Search

- Can often solve a problem using **search**.

Two requirements to use search:

- **Goal Formulation**.
  - Need goals to **limit search** and **allow termination**.
- **Problem formulation**.
  - Compact representation of problem space (**states**).
  - Define **actions** valid for a given state.
  - Define **cost** of actions.
- Search then involves moving from state-to-state in the problem space to find a goal (or to terminate without finding a goal).

Outcome of Search

- Possible outcomes:
  - **Goal** itself (i.e., does problem have solution?)
  - **Path** from initial state to goal state (i.e., sequence of steps to achieve something?)

- Search assumes that an environment is:
  - Static,
  - Observable,
  - Discrete, and
  - Deterministic.

Performance of Search

- Need to measure the performance of a search; i.e., given a search **strategy**, how good is it?

Four criteria:

- **Completeness**
  - Is the search strategy guaranteed to find a solution?
- **Optimality**
  - Is the solution found the best possible?
- **Time Complexity**
  - How long does the search strategy take to run?
- **Space Complexity**
  - How much memory does the search strategy require?

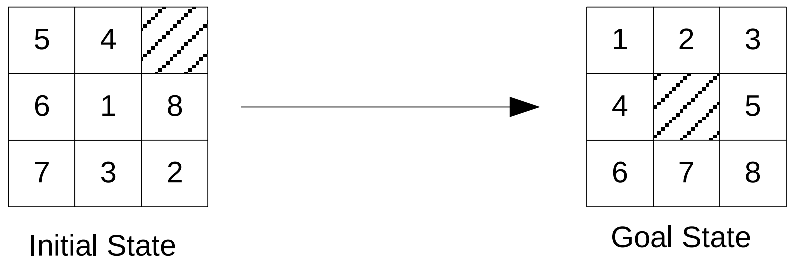
Problem Formulation

- Search requires a well-defined problem space including:
  - **Initial state**
    - A search starts from here.
  - **Goal state and goal test**
    - A search terminates here.
  - **Sets of actions**
    - This allows movement between states (successor function).
  - **Concept of cost** (action and path cost)
    - This allows costing a solution.

- The above defines a **well-defined state-space formulation** of a problem.

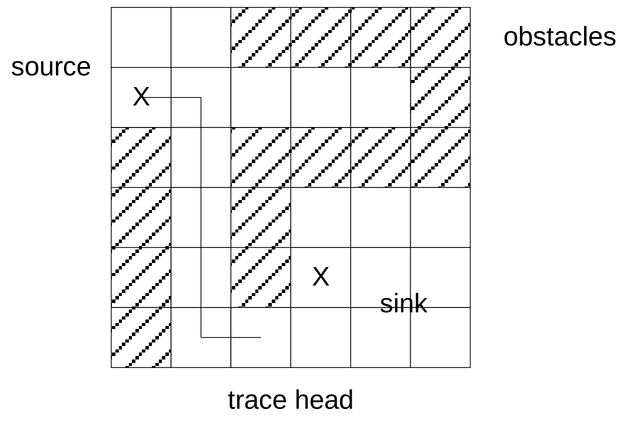
- Note: A state can represent either a complete configuration of a problem (e.g., 8-puzzle) or a partial configuration of a problem (e.g., routing).

Problem Formulation Example – 8 Puzzle



- Want to take an initial arrangement of tiles and get them into a desired arrangement.
  - States: encode location of each of 8 tiles and the blank.
  - Initial State: any arrangement of tiles.
  - Goal State: predefined arrangement of tiles.
  - Actions: slide blank UP, DOWN, LEFT or RIGHT (without moving off the grid).
  - Goal Test: position of tiles and blank match goal state.
  - Cost: sliding the blank is 1 move (cost of 1). Path cost will equal the number of moves from initial state to goal state.
- Solution is a path of moves to get to the goal state. Fewest moves is best.

