CS324: State-space, DFS, BFS, Uninformed Search. **Spring 2015**

Lecture 14: March 9, 2015

Lecturer: K.R. Chowdhary *: Professor of CS (VF)*

14.1 Introduction

Search is one of the operational task that characterize AI programs best. Almost every program depends on search procedure to be performed to carry out its prescribed functions. Problems are generally specified in terms of states, and solution corresponds to the goal states. Solving a problem then amounts to searching through the different states, called *state space*, until one of the goal state is reached. If goal state is not found, it concludes that solution does not exist for the problem or the goal state is not reachable.

The state space consists set of vertices V and set of connections between them in the form of edges (links) from E. Thus, the search space is a graph $G = (V, E)$. 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. Hence, you cannot determine the size of tree to be searched in advance.

14.2 Representation of Search

The search space is generally represented as a directed graph (in fact a tree), with vertices as states, and edges of the graph as transitions for moves to explore for the goal state. The initial configuration of the problem description is *start* state, at which we apply the specified rules to generate new states (vertices).

Example 14.1 *8-Puzzle Game.*

The 8-puzzle game is shown in figure 14.1, with *initial configuration*, *goal configuration*, and transitions (i.e., 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*.

Using the allowed moves of a numbered tile (1-8), to tile left, right, up, down, to the destination blank-tile, we generate new states starting from the start state (S) as $A, B, C, \ldots,$. As a state is generated, it checked for the goal state. If yes, we terminate the process and declare the fact that goal is reached. If not, the generated states are further expanded by application of *generate rules* (moves), until the goal is reached.

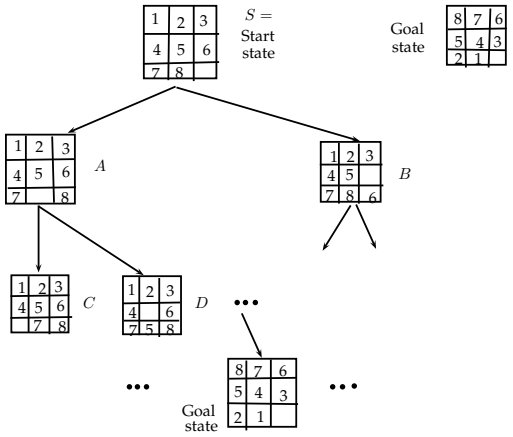

Figure 14.1: The 8-puzzle Game.

□

The Search Strategies are evaluated along the following dimensions:

- *Completeness:* Does it always find a solution if one exists?
- *Optimality:* Does it always find a least-cost solution, i.e., in terms of minimum number of transitions?
- *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?

14.3 Complexities of State-space Search

From the previous section, it is clear that the process of searching can be automated by applying the rules of moves to cause transitions from one state to next, until you reach to the goal state. However, in the process a state should not repeat in the path, for example, like $v_1 \to v_2 \to v_3 \to v_1 \to v_4 \to v_5 \to v_g$. Such repeated vertices form a loop and we never reach to goal state. Since the previously visited vertex v_1 is repeated, the path may instead be taken as $v_1 \to v_4 \to v_5 \to v_g$.

Let us try to find out number of states to be visited in the worst case, to reach to goal state, in the search required in figure 14.1. Thus, all unique states generated (like shown in figure as $A, B, C, \ldots, goal$), in the path to *goal*, in the worst it is $9! = 362880$.

Example 14.2 *Consider the graph shown in figure 14.2 for a small size problem, where it is required to reach to goal state E from the start state A.*

Solution: For the graph shown in figure 14.2(a), when it is searched from the start state A, the states generated are shown in the tree in figure 14.2(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 14.2(b) shows the total possibles states generated in the worst case, of course many repeated, but in a path no vertex is repeated. □

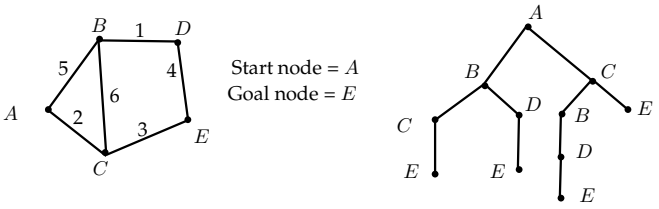

Figure 14.2: Undirected graph.

14.4 Uninformed Search

It is also called *blind search*, because we do not know, which moves ultimately will lead to the goal faster. Since all the nodes are required to be searched - the search is called *exhaustive* search. To search the entire state space, the two important approaches are - *breadth-first search* (BFS) or *depth-first search* (DFS). What others approaches exist are variants of DFS and BFS. For example, *depth-limited* search, *iterative deepening DFS*, and *bidirectional search*.

14.4.1 Breadth-First Search

To carry out the BFS for a graph or to solve a puzzle, like, shown in figure 14.1, first, root node is checked if it is goal, if yes then terminate the process after declaring that goal is found. If goal is not found, the children nodes are generated for root node and tested if any one of them is goal. Then further children are generated for each of them, and so on, until goal is reached or the search is terminated. In the latter, no new state can be generated in any of the search path. A BFS phenomena is indicated in 14.3 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. The inputs to this algorithm are: the graph **G**, start node **S**, and goal node **Goal**. If goal is reached it returns the *success*. After searching all the vertices, which will be indicated by a empty **List**, if goal is still not found, the algorithm returns fail and exits.

