is not empty)
            u = F.pop()
            for all children v of u
                if (GoalTest(v))
                    return path(v)
                else
                    F.push(V)
        return failure
```

Disadvantage of tree search:
Because same state can appear more than once in the tree, the same state may be visited multiple times during the searching, resulting in a greater time cost.

2.3.2 Second approach: graph search

Graph search is much similar to tree search, as seen from the pseudocodes. The only difference is that graph search does not convert a graph into a tree.

- Difference between graph search and tree search: graph search does not treat graph as a tree
- How does this difference help: if we can solve the problem of going into infinite cycles, we can avoid visiting the same state multiple times
- How do we avoid infinite cycles: use a collection E to keep track of all visited nodes
- Everything is a trade off: while time complexity is reduced, space complexity increases since we need to store all visited nodes

Pseudocode:

```
FindPathToGoal(u)
    if (GoalTest(u))
        return path(u)
    else
        F = Queue(u)      //frontier
        E = {u}           //explored
        while (F is not empty)
            u = F.pop()
            for all children v of u
                if (GoalTest(v))
                    return path(v)
                else
                    if (v not in E)
                        E.add(v)
                        F.push(V)
        return failure
```

2.3.3 An analysis of BFS

Both tree search and graph search are BFS, let's see how well BFS works.

- Completeness: As long as the goal state is reachable from the start state, BFS will return a path. This is guaranteed because BFS would not return failure until every vertex is visited and has still yet to find a path from source to goal.
- Optimality: BFS does not guarantee shortest path when edges are of different weightage. A counter example is, if we apply BFS on figure 2.3, set S_0 as source and S_5 as goal, BFS would return the path $S_0 \rightarrow S_5$ whose length is 9 instead of the shorter path $S_0 \rightarrow S_4 \rightarrow S_5$ whose length is only 8.
- Time complexity: Assuming that every node has at most b children, and the graph has d layers. Time complexity $= b + b^2 + b^3 + ... + b^d = O(b^d)$
- Space complexity: Space complexity = space complexity(E) + space complexity(F) = number of nodes in the graph + number of nodes on the last layer $= O(b^d)$

Can we find a better approach that meets both completeness and optimality?

2.3.4 The third approach: Dijkstra

Dijkstra algorithm is invented for finding shortest paths from a source node to other nodes in a graph, it does so by keeping updating the distance between source node and every visited node.

Pseudocode:

```
function Dijkstra(Graph, source, goal):
    new PriorityQueue Q

    //initialization
    for each vertex v in Graph:
        dist[v] = INFINITY
        pre[v] = UNDEFINED
        Q.add(v, dist[v])    //the PriorityQueue is sorted by distance
    dist[source] = 0

    //start visiting vertices and updating distance
    while Q is not empty
        u = Q.extractMin()
        if u is goal
            return path(u)
        else
            for each neighbor v of u
                altDistance = dist[u] + length(u, v)
                if altDistance < dist[v]
                    dist[v] = altDistance
                    pre[v] = u
                    Q.decrease_priority(v, dist[v])
    return path(goal)
```