Problem Formulation Example – Route Finding

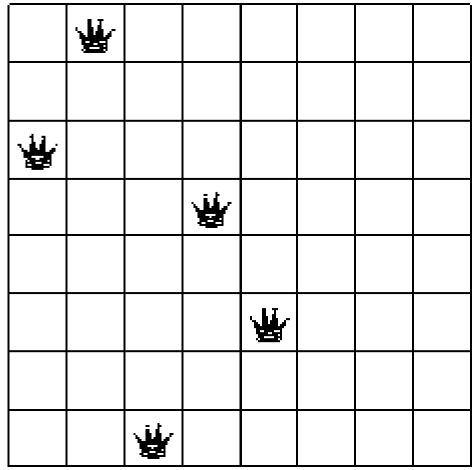


- Problem is to connect a source to a sink while avoiding obstacles on 2-D grid.

  - States: ordered pair (x,y) of the trace head.
  - Initial State: trace at location of source.
  - Goal State: trace at location of sink.
  - Actions: move trace head UP, DOWN, LEFT or RIGHT (without moving off the grid and avoiding obstacles).
  - Goal Test: trace at location of sink.
  - Cost: moving trace head costs 1. Path cost is length of trace.

- Solution might be (i) trace with shortest path or (ii) a path if one even exists.

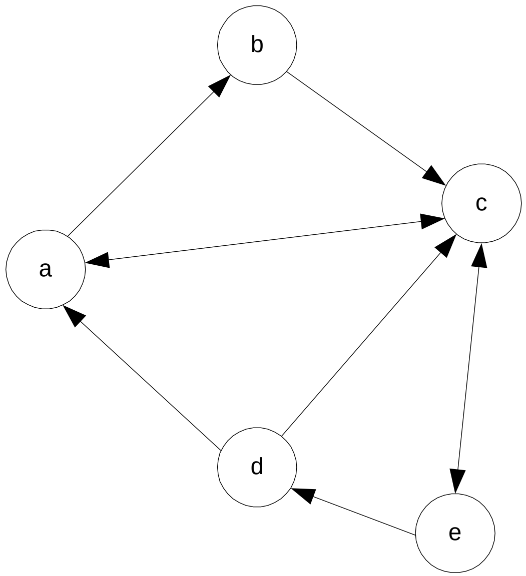
Problem Formulation Example – 8 Queens Problem



- Position 8 queens on a chess board such that no 2 queens attack each other.
  - States: placement of 0 to 8 queens on board such that no attacks.
  - Initial State: no queens placed.
  - Goal State: position of 8 queens in non-attacking arrangement.
  - Actions: place next queen into next column in a non-attacking position.
  - Goal Test: position of queens in non-attacking position.
  - Cost: placing next queen costs 0. Solution is simply one of many goal states.
- Solution is any goal state (no concept of path…)

Problem Formulations, Graphs and Search Trees

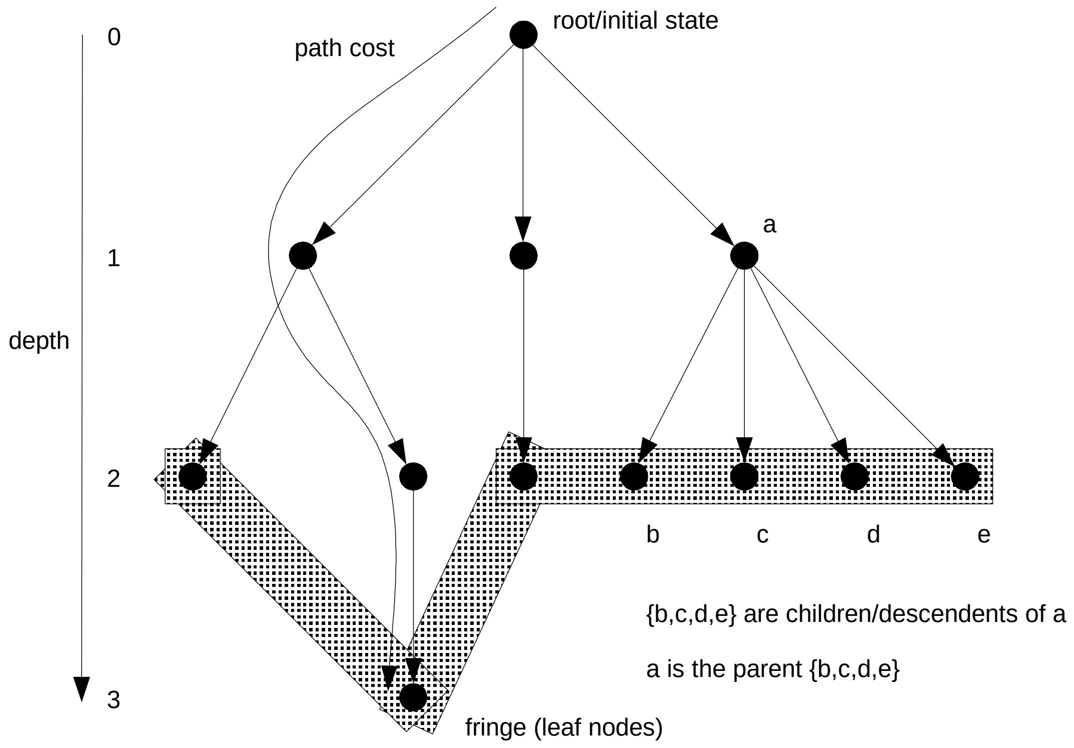
- Problems formulated for search has an analogy with directed graphs:
  - Nodes represent states (configurations or partial configurations).
  - Edges represent actions (directional)
- Output of search is either a path in the graph or a node that passes the goal test.



