Classical Search: Uninformed Search

Joe Johnson, M.S., Ph.D., A.S.A.

Goal

- The goal in this lecture is to learn what a search problem is and to learn the algorithms that exist for solving search problems.

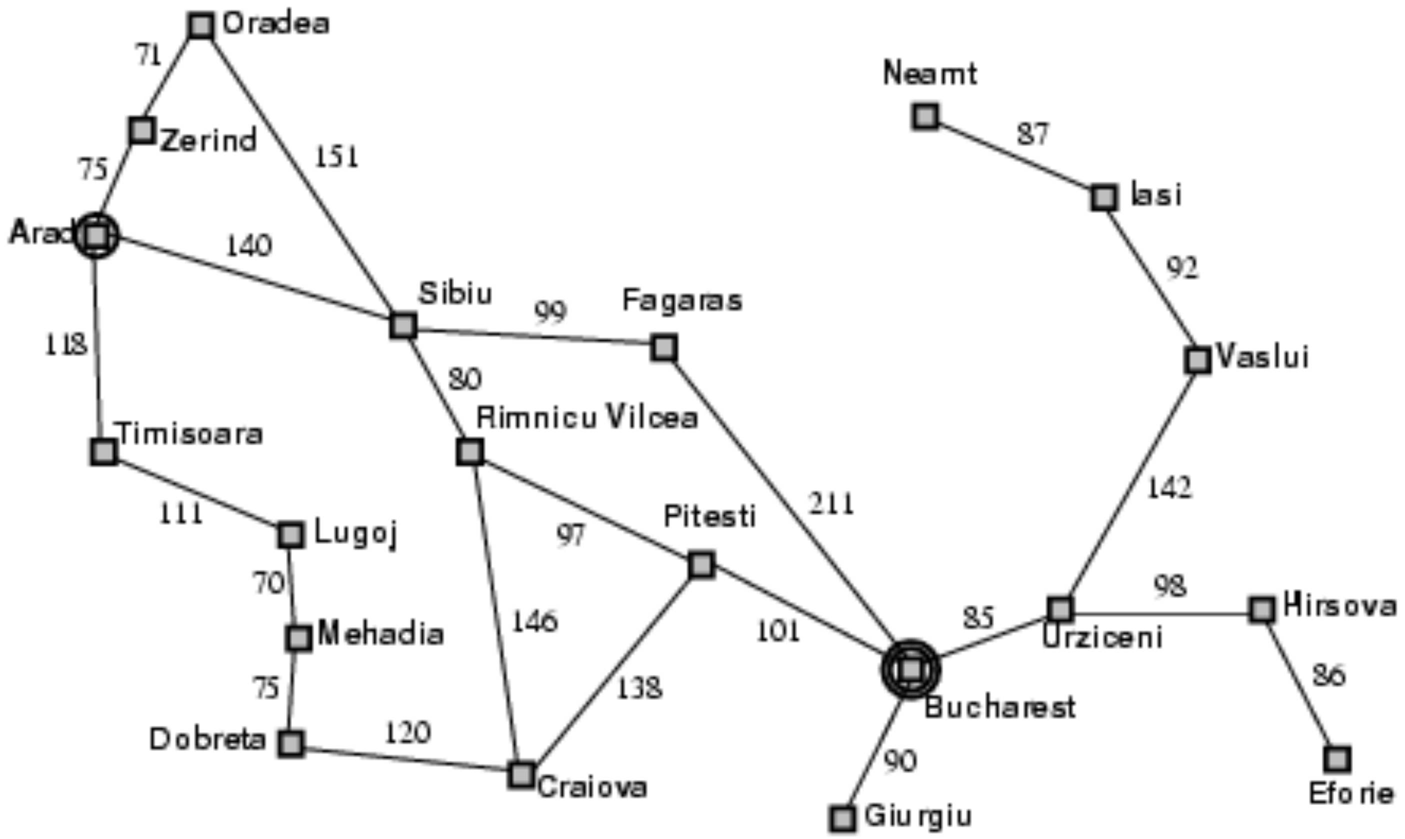
Search Problem

- Definition
  - A search problem is a problem in which an agent must search for a sequence of actions that achieve some state in which some desirable condition is present, as defined by a goal statement.

Problem Solving Agent

- Problem Solving Agent
  - Definition
    - A problem solving agent is a goal-based agent that uses representations of the world to consider future actions and the desirability of their outcome to arrive at a sequence of actions that achieve some goal.
  - Assume for now:
    - Environment is:
      - Fully observable – agent knows its current state (location).
      - Deterministic
      - Discrete – at each state there are only a finite number of actions to consider.

Example: Agent must find path from Arad to Bucharest



Search Problem - Specification

- Specification
  - A search problem is specified by the following four elements:
    - Initial state
      - the state in which the agent begins its search
    - ACTION(s)
      - the set of actions available to the agent in a given state, s.
    - RESULT(s, a)
      - the result of an action, a, by an agent while in state, s.
    - Goal test
      - a test to determine whether the desirable condition exists in a given state, e.g., (In(Bucharest)).
    - Path cost
      - A function that assigns a numeric cost to each path.

Search Problem - Goal

- Goal Statement
    - a sentence which defines the desirable condition upon which to base the search
- Goal Set
    - a set of states for which some desirable condition is present, as specified in the goal statement
- Goal test
    - A test to see if the agent is in a state that is an element of the goal set.

Search Problem - Solution

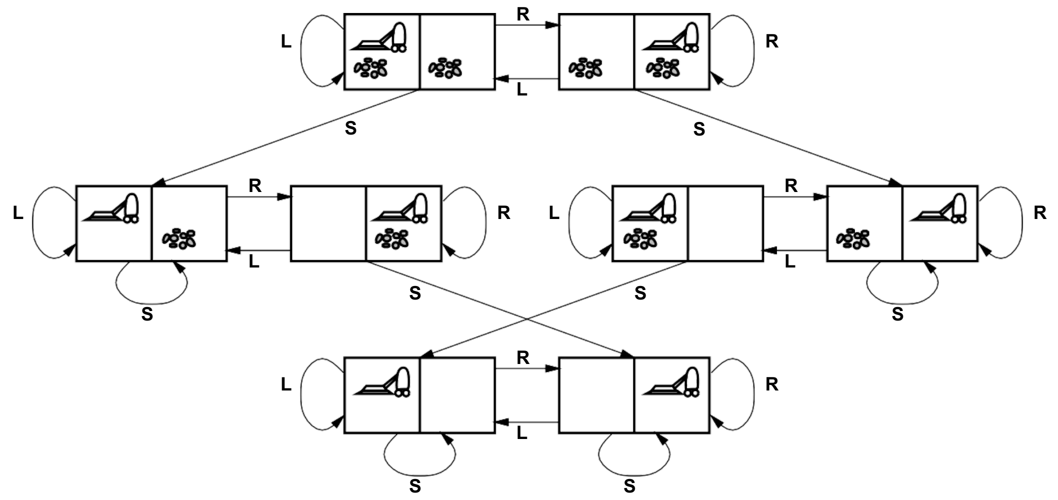
- Solution
  - a sequence of actions which when executed starting from the initial state achieve a state that is a member of the set of goal states.

Example – Vacuum World

- Problem Specification:
  - States
    - State is determined by both the agent location and the dirt locations.
    - The agent is in one of two locations, each of which might or might not contain dirt.
    - Number of *world* states:
      - 2 x 2^2 = 8
      - A larger environment with n locations has n x 2^2 states.
  - Initial state
    - Any state can be designated as the initial state
  - Actions
    - Left, Right, Suck

Example – Vacuum World

- Problem Specification (contd.):
  - Transition model:
    - The actions have their expected effects, except moving Left in the left square has no effect, and corresponding effects for moving Right in the right square.
    - Sucking cleans the square.



Example – Vacuum World

- Problem Specification (contd.)
  - Goal test
    - Check whether all squares are clean.
  - Path cost
    - Each step costs 1 so the path cost is the number of steps in the path.

Search - Algorithms

- General Framework
  - Search Problem Representation
    - Search Graph
      - A search problem is typically represented by a graph in which each node represents a state and each edge represents an action with some predefined cost.
    - Set of explored nodes
      - The set of explored nodes is the set of nodes that the agent has already visited.
    - Set of unexplored nodes
      - The set of unexplored nodes is the set of nodes the agent has not yet visited during the execution of a search algorithm
    - Frontier
      - the frontier is a subset of the set of unexplored nodes which serves as the border between the set of unexplored nodes and the set of explored nodes and from which the next node selected for exploration is chosen.

Search Algorithms – Measurement Criteria

- Measurement Criteria
  - Completeness
    - A search algorithm is complete if it finds a solution whenever one exists.
  - Optimality
    - A search algorithm is optimal if it finds the solution with minimum path cost.
  - Time Complexity
    - is a measure of the number of steps required by the search algorithm, as a function of the branching factor, b, and depth, d.
  - Space Complexity
    - is a measure of the amount of space required to carry out the search, as a function of the branching factor, b, and the depth, d.

Types of Search Algorithms

- Types of Search Algorithms
  - Uninformed Search Algorithms
    - Breadth First Search
    - Uniform Cost Search
    - Depth First Search
    - Limited Depth First Search
    - Iterative Deepening Depth First Search
  - Informed Search Algorithms
    - Best First Search
    - Greedy Best First Search
    - A* Search

Types of Search Algorithms

- Uninformed Search Algorithm
  - Definition
    - An uninformed search is one in which the agent has no more information than that made available by the problem statements

Breadth-First Search (BFS)

- Breadth-First Search Algorithm
  - A BFS algorithm is a search algorithm in which the agent traverses the search space according to the following rule:
    - Select the next node from the frontier for exploration which has the minimum depth from the initial state node.

Breadth-First Search (BFS)

- Description
  - Suppose we are given a graph G = (V,E) and two particular nodes, s and t. We'd like to find an efficient algorithm that answers the following question:
    - Is there a path from s to t in G?
      - This is typically called the problem of determining s-t connectivity.

Breadth-First Search (BFS)

- Description (contd.)
  - In BFS, we search outward from our starting node, s, in all possible directions, adding nodes one layer at a time.
    - Start with node s.
    - Visit each node v for which there exists an edge from s to v: first layer.
    - Visit each node w for which there exists an edge from a node in the first layer to w: second layer.

Breadth-First Search (BFS)

- Consider the following example:

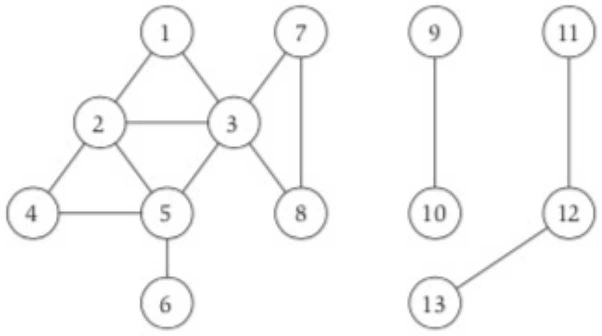


Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

Breadth-First Search (BFS)

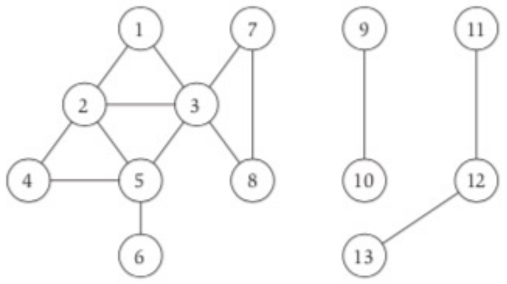
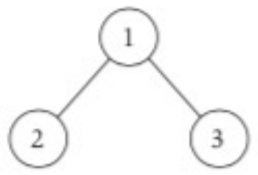


Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

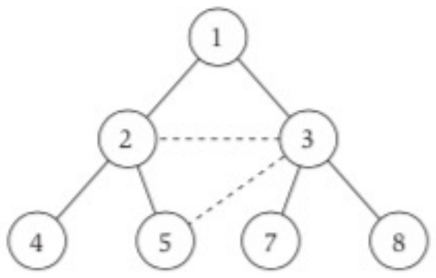
- Start with node 1 as our starting node.
- First layer: 2 and 3
- Second layer: 4, 5, 7, 8
- Third layer: 6
- Terminate - no more new, connected nodes

Breadth-First Search (BFS)

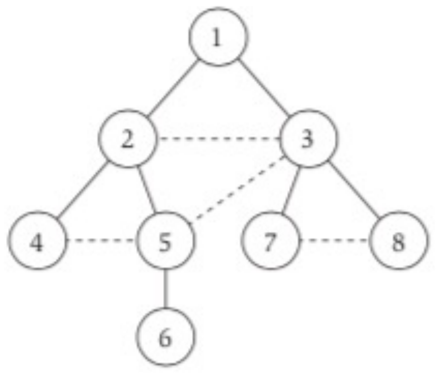
- Breadth First Search Tree:



(a)



(b)



(c)

BFS - Implementation

```
bfs(Graph g = (V,E), Node start_node):
    visited_nodes = {}                                   // dictionary of visited nodes, initially empty
    frontier = []                                        // set up queue, initially empty

    start_node.predecessor = None                        // start node has no predecessor
    visited_notes.add(start_node)                        // add start node to visited nodes
    frontier.put(start_node):                            // load start node into back of queue

    while not frontier.empty():                          // repeat while queue is not empty
        current_node = frontier.get()                    // remove the node from the front of queue
        for neighbor_node in current_node.unvisited_adjacent_nodes():   // visit each unvisited neighbor
            neighbor_node.predecessor =  current_node    // maintain a trail of bread crumbs
            if neighbor_node.is_goal_state():            // does this node represent a goal state?
                return path: start_node→neighbor_node.   // if so, we are done – return path
            visited_nodes.add(neighbor_node)             // mark neighbor as visited
            frontier.put(neighbor_node)                  // load neighbor node into back of queue

    return None                                          // if we made it here, no path to goal found
```

Breadth-First Search (BFS)

- Measurement Criteria
  - Completeness
    - Yes
  - Optimal
    - Yes, when all edges have the same cost
  - Time complexity
    - O(b^d) – expensive
  - Space complexity
    - O(b^d) – expensive

Depth-First Search (DFS)

- Depth-First Search
  - Definition
    - A DFS is a search algorithm in which the agent traverses the search space according to the following rule:
      - Select the next node from the frontier for exploration which has the maximum depth (distance from the starting node).

Breadth-First Search (BFS)

- Consider our example graph once again:

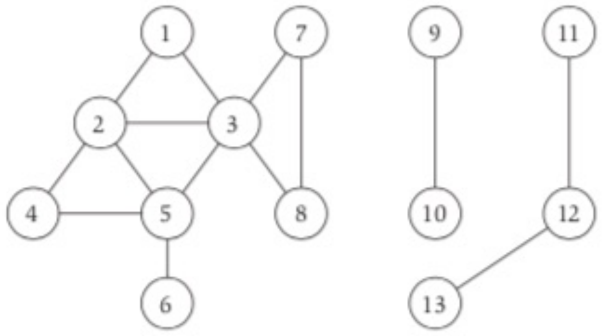
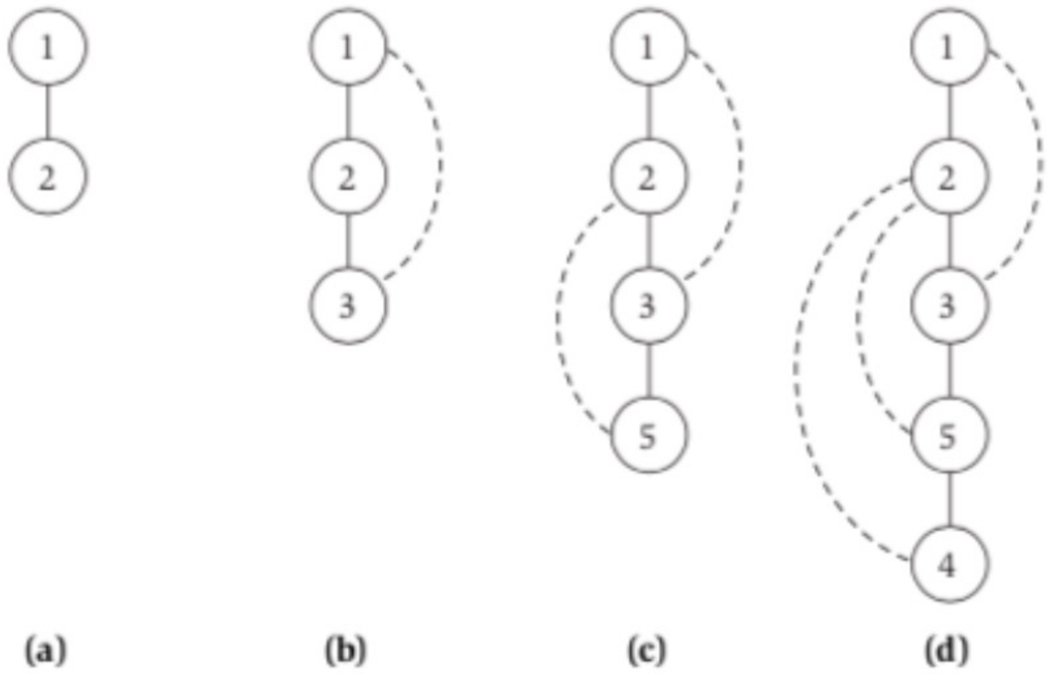
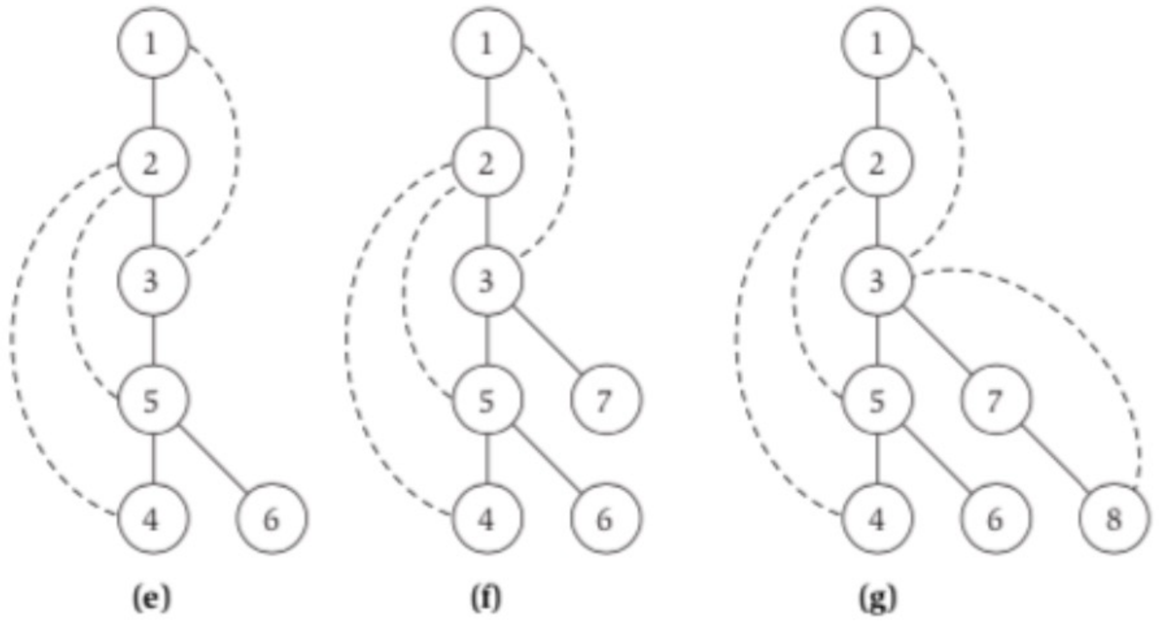


Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

Breadth-First Search (BFS)



Breadth-First Search (BFS)



DFS - Implementation

```
dfs(Graph g = (V,E), Node start_node):
    visited_nodes = {}                                    // dictionary of visited nodes, initially empty
    frontier = []                                         // set up **stack**, initially empty

    start_node.predecessor = None                         // start node has no predecessor
    visited_notes.add(start_node)                         // add start node to visited nodes
    frontier.put(start_node):                             // push start node onto top of stack

    while not frontier.empty():                           // repeat while stack is not empty
        current_node = frontier.get().                    // pop node from top of **stack**
        for neighbor_node in current_node.unvisited_adjacent_nodes():   // visit each unvisited neighbor
            neighbor_node.predecessor =  current_node     // maintain a trail of bread crumbs
            if neighbor_node.is_goal_state():             // does this node represent a goal state?
                return path: start_node→neighbor_node.    // if so, we are done – return path
            visited_nodes.add(neighbor_node)              // mark neighbor as visited
            frontier.put(neighbor_node)                   // push neighbor onto top of stack

    return None                                           // if we made it here, no path to goal found
```

Depth-First Search (DFS)

- Measurement Criteria
    - Completeness
        - No
    - Optimal
        - No
    - Time Complexity
        - $O(b^d)$
    - Space complexity
        - $O(bd)$ - this is what makes DFS attractive.

Uniform Cost Search (UCS)

- Uniform Cost Search
  - A uniform cost search is a search in which the agent traverses the search space according to the following rule:
    - Select the next node, n, from the frontier for exploration which has a minimum value for a path cost function, g(n), which measures cost from the start node to the node n.

Uniform Cost Search (UCS)

- Implementation
  - Order nodes in a priority queue in increasing order of g(n).

Uniform Cost Search (UCS)

- Measurement Criteria
  - Complete
    - Yes
  - Optimal
    - Yes
  - Time Complexity
    - $O(b^d)$
  - Space Complexity
    - $O(b^d)$

Uniform Cost Search Variant: Dijkstra's Algorithm

- Dijkstra's Algorithm:
  - Finds the shortest path from a starting node, s, to every other node in a connected graph.
  - Variant of the uniform cost search algorithm
    - Dijkstra's algorithm does not have a goal state
    - Seeks to traverse the entire graph to which the starting node is connected, as opposed to only traversing just enough to find a goal node.
    - Dijkstra's algorithm requires loading the entire set of nodes into a queue prior to starting the the search, unlike UCS, which has potentially infinite cost if the graph is infinite in size.

Dijkstra’s Algorithm

- Background:
  - Given a directed graph, G = (V, E), and a designated start node, s.
  - We assume that s has a path to every other node,
    - i.e., we have a connected graph.
  - Each edge e has a length, l_e >= 0, which gives the cost/distance/time to traverse the edge.
  - For a path, P, the length of P, given by l(P), is the sum of the lengths of the edges in P.
  - Note that although this problem is specified for a directed graph, we could extend this to an undirected graph simply by using a directed graph with edges in both directions.

Dijktra's Algorithm

- The algorithm maintains a set S of vertices, u, for which we have determined a shortest-path distance d(u) from s.
  - We call this the explored part of the graph.
- Initially S = {s} and d(s) = 0.
- For each node v in V - S we determine the shortest path that can be constructed by traveling along a path through the explored part of S to some u in S, followed by a *single* edge, e = (u,v). That is, we consider the quantity, $d'(v) = \min_{\{e = (u,v): u \text{ in } S\}} d(u) + l_e$.
- We choose the node, v in V - S for which this quantity is minimized, add v to S, and define d(v) to be the value d'(v).

Dijkstra’s Algorithm

- Execution of Dijkstra’s Algorithm:

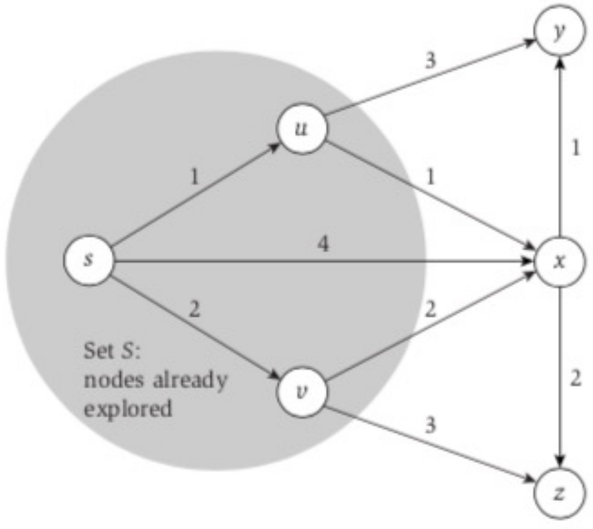


Figure 4.7 A snapshot of the execution of Dijkstra’s Algorithm. The next node that will be added to the set S is x, due to the path through u.

UCS - Implementation

```
ucs(Graph g = (V,E), Node start_node):
    start_node.predecessor = None                                   // start node has no predecessor
    frontier = []                                                   // set up **priority queue** for frontier, initially empty
    frontier.put(start_node):                                       // put start node in frontier

    explored_nodes = []                                             // set up list of explored nodes, initially empty

    while not frontier.empty():                                     // repeat while frontier not empty
        current_node = frontier.get().                              // get node from front of frontier (priority queue)
        explored_nodes.add(current_node)                            // add node to list of explored nodes
        if current_node.is_goal_state():
            return path: start_node→neighbor_node.                  // if so, we are done – return path
        for neighbor_node in current_node.unexplored_adjacent_nodes():   // visit each unexplored neighbor
            neighbor_node.predecessor =  current_node               // maintain a trail of bread crumbs
            frontier.put(neighbor_node, path_cost)                  // put neighbor node in search frontier, path_cost score
            if frontier.count(neighbor_node) > 1:                   // keep only 1 copy of neighbor node with min path cost
                keep neighbor_node with min(path_cost), remove others // g(n) is score for priority queue
    return None                                                     // if we made it here, no path to goal found
```

Limited Depth-First Search (LDFS)

- Limited Depth-First Search
  - Definition
    - A LDFS is a search algorithm modeled after DFS with the following restriction:
      - the DFS does not extend to nodes of depth greater than some maximum depth, d = L.

LDFS

- Measurement Criteria
  - Complete
    - No – if the solution with min depth is $d > L$.
  - Optimal
    - No - if the solution with min depth is $d > L$.
  - Time Complexity
    - O(b^L)
  - Space Complexity
    - O(bL)

Iterative Deepening Depth-First Search (IDS)

- Iterative Deepening Depth-First Search (IDS)
  - Definition
    - An IDS searches is a search algorithm modeled after LDFS, structured as follows:
      - Conduct a LDFS with L = 1.
      - Conduct a LDFS with L = 2.
      - Conduct a LDFS with L = 3, and so on.

IDS

- Benefits
  - Combines the benefits of DFS and BFS
  - Like DFS, memory requirements are modest - O(bd).
  - Like BFS, it is complete.
  - Like BFS, it is optimal (when path cost is a non-decreasing function of the depth of the node).

IDS

- Is it wasteful?
  - We’re generating the same nodes multiple times, after all.
  - Not as bad as we might think, initially.
  - Most of the nodes are in the bottom level of the search tree.
  - Does not really matter too much how many times the nodes are reproduced at higher levels in the tree.
    - Bottom level nodes produced once (depth = d)
    - Next level up, nodes produced twice (depth = d-1)
    - Next level after that, nodes produced 3 times (depth = d-2)
    - N(IDS) = $(d)b + (d-1)b^2 + (d-2)b^3 + \ldots. + (1)b^d$
    - For b = 10 and d = 5, we have:
      - N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123,450
      - N(BFS) = 10 + 100 + 1000 + 10000 + 100000 = 111,110
      - Overhead = (123,450 – 111,110) / 111,110 = 11%

IDS

- Measurement Criteria
  - Complete
    - Yes
  - Optimal
    - No – if a shallower solution has greater path cost than a deeper solution.
  - Time complexity
    - $O(b^d)$
  - Space Complexity
    - $O(bd)$ - where d is the depth of the shallowest solution