32002: AI (Uninformed Search) **Spring 2014**

Lecture 24-26: March 06, 2014

Lecturer: K.R. Chowdhary *: Professor of CS*

24.1 Introduction

The state space consists set of vertices V and set of connections between them in the form of edges (links), E. Thus, the search space is a graph $G = (V, E)$. Since, the states, already visited in the process of search should not be visited again to avoid cycles, the search paths takes shape of a tree, rather than graph.

In the conventional trees and graphs the space limits are fixed, i.e., the number of vertices, and edges connecting them are already given. However, in AI search problems, the alternate moves are generated from each vertex, like in a chess game, the search tree is to be generated and simultaneously be searched.

The initial configuration of the problem description is *start* state, at which we apply the specified rules to generate new states (vertices). Thus, it becomes similar to a tree with start configuration as *root* node, then there are interior nodes, and child nodes at the lowest level having no further children.

The 8-puzzle game is shown in figure 24.1, with initial configuration, goal configuration, and transitions (moves) possible from each state. We refer the configuration of the game equal to the collective status of tiles on the board. This configuration we call as state.

The Search Strategies are evaluated along the following dimensions:

- *Completeness:* Does it always find a solution if one exists?
- *Time complexity:* What is to be Number of nodes generated ? The duplicate nodes generated due to multiple paths, are also counted.
- *Space complexity:* What is maximum number of nodes in memory at any time?
- *Optimality:* Does it always find a least-cost solution?

24.2 Complexities of State-space Search

Let us try to find out number of states to be visited in the worst case to reach to goal state in the case of search required in figure 24.1. Once a state is visited in a search path, it should not be reached again in the future in that path, otherwise there will be a cycle formed and we never reach to goal state. Thus all unique states generated (like shown in figure as $a, b, c, \ldots, z$), in the path to *goal*, in the worst is $9! = 362880$.

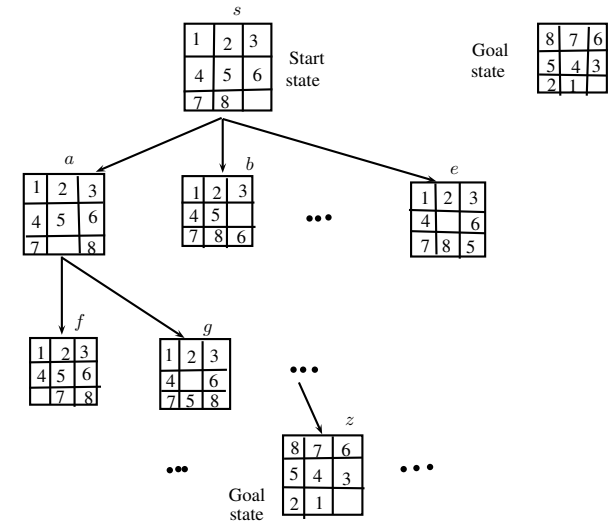
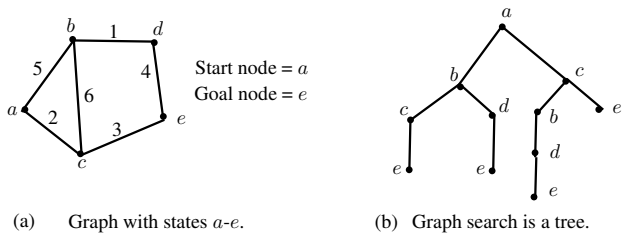


Figure 24.1: The 8-puzzle Game.



(a) Graph with states a-e. (b) Graph search is a tree.

Figure 24.2: Undirected graph.