Problem Formulations, Graphs and Search Trees

- When we search in a graph, we build a **search tree**.
- Search trees also have nodes and edges:
  - Nodes are states (root is the initial state).
  - Nodes have a parent and descendants (neighboring states reachable via an action).
  - Nodes at the bottom of the tree are **leaf nodes** and define the **fringe**.
    - We will find a goal state in the fringe.
  - Edges represent actions and have costs.
- Each tree node has a path from the root and a path cost from the root.
- Nodes in a search tree are more than just states since they hold state information, reference to actions, path costs, etc.

Illustration of Search Tree



Comments on Search Trees

- Search trees are superimposed over top of the graph representation of a problem.
- While the graph might be finite, the search tree can be either finite or infinite.
  - Infinite if we allow repeated states due to reversible actions and/or cycles of actions.
- Some useful terminology:
  - The maximum number of children possible for a node, b, in a search tree is called the **branching factor**.
  - A finite tree has a **maximum depth**, d.
  - Any node in the search tree occurs at a **level**, l, in the tree ($l \leq d$).

Template of Generic Search

- Given concept of graph and search tree, generic search is a repetition of **choose, test, and expand**.

- A particular search strategy (uninformed or informed, more in a minute …) influences how we choose the next node to consider in the search! (this is where types of searches differ).

- Generally use a **queue** to store nodes on the fringe to be expanded. Different search strategies use different queue structures.

Template of Generic Search

```
open_queue.insert(init_state);
while (open_queue.size() != 0) {
        curr_state = open_queue.remove_front();
        if (is_goal(curr_state)) {
                return success; // and solution.
        }
        closed_queue.insert(curr_state); // state visited.

        child_state = expand(curr_state); // other states reachable via an action.
        for (i = 1 ; i <= child_state.size() ; i++) {
                if (open_queue.find(child_state[i]) || closed_queue.find(child_state[i])) {
                        ; // child state already expanded or in fringe.
                } else {
                        open_queue.insert(child_state[i]);
                }
                // nb: what if better path to child states?????
        }
};
return failure; // no solution.
```

Repeated States

- During search, desirable to avoid repeated states (i.e., revisiting the same state again and again).
  - This is possible due to reversible actions and/or cycles of actions.

- We can keep track of visited states and ignore repetition via a closed queue.
  - However, what happens if, upon revisiting a state, we find a that we have a better path/solution to the revisited state?
  - We need to deal with this (not shown in the generic template!).

Types of Search

Two main types of search:

- Uninformed search:
  - Has only the knowledge provided in the problem formulation (e.g., actions, costs, goal or not goal, etc.)
- Informed search:
  - Has additional knowledge in order to better judge the overall promise of an action in reaching a goal; e.g., has an estimate of cost to goal from current location).

Breadth First Search (BFS)

- Uninformed search strategy.
- Ignores action costs (equivalently, assumes all actions cost 1).
- Explores the state space systematically from initial state outward;
  - In terms of the search tree, explores all nodes at level k before exploring nodes at level k+1.

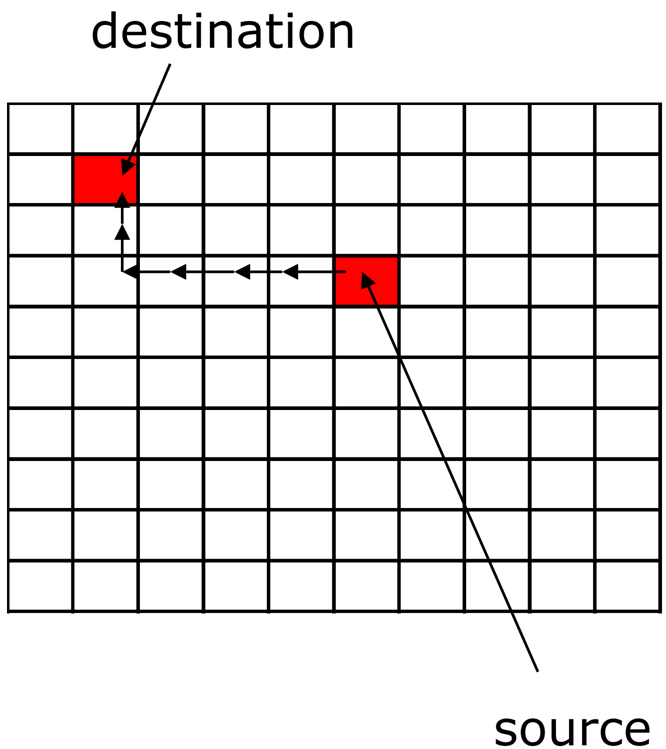
- Algorithm:
  - Uses an open and closed queue.
  - Open queue holds states that have been expanded, but not explored.
  - Closed queue holds states that have been explored (avoid repeats).

- The open queue is FIFO (first-in, first out) which guarantees the level-by-level exploration of the search tree.

Pseudo-code for BFS

```
open_queue.insert(init_state); closed_queue.clear();
while (open_queue.size() != 0) {
     curr_state = open_queue.remove_front(); // FIFO!!!
     if (is_goal(curr_state)) {
          return success; // Found goal.
     } else {
          closed_queue.insert(curr_state);
          child_state = expand(curr_state);
          for (i = 1 ; i <= child_state.size() ; i++) {
             if (!open_queue.find(child_state[i]) &&
                      !closed_queue.find(child_state[i])) {
                  open_queue.push_back(child_state[i]); // FIFO!!!
             } else ; // already expanded or explored.
          }
     }
}
return failure; // no solution found.
```

Illustration of BFS



- Search grid for a path from a source position S at (5,6) to a destination position D at (1,8).
- Valid actions are move: up, down, left or right.
- Desired solution: path from source to destination with shortest possible length.
  - BFS assumes each move costs the same.
- One solution: (5,6)->(4,6)->(3,6)->(2,6)->(1,6)->(1,7)->(1,8)

Illustration of BFS

-	-	6	5	4	3	4	5	6	-
7	6	5	4	3	2	3	4	5	6
6	5	4	3	2	1	2	3	4	5
5	4	3	2	1	0	1	2	3	4
6	5	4	3	2	1	2	3	4	5
7	6	5	4	3	2	3	4	5	6
-	7	6	5	4	3	4	5	6	-
-	-	-	6	5	4	5	6	-	-
-	-	-	-	6	5	6	-	-	-
-	-	-	-	-	6	-	-	-	-

-	-	-	46	31	21	36	53	-	-
-	59	44	29	17	10	22	37	54	-
57	42	27	15	7	3	11	23	38	55
40	25	13	5	1	0	4	12	24	39
56	41	26	14	6	2	9	20	35	52
-	58	43	28	16	8	19	34	51	-
-	-	-	45	30	18	33	50	-	-
-	-	-	-	47	32	49	-	-	-
-	-	-	-	-	48	-	-	-	-
-	-	-	-	-	-	-	-	-	-

- Left figure shows depth at which a square was encountered in the search tree.
  - Yellow squares correspond to squares **expanded and explored** (**in closed queue**).
  - Green squares correspond to squares **expanded but not explored** (**in open queue**). These squares are the leaves/fringe of the search tree.
- Right figure shows the order (time) at which a square was explored.
  - BFS works out from the source in all directions uniformly.
- NOTE THAT WE KEPT TRACK OF PARENT POINTERS WHILE SEARCHING IN ORDER TO TRACE THE PATH.

Performance of BFS

- Need to judge the performance of BFS according to our 4 criteria:
  - **Complete? YES.**
    - BFS is systematic and will find a solution (if one exists).
  - **Optimal? YES**
    - If action/path cost is equal to depth.

Performance of BFS

- Assume branching factor **b** and assume goal is the **last** node at level **d**.
  - Number of **expanded** nodes is given by:

$$1 + b + b^2 + b^3 + \cdots + b^d + (b^{d+1} - b) = O(b^{d+1}).$$

  - **Time complexity? O(b^{d+1})**
    - Can assume CPU time proportional to #nodes expanded.
  - **Space complexity? O(b^{d+1})**
    - Need to store all nodes to trace solution path back from goal to root.

Uniform Cost Search (UCS)

- BFS only optimal when action costs are equal (lowest cost goal is guaranteed to be explored first!)

- If action costs are **not** equal, then we can alternatively **expand the lowest cost node on the fringe, rather than the shallowest node**.

- This leads to a modification to **BFS** called **Uniform Cost Search.**
  - The modification is simply to sort the open queue according to path cost prior to selecting the node to expand.

Performance of UCS

- For BFS, let $g(n)$ = path cost from root to some other node n and equals the depth of node n in the tree (i.e., $g(n)$ = DEPTH(n)).

- For UCS, $g(n)$ is sum of action costs. UCS is complete and optimal if $g(successor(n)) > g(n) + \varepsilon$; i.e.,

  - The cost of a child node is ε larger than the cost of its parent.

  - Implies the shortest path to each node is found first.

  - Implies the shortest path to any goal node is found first.

Performance of UCS

- Let C^* be the cost of the path to the goal and let the smallest action cost be $\varepsilon > 0$.
- Could be true that we need at least enough actions (costing only ε each) to accumulate total cost C^*.
- Hence, both the time and space complexity of UCS is: $O(b^{\lceil C^*/\varepsilon \rceil})$
  - Just like in BFS, assume CPU time proportional to #nodes expanded.
  - Just like in BFS, we need to store all nodes in order to trace back the path from root to goal.

Depth First Search (DFS)

- Uninformed search strategy.
- Ignores action costs (equivalently, assumes all actions cost 1).
- In terms of the search tree, **explores the deepest nodes first**.

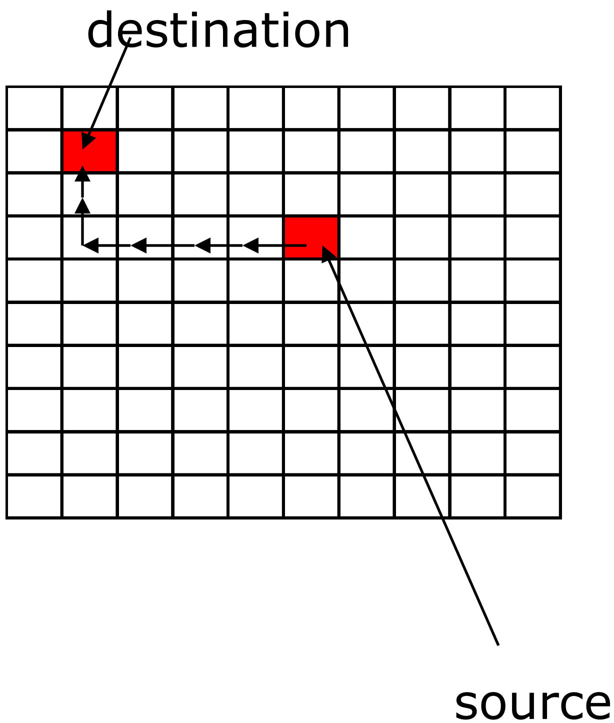
- Algorithm:
  - Uses an open and closed queue.
  - Open queue holds states that have been expanded, but not explored.
  - Closed queue holds states that have been explored (avoid repeats).

- The open queue is LIFO (last-in, first out) which guarantees the depth-first exploration of the search tree.

Pseudo-code for DFS

```
open_queue.insert(init_state);  closed_queue.clear();
while (open_queue.size() != 0) {
    curr_state = open_queue.remove_front(); // LIFO!!!
    if (is_goal(curr_state)) {
         return success; // and solution.
    } else {
        closed_queue.insert(curr_state);
        child_state = expand(curr_state);
        for (i = 1 ; i <= child_state.size() ; i++) {
            if (!open_queue.find(child_state[i]) &&
                     !closed_queue.find(child_state[i])) {
                  open_queue.push_front(child_state[i]); // LIFO!!!
             } else ; // already expanded or explored.
        }
    }
}
return failure; // no solution found.
```

Illustration of DFS



- Search grid for a path from a source position S at (5,6) to a destination position D at (1,8).
- Valid actions are move: up, down, left or right.
- Desired solution: path from source to destination with shortest possible length.
  - DFS assumes each move costs the same.
- One solution: (5,6)->(6,6)->(7,6)->(8,6)->(9,6)->(9,7)->(9,8)->(9,9)->(8,9)->(7,9)->(7,8)->(6,8)->(5,8)->(5,9)->(4,9)->(3,9)->(3,8)->(3,7)->(3,6)->(3,5)->(4,5)->(4,4)->(5,4)->(6,4)->(7,4)->(8,4)->(9,4)->(9,3)->(9,2)->(9,1)->(9,0)->(8,0)->(7,0)->(7,1)->(7,2)->(6,2)->(5,2)->(5,1)->(5,0)->(4,0)->(3,0)->(3,1)->(3,2)->(3,3)->(2,3)->(2,4)->(1,4)->(1,5)->(1,6)->(1,7)->(1,8)
  - WHICH IS NOT NEAR TO OPTIMAL! HOW DID THIS HAPPEN?

Illustration of DFS

-	-	16	15	14	13	10	9	8	7
-	50	17	16	13	12	11	10	7	6
50	49	18	17	18	1	2	3	4	5
49	48	19	18	1	0	1	2	3	4
48	47	20	19	20	1	2	3	4	5
47	46	45	20	21	22	23	24	25	26
-	45	44	43	22	23	24	25	26	27
-	-	43	42	37	36	35	34	29	28
-	-	42	41	38	37	34	33	30	29
-	-	41	40	39	38	33	32	31	30

-	-	-	15	14	13	-	9	8	7
-	51	-	16	-	12	11	10	-	6
-	50	-	17	18	-	-	-	-	5
-	49	-	19	-	0	1	2	3	4
-	48	-	20	21	-	-	-	-	-
-	47	46	-	22	23	24	25	26	27
-	-	45	44	-	-	-	-	-	28
-	-	-	43	-	37	36	35	-	29
-	-	-	42	-	38	-	34	-	30
-	-	-	41	40	39	-	33	32	31

- Left figure shows depth at which a square was encountered in the search tree.
  - Yellow squares correspond to squares **expanded and explored** (**in closed queue**).
  - Green squares correspond to squares **expanded but not explored** (**in open queue**). These squares are the leaves/fringe of the search tree.
- Right figure shows the order (time) at which a square was explored.
  - DFS works out from the source always trying to push “forward”.
- NOTE THAT WE KEPT TRACK OF PARENT POINTERS WHILE SEARCHING IN ORDER TO TRACE THE PATH.

Illustration of DFS (Slight modification to implementation)

-	-	-	-	-	-	-	-	-	-
-	6	-	-	-	-	-	-	-	-
6	5	4	3	2	1	-	-	-	-
5	4	3	2	1	0	1	-	-	-
-	5	4	3	2	1	-	-	-	-
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-

-	-	-	-	-	-	-	-	-	-
-	6	-	-	-	-	-	-	-	-
-	5	-	-	-	-	-	-	-	-
-	4	3	2	1	0	-	-	-	-
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-

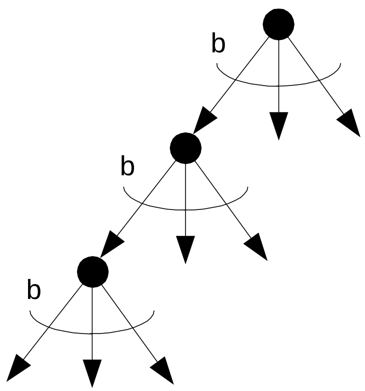
- A slightly different implementation (but still DFS) gives entirely different performance!
  - Gives a path (5,6)->(4,6)->(3,6)->(2,6)->(1,6)->(1,7)->(1,8) which is optimal.
  - Can you see what the difference is in terms of the algorithm's implementation?

Performance of DFS

- DFS is potentially good if the solution is known to be deep in the tree.
- **Optimal**? NO
  - DFS has the potential to go down the **wrong path** and may miss a "shallow goal" in favor of a "deeper goal".
- **Complete**? NO (theoretically)
  - Search tree can be infinite in depth (if not accounting for repeated states), so DFS can get "stuck going deeper" forever.
  - In practice, it is complete.

Performance of DFS

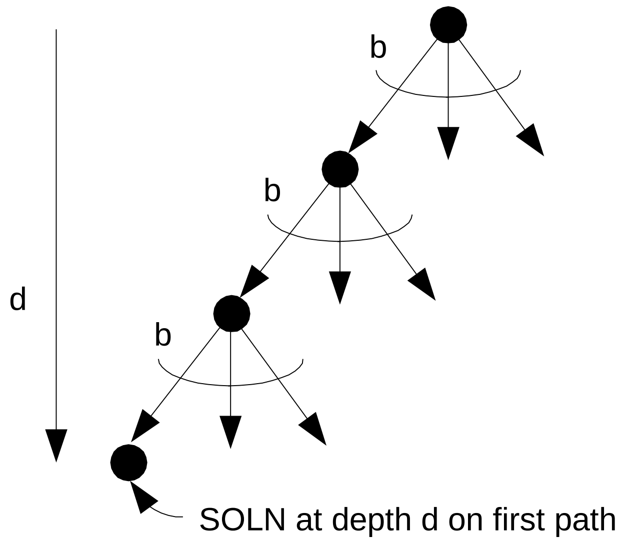
- Assume branching factor **b**, solution at depth **d**, and maximum tree depth of **m**.
- Only need to store the path from root node to current leaf node, and to those unexpanded nodes on the fringe.
- Space Complexity? $O(bm)$
  - Only need to keep track of current path.



- Time complexity? $O(b^m)$
  - Might need to expand the entire tree.

Comments on DFS

- Performance was in the worst case (as it should be) for time and space complexity.
- Often, DFS can be **MUCH** better.
  - Consider when solution happens to be on the first path considered…
  - DFS can be extremely fast, and consume very little time and space (at the price of loss of optimality).



$O(bd)$ time complexity/nodes expanded

Depth-Limited Search

- A simple modification to DFS.
  - Imposes a cutoff at depth l in the search tree (basically, prevents continuation of the search down any given path too far).
  - Makes DFS complete, **if** the solution happens to be at level <= l.
  - Still not optimal.

- Also bounds space and time complexity:

$O(bl)$ space complexity

$O(b^l)$ time complexity

Iterative-Deepening Search

- Another simple modification to DFS.
  - Tries to combine the different benefits of DFS and BFS.
- Use depth-limited search (i.e., DFS with cutoffs).
- Continually increases depth limit (i.e., l = 1, l =2, l = 3, etc…) until solution found.
- This search is complete and optimal (if equal action costs) since, in effect, it goes level by level.
- Requires modest memory requirements of DFS for any particular depth limit.
- However, **does require restarts** (implying **repeating some previously done work**).

Summary of Uninformed Search

- Branching factor is b, maximum tree depth is m, optimal cost is C*, depth limit is l, solution at depth d.

Criterion	BFS	Uniform Cost	DFS	Depth Limited	Iterative Deepening
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes[1]	Yes[3]	No	No	Yes[1]
Complete?	Yes[2]	Yes[2]	No	No	Yes[2]

- **Note 1:** assuming equal action costs.
- **Note 2:** assuming finite branching factor.
- **Note 3:** assuming actions costs are at least $\varepsilon > 0$.