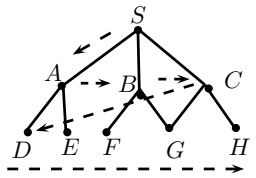

Figure 14.3: Breadth-first Search.

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 14.3 gets constructed, and this entire tree needs to be searched in the worst case to find the goal state. The BFS order of this tree is $S, A, B, C, D,\ E, F, G, H$. The search process is terminated when goal node G is encountered as the next node.

Since BFS always explores the shallow nodes before the deeper ones, thus BFS finds the shallowest path leading to the goal state. A modified and improved search can be obtained by expanding only the lowest cost node (n). This is called *Uniform cost search*. This modifies BFS such that the cost of a path should remain low but not decreasing hence the term "uniform". Since, a BFS algorithm is bound to find a goal, if at all the goal exists, the BFS is *complete* inference system, as well as optimal.

14.4.2 Depth-First Search

Backtracking, or *depth-first search*, is a technique which has been widely used for finding solutions to problems in combinatorial theory and artificial intelligence, but whose properties have not been widely analyzed. Suppose G is a graph which we wish to explore. Initially all the vertices of G are unexplored. We start from some vertex of G and choose an edge to follow. Traversing the edge leads to a new vertex. We continue in this way; at each step we select an unexplored edge leading from a vertex already reached and we traverse this edge. The edge leads to some vertex, either new or already reached. Whenever we run out of edges

leading from old vertices, we choose some unreached vertex, if any exists, and begin a new exploration from this point. Eventually, we will traverse all the edges of G, each exactly once.

Consider the following choice rule: when selecting an edge to traverse, always choose an edge emanating from the vertex most recently reached which still has unexplored edges. A search which uses this rule is called a *depth-first search* (DFS). The set of old vertices with possibly unexplored edges may be stored on a stack type data structure. Thus a depth-first search is very easy to program either *iteratively* or *recursively*, provided we have a suitable computer representation of a graph.

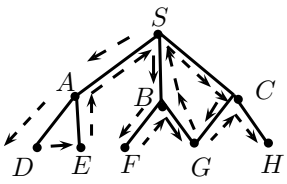

Figure 14.4: Depth-first Search.

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 like one shown in figure 14.4. The order of nodes visited in DFS order 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, to find the path from root node to goal node.

14.4.3 Analysis of BFS and DFS

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 worst-case time spent for any search is the maximum number of nodes visited to determine the goal. This is 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:

$$1 + b + b^2 + \cdots + b^d = O(b^d), \tag{14.1}$$

which is worst-case *time complexity*, and the maximum number of nodes in the 'Open-list'will exist 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 also, in the worst-case, all the nodes are required to be 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 in memory are $b \times d$, hence *space-complexity* is $O(bd)$. Usually the branching factor b is much smaller than d, results to the space complexity as $O(d)$.

In the BFS, since all the nodes at a given depth are stored in order to generate the nodes at the next depth, the minimum number of nodes that must be stored to search to depth d is b^d, which is $O(b^d)$. As with time, the average-case space complexity is roughly one-half of this, which is also $O(b^d)$. This space requirement of breadth-first search is its most critical drawback. As a practical matter, a breadth-first search of most problem spaces will exhaust the available memory long before an appreciable amount of time is used.

The Depth-first search avoids the memory limitation of BFS. It works by always generating a descendant of the most recently expanded node, until some depth cutoff is reached, and then backtracking to the next most recently expanded node and generating one of its descendants. Therefore, only the path of nodes from the initial node to the current node must be stored in order to execute the algorithm. If the depth cutoff is d, the space required by DFS is only $O(d)$.

Another drawback, however, to DFS is the requirement for an arbitrary cutoff depth. If branches are not cut off and duplicates are not checked for, the algorithm may not terminate.

14.4.4 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, hence checks the root node for goal. Then discards all the nodes generated in the previous search, starts over and do a depth-first search for $d = 1$. Next, starts over and do a DFS for $d = 2$, and so on. In ordinary DFS, the goal node C (figure 14.5) 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 given depth, it is guaranteed to find a solution in shortest-time. The disadvantage of DFID is that it performs wasted computation prior to reaching to goal depth. However, this wasted computation does not effect the asymptotic growth of the run time for exponential search. The intuitive reason is that almost all the work is done at the deepest level of the search.

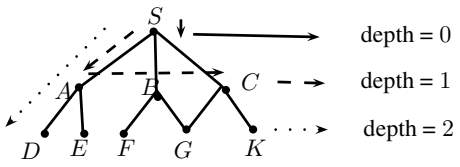

Figure 14.5: Search tree of Iterative deepening DFS.

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 $O(d)$ only.

14.4.5 Bidirectional Search

If the search is carried out in both the directions, it can be sped-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 $2 \times O(b \times \frac{d}{2}) = O(bd)$, which is $O(d)$, for *space*. For bidirectional it is necessary that *invertible functions* must be available for generating nodes.