In popular board game like *Chess*, for a particular strategy of opponent, we definitely have some winning strategy, represented by a configuration. Our aim is to reach to that configuration from the starting configuration, in step-by-step way, where each step is a move by us, followed by move by opponent. Each move transforms the chessboard from one configuration to another configuration. Suppose we assign the job to a computer to play against us as opponent. So let us try to compute the number of total moves required in the worst case. Assuming on the average 20 alternate moves for a configuration, and assuming there are 100 different configurations, the total states which can be generated and searched are 20^{100}. This is greater than 10^{130}, a combinatorial explosion. This value is even more than number of pico-seconds passed since Big-bang occurred and number of molecules in the known universe. Winning a game on computer, using an exhaustive search method, amounts to going through these sequence of states, which ultimately should lead to goal state. from above, we note that the search time is exponential in nature.

Consider the graph shown in figure 24.2 for a small size problem, where it is required to reach to goal state g from the start state a.

For the graph (figure 24.2(a)), when searched from the start state s, the states generated are shown in the tree in figure 24.2(b). Since objective of search is to reach to the goal, a search process terminates the moment the goal state is reached through any path. The tree in figure 24.2(b) shows the total possibles states generated in the worst case. □

Thus, we can conclude that problem solution through search process is a tree search (even if it is a graph

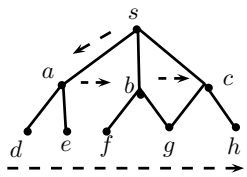


Figure 24.3: Breadth-first Search.

originally) with specified start node and goal node. Tree search can end-up repeatedly visiting the same nodes, unless it keeps track of all nodes visited. But, this could take vast amounts of memory, so large that most computers do not have, even for small size graph of say 50 nodes.

For the graph shown in figure 24.2(b), the possible paths set for start a and destination node e, is $\{abce, abde, acbde, ace\}$.

Since all the nodes are required to be searched - the search is called *exhaustive* search. The word 'blind', because, it is not known, which direction first exploring will lead to goal faster. To search the entire state space, the two important approaches are - *breadth-first search* (BFS) or *depth-first search* (DFS). In addition, what others approaches exist are their variants, for example, *depth-limited* search, *iterative deepening DFS*, and *bidirectional search*.

24.2.1 Breadth-First Search

A BFS phenomena is indicated in 24.3 where dotted trace is showing order in which nodes are tested for the goal.

The *BFS* can be implemented using a *queue* type data structure, named here as **List**. The front node of the of the queue is represented by **List**.$Head$. The algorithm for this is shown as: Algorithm 1, which checks all the paths at a given length, before testing the paths of longer length.

Algorithm 1 BFS(Input: **G**, **S**, **Goal**)

```
1: List = [S]
2: repeat
3:    if List.Head = Goal then
4:       return success
5:    end if
6:    generate children set C of List.Head
7:    append C to List
8:    delete List.Head
9: until List = []
10: return fail
```

If the BFS algorithm 1 is applied for generate-and-search, a tree like one shown in 24.3 gets constructed, and entire tree needs to be searched in the worst case. The BFS order of the above case is $s, a, b, c, d, e, f, g, h$. Though we certainly locate the goal if it exists, the path to goal from start state is not remembered. This however, is simple to find in the following way. In the *queue* data structure, used to implement the BFS, each entry is stored like (x, y), where x is next child node to be explored for goal, and y is its parent node. The value $y = 0$ represent the root node. We call this queue as 'open-list', i.e, the nodes which have not been fully explored. When a node is deleted from front of list, we add this deleted node into a separate list,

Table 24.1: Trace of state space search (BFS).

Open-list	**Closed-list**
$[(s,0)]$	[]
$[(a,s),(b,s),(c,s)]$	$[(s,0)]$
$[(a,s),(b,s),(c,s),(d,a),(e,a)]$	$[(s,0)]$
$[(b,s),(c,s),(d,a),(e,a)]$	$[(s,0),(a,s)]$
$[(b,s),(c,s),(d,a),(e,a),(f,b),(g,b)]$	$[(s,0),(a,s)]$
$[(c,s),(d,a),(e,a),(f,b),(g,b)]$	$[(s,0),(a,s),(b,s)]$
$[(c,s),(d,a),(e,a),(f,b),(g,b),(g,c),(h,c)]$	$[(s,0),(a,s),(b,s)]$
$[(d,a),(e,a),(f,b),(g,b),(g,c),(h,c)]$	$[(s,0),(a,s),(b,s),(c,s)]$
$[(e,a),(f,b),(g,b),(g,c),(h,c)]$	$[(s,0),(a,s),(b,s),(c,s),(d,a)]$
$[(f,b),(g,b),(g,c),(h,c)]$	$[(s,0),(a,s),(b,s),(c,s),(d,a),(e,a)]$
$[(g,b),(g,c),(h,c)]$	$[(s,0),(a,s),(b,s),(c,s),(d,a),(e,a),(f,b)]$

