



USMAN INSTITUTE OF TECHNOLOGY

Department of Computer Science CS321 Artificial Intelligence

Lab# 04 Uninformed Search Algorithms

Objective:

This experiments models problems as search problems and then explores their possible solutions using different uniformed search algorithms.

Name of Student: _____

Roll No: _____ Sec. _____

Date of Experiment: _____

Marks Obtained/Remarks: _____

Signature: _____

Search

Search is looking for a sequence of actions that reaches the goal states. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. After formulating a goal and a problem to solve, a search procedure is called to solve it. The solution is then used to guide the actions, whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the goal is reached.

Formulating Problems

There are five components which formulate a problem as a search problem.

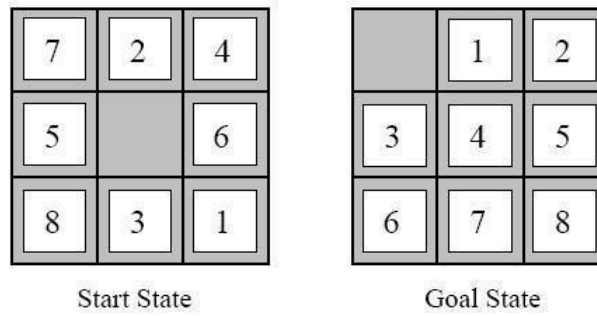
- Initial state
- Actions
- Transition model
- Goal test
- Path cost

form a problem. This formulation is abstract, i.e., details are hidden. Abstraction is useful since they simplify the problem by hiding many details but still covering the most important information about states and actions (retaining the state space in simple form), therefore abstraction needs to be valid. Abstraction is called valid when the abstract solution can be expanded to more detailed world. Abstraction is useful if the actions in the solution are easier than the original problem, i.e, no further planning and searching. Construction of useful and valid abstraction is challenging.

Example Problem: 8-Puzzle Problem

The 8-puzzle is often used as test problem for new search algorithms in AI. The 8-puzzle has $9!/2 = 181,440$ reachable states and is easily solved. The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24-puzzle (on a 5×5 board) has around 1025 states, and random instances take several hours to solve optimally. The 8-puzzle, an instance of which is shown in the figure, consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

- **States:** Description of the location of each of the eight tiles and the blank square.
- **Initial State:** Initial configuration of the puzzle.
- **Actions & transition model:** Moving the blank; left, right, up, or down.
- **Goal Test:** Does the state match the goal state?
- **Path Cost:** Each step costs 1 unit



Search Schemes

Searching is the universal technique of problem solving in AI.

Infrastructure for a Search Algorithm

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

- $n.STATE$: the state in the state space to which the node corresponds;
- $n.PARENT$: the node in the search tree that generated this node;
- $n.ACTION$: the action that was applied to the parent to generate the node;
- $n.PATH-COST$: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function `CHILD-NODE` takes a parent node and an action and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Next, the frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand QUEUE according to its preferred strategy. The appropriate data structure for this is a queue. The operations on a queue are as follows:

- EMPTY? (queue) returns true only if there are no more elements in the queue.
- POP (queue) removes the first element of the queue and returns it.
- INSERT (element, queue) inserts an element and returns the resulting queue.

Types of Search Algorithm

There are some single-player games such as tile games, Sudoku, crossword, etc. The search algorithms help you to search for a particular position in such games. There are two kinds of AI search techniques:

- Uninformed search
- Informed search

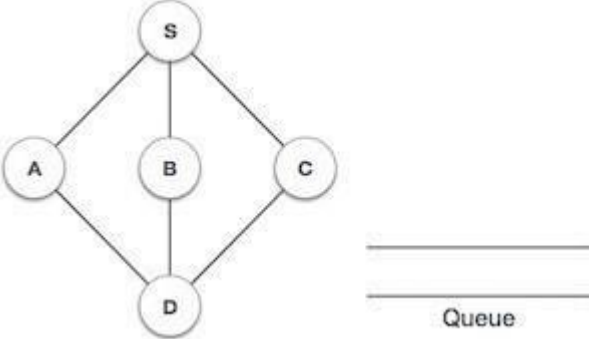
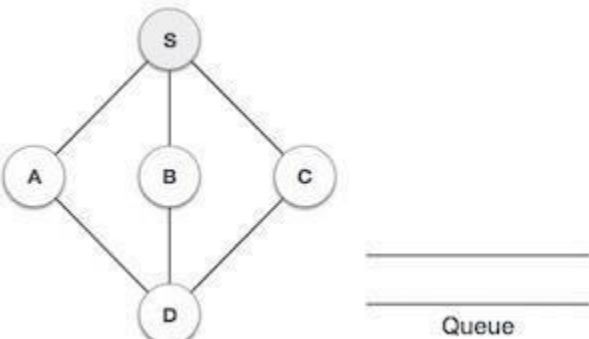
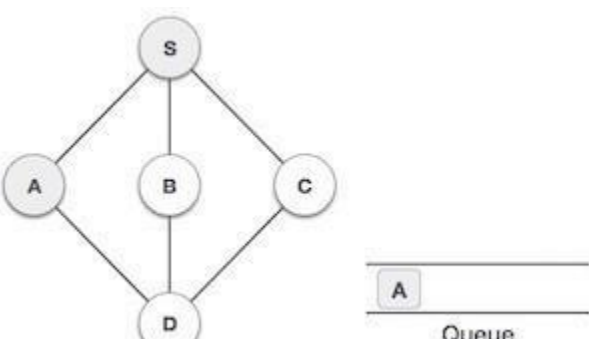
Uninformed Search

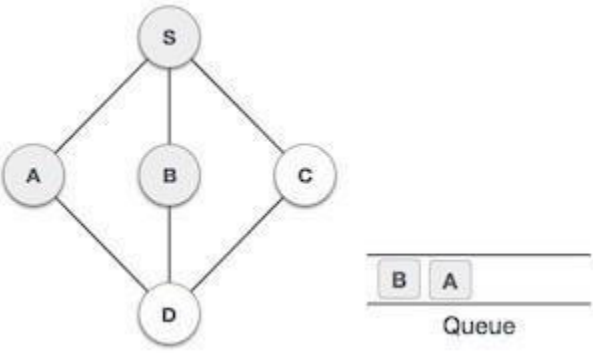
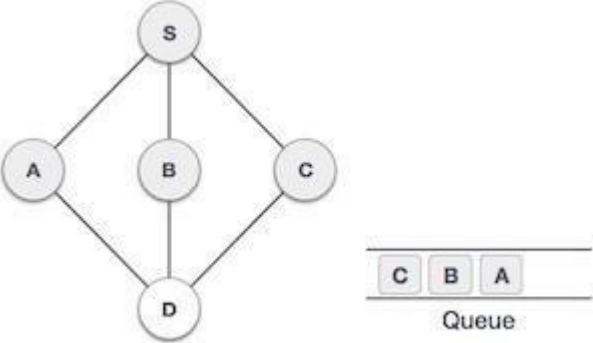
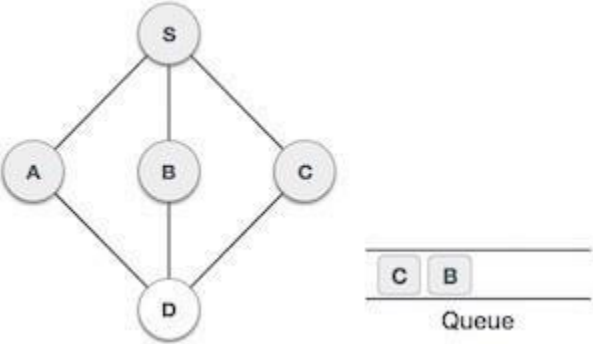
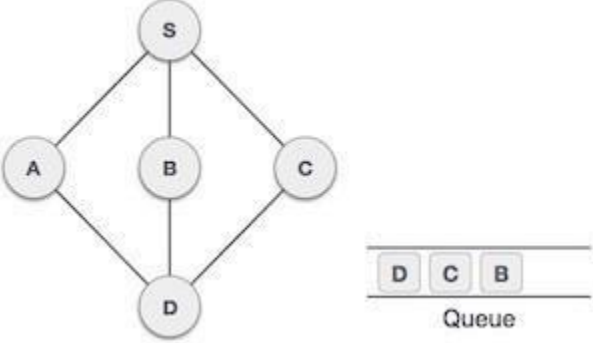
Sometimes we may not get much relevant information to solve a problem. Suppose we lost our car key and we are not able to recall where we left, we have to search for the key with some information such as in which places we used to place it. It may be our pant pocket or may be the table drawer. If it is not there then we have to search the whole house to get it. The best solution would be to search in the places from the table to the wardrobe. Here we need to search blindly with fewer clues. This type of search is called uninformed search or blind search. There are two popular AI search techniques in this category:

- Breadth first search
- Depth first search

Breadth-First Search

It starts from the root node, explores the neighboring nodes first and moves towards the next level neighbors. It generates one tree at a time until the solution is found. It can be implemented using FIFO queue data structure. This method provides shortest path to the solution.

Step	Traversal	Description
1.		Initialize the queue.
2.		We start from visiting S (starting node), and mark it as visited.
3.		We then see an unvisited adjacent node from S . In this example, we have three nodes but alphabetically we choose A , mark it as visited and enqueue it.

4.		Next, the unvisited adjacent node from S is B . We mark it as visited and enqueue it.
5.		Next, the unvisited adjacent node from S is C . We mark it as visited and enqueue it.
6.		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A .
7.		From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

Pseudocode

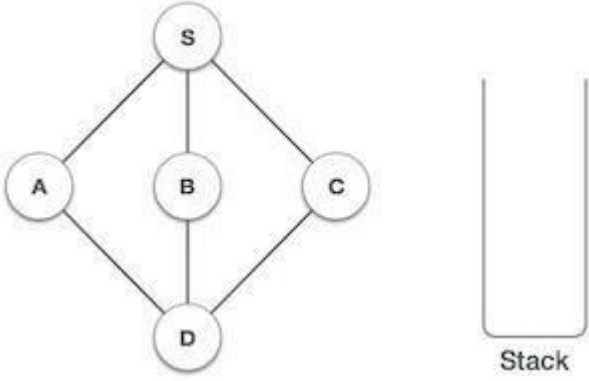
```

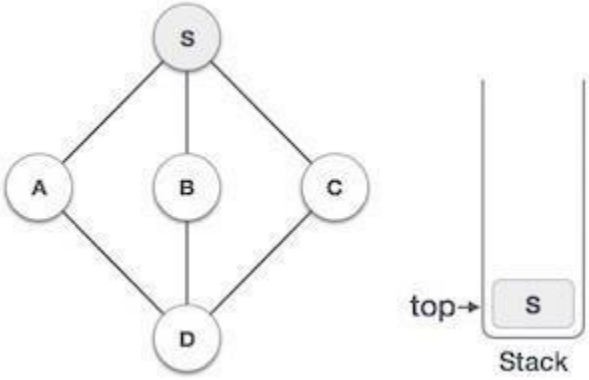
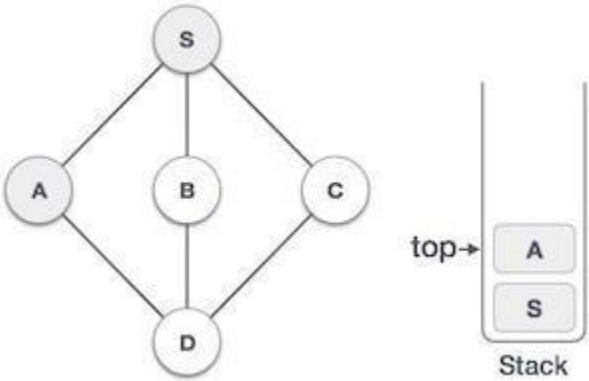
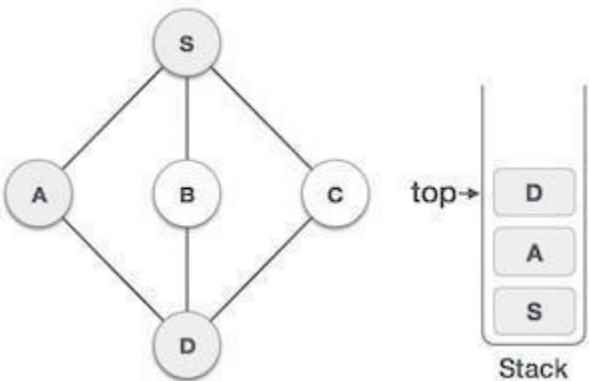
BFS (G, s)           //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s ) //Inserting s in queue until all its neighbor vertices
are marked.

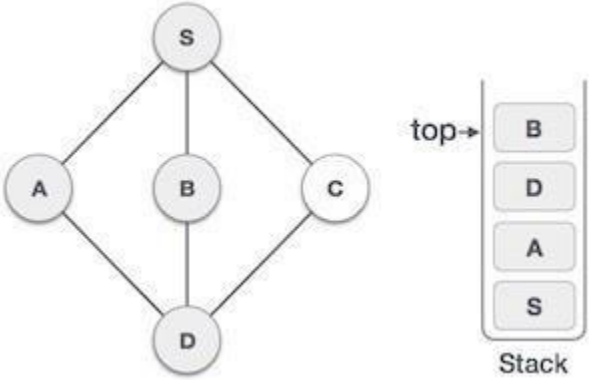
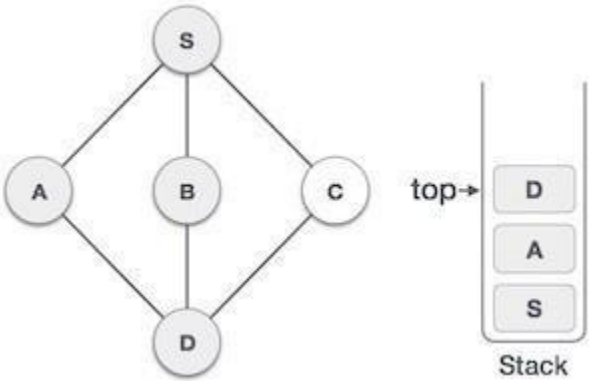
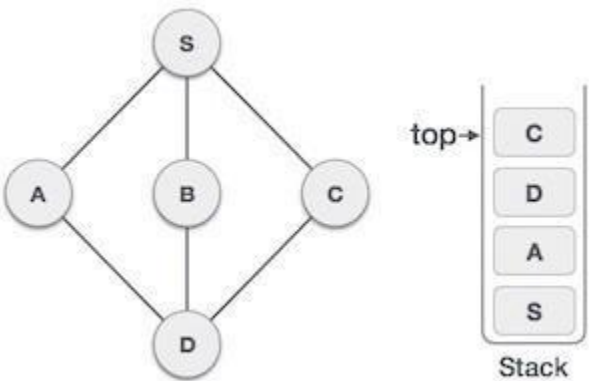
    mark s as visited.
    while (Q is not empty)
        //Removing that vertex from queue,whose neighbor will be visited now
        v = Q.dequeue( )
        //processing all the neighbours of v
        for all neighbours w of v in Graph G
            if w is not visited
                Q.enqueue( w ) //Stores w in Q to further visit
                               its neighbour
                mark w as visited.
    
```

Depth-First Search

It is implemented in recursion with LIFO stack data structure. It creates the same set of nodes as Breadth-First method, only in the different order.

Step	Traversal	Description
1.		Initialize the stack.

2.		<p>Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.</p>
3.		<p>Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4.		<p>Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.</p>

5.		<p>We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.</p>
6.		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>
7.		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>

Pseudocode

```

DFS-iterative (G, s):           //Where G is graph and s is source vertex
    let S be stack
    S.push( s )                 //Inserting s in stack

```

```

mark s as visited.
while (S is not empty):
    //Pop a vertex from stack to visit next
    v = S.top( )
    S.pop( )
    //Push all the neighbors of v in stack that are not visited
    for all neighbors w of v in Graph G:
        if w is not visited :
            S.push( w )
            mark w as visited

DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)

```

Student Exercise

Task 1

Solve 8- Puzzle problem:

- Consist of 3×3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space.
- The objective is to reach a specified goal state, such as the one shown in the discussion above.

Task2

Solve depth first search and breadth first search of following graph starting from node A and reaching goal node G:

