Study Notes for AI (CS 440/520)
Lecture 2: Uninformed Search

Corresponding Book Chapters: 3.1-3.2-3.3-3.4
Note: These notes provide only a short summary and some highlights of the material covered in the corresponding lecture based on notes collected from students. Make sure you check the corresponding chapters. Please report any mistakes or any other issues with the notes to the instructor.

1 AI Problems

Four components (P.E.A.S) form an AI problem:

1. a performance measure,
2. the environment where an intelligent agent is located,
3. the agent's actuators, as well as
4. the agent's sensors.

As shown in Figure 1 the intelligent agent is interacting with the environment through its sensors and actuators. It gets data from the environment with its sensors, processes them, makes a decision and puts that decision into action with its actuators. A performance measurement is used to evaluate the agent's performance.

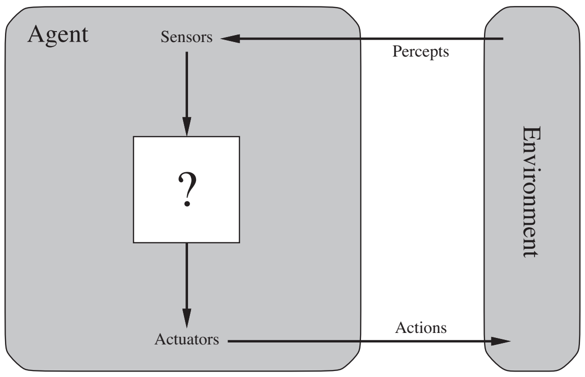

Figure 1: An intelligent agent.

2 Types of Agents

There are 4 main classes of agents: reflex, model-based reflex, goal and utility based agents.

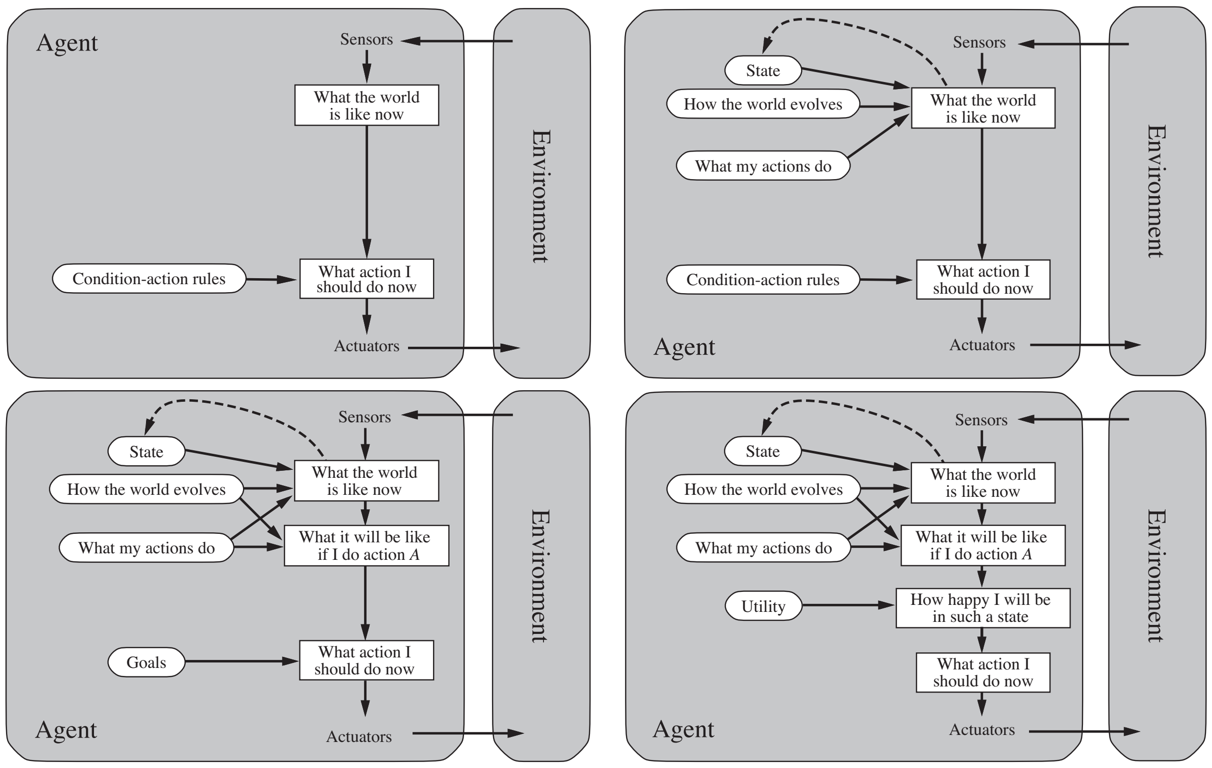
Agent
Sensors
What the world is like now
Condition-action rules
What action I should do now
Actuators
Environment
State
How the world evolves
What my actions do
What the world is like now
Condition-action rules
What action I should do now
Agent
Actuators
Environment
Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment
Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Table 1: The components for each agent type (top left: simple reflex agent, top right: model-based reflex agent, bottom left: goal-based agent, bottom right: utility-based agent)

These all build on each other beginning with the **reflex agent**. This is the simplest class of agent, which simply acts directly on the perceived state of its environment, typically accomplished through "if-else" statements. A simple example of this is a thermostat.

Building on the reflex agent is the **model-based reflex agent**, which adds memory to store and update the current state based on perceptual input. Using this memory the agent is able to make better decisions.

The **goal-based agent** incorporates a model-based agent and adds a goal. The agent must then take all of the information that it has in memory and what it knows about the goal to reason about the best action or sequence of actions to take.

The **utility-based agent** is the most complex version, building upon a goal-based agent and adding a utility function to every state. This utility function is used by the agent to determine how much is the performance measure satisfied on each state.

3 P.E.A.S for Goal Based Agents

The environment corresponds to a set of states, denoted by X, and the state in which the agent begins is denoted X_0. Actuators allow the agent to interact with or move through the environment. This is achieved by using a successor function: $X \times U \to X$, where U denotes the action space. For goal-based agents the environment is fully observable, meaning that the agent has full knowledge of the environment at all times. This has a few implications. For instance, the agent must know where it is in the state space, which states are goal states and how the states are connected to each other. To measure performance the agent must be aware of the goal states, $G \subset X$. Typically goal-based agents aim to optimize the path cost to a goal state.

4 Toy Problems

Toy problems are typically simple, exemplary problems that can shed some light to the challenges of more complicated, realistic problems.

4.1 Vacuum World

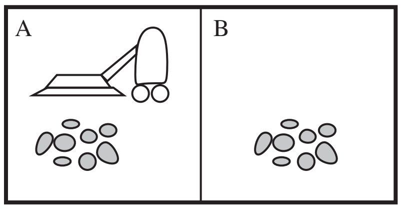

Figure 2: 2 Cell Vacuum World

The vacuum world is a simple problem which is set in a strip of N "rooms". Any room can be dirty or clean. The vacuum is placed in a random cell and there is no prior knowledge of the distribution of which rooms are dirty. The size of the state space is $\mid X \mid = N \cdot 2^N$, where N is the number of cells. For $N = 2$, as in the above figure, the size of the state space is 8. The initial state X_0 might be any cell in the space. The vacuum is able to sense which cell it is in, and whether or not the cell is clean. In order to interact with the environment the vacuum can move either left or right, clean the current cell or do nothing. To measure how well algorithms do in this problem a point is awarded for each clean cell at each time step. A simple reflex agent for the above problem can be formulated as follows:

```
is the cell dirty?
    yes:
        clean it
    no:
     switch cells
```

4.2 8 Puzzle

The goal of this problem is to take a 3×3 grid, as shown in Figure 3, and arrange the numbers in a sequence. For the 3×3 grid the size of the state space is:$\mid X \mid = 9!/2 = 181,440$ (the divide by 2 treats mirrored states as being the same). For the 15 puzzle (4×4 grid) $\mid