called 'closed-list'. The entries in table 24.1 shows the search operations to search the goal node f, with each row in table showing a deletion or append operation.

The search process is terminated when goal node g is encountered as the first node in 'open-list'. After reaching the goal node, we can backtrack to find out the path from goal node to start as: '$(g,b),(b,s)$'. Another path is '$(g,c),(c,s)$'. We consider the first, or if path lengths are given then the one which is shorter.

Since BFS always explores the shallow nodes before the deeper ones, thus BFS finds the shallowest path leading to the goal state. Since, a BFS algorithm is bound to find a goal, if at all a goal exist, the BFS is *complete* inference system, as well as optimal.

24.2.2 Depth-First Search

To perform a DFS search, we generate all the next states for the root node, then pickup the left-most node, generate the children for this, check for goal, and repeat this, until we reach to goal or the dead end.

Algorithm 2 DFS(Input: **G**, **S**, **Goal**)

```
1: List = [S]
2: repeat
3:    if List.Head = Goal then
4:       return success
5:    end if
6:    generate children set C of List.Head
7:    delete List.Head
8:    insert C at begin of List
9: until List = []
10: return fail
```

When reached to the dead end (the child node), from then, back track, to the siblings of this node, apply the DFS, then reach to siblings of its parent, and so on. Applying the DFS algorithm 2, generates a tree

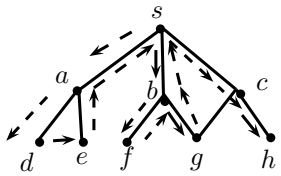


Figure 24.4: Depth-first Search.

like one shown in figure 24.4. The order of nodes visited in DFS are: $s, a, d, e, b, f, g, c, h$. For DFS also, a table of trace can be constructed as it was done for BFS.

24.2.3 Analysis of BFS and DFS

Consider the figure 24.3, and 24.4. Assume that the goal node is e. Using the DFS method it needs only three steps to reach to node e. Where as it requires 5 steps to reach to e if BFS search method is used. Thus, for a deep goal node, DFS is considered better. If the node to be searched was c, the BFS required three steps, and DFS, which first search deeper information, then shallow, needs total 7 comparison or the number of generate steps to reach the goal. Thus, which approach is best, depends on the position of goal node. Also, if branching factor b (number of branches per node) is large, the DFS is better suited, and BFS is worst. Thus, the efficiency of search depends on the structure of tree as well as the search method used.

Let us assume that in the tree constructed in a search has depth d and for each node there are b nodes that gets generated, called *branching factor* of the tree. For a BFS tree the time spent for any search is maximum number of nodes visited to determine the worst-case *time-complexity* of the search. At a time how many nodes are in the 'open-list', determine the space, or *space-complexity* of the algorithm.

For a BFS search, total nodes visited for tree of depth d are:

$$1 + b + b^2 + \cdots + b^n = O(b^d), \tag{24.1}$$

which is worst-case time complexity, and the maximum number of nodes in the 'Open-list' will be at the lowest level of the tree. Thus space-complexity is $O(b^d)$. Hence, in the case of BFS, both time and space complexity are $O(b^d)$.

For a DFS search, in the worst-case all the nodes are visited. Hence, time-complexity is same as for BFS, and equal to $O(b^d)$. Since, DFS needs to store only b nodes per level for a depth of d, the total nodes to be stored are $b \times d$, and space-complexity is $O(bd)$.

However, the DFS is not good at finding the goal, if the goal is on the opposite side of the tree which is searched, even though it is shallow. So many a times, a *depth-limited DFS search* is performed, i.e., searching the space for predetermined depth only. A method which combining the advantages of both the BFS and DFS is *iterative deepening DFS*, discussed below.

24.2.4 Depth-First Iterative Deepening (DFID) Search

The iterative deepening DFS reaches to shallow goals much faster than the ordinary DFS. It first sets the tree depth $d = 0$, performs DFS and checks root. Then discards all the nodes generated in the previous

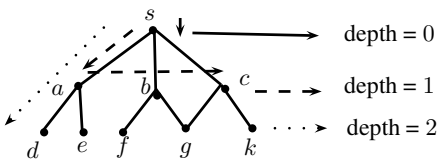


Figure 24.5: Search tree of Iterative deepening DFS.

search, starts over and do a depth-first search for $d = 2$. next, starts over and do a DFS for $h = 3$, and so on. In ordinary DFS, the goal node c (figure 24.5) will get searched in 8 comparisons, where as using iterating deepening DFS, search is carried out for tree depth $d = 0$, next in $d = 1$ the node c gets located, requiring total $1 + 4 = 5$ comparisons.

Since, DFID expands all the nodes at at given depth. it is guaranteed to find a shortest-length solution. The worst case complexities for the this method remains the same as ordinary DFS, however, the average case improves. This is because, in the worst case you still need all nodes to be compared, taking time equal to $O(b^d)$, and worst case space requirements is $O(bd)$ only.

24.2.5 Bidirectional Search

If the search is carried out in both the directions, it can be speeded up. Consider that a bidirectional search is carried out with the depth as d and branching factor b. If each side progresses with same depth, the search required from each, for example, for iterative deepening DFS, has $O(b^{\frac{d}{2}})$ for time, and $O(b\frac{d}{2})$ for for space. When combined from both opposite sides, it becomes $2 \times O(b^{\frac{d}{2}}) = O(b^{\frac{d}{2}})$ for *time*, and $2 \times O(b \times \frac{d}{2}) = O(bd)$ for *space*.

However, for bidirectional search to be implemented, it is necessary that invertible functions must be available for generating nodes. Which, in fact, exists in the case of board game.

24.3 Problem formulation for Search

Given a problem for solution using AI search, we need to formulate its solution through a state space search. This requires discovering the start and goal states, which may be explicitly specified in the problem, else we need to discover from the problem specification. The next step is to find out the possible moves to generate children at each state, i.e, expand the states. Applying this we conclude the graph, if the size of the problem is small, which can be searched as BFS or DFS, or using any other method discussed above.

If the size of state space is large and ordinarily cannot be accommodated on a page, we directly generate the tree using DFS or BFS, and expand the nodes until goal is reached.

A farmer (F) wants to move himself, a fox (X), a goose (G), and some grains (R) across a river. Unfortunately, his boat is so tiny that he can take only one of his possessions across on any trip. Worse yet, an unattended fox will eat a goose, and an unattended goose will eat grains, so the farmer must not leave the fox alone with the goose nor the goose alone with the grains. What is he to do?

We create a node for each safe position. Then draw the possible next moves by a link from the one to the other. The end result is a graph with 10 nodes shown in figure 24.6. What one needs to do is to find a

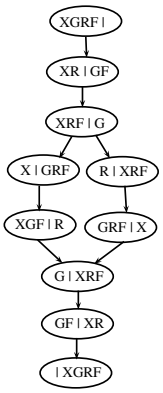


Figure 24.6: Farmer-goose-grains-fox Problem as Graph search.

shortest path from the node with all the participants on one side to the node with all the participants on the other side.

References

[1] Chowdhary K.R. (2020) Logic and Reasoning Patterns. In: Fundamentals of Artificial Intelligence. Springer, New Delhi. https://doi.org/10.1007/978-81-322-3972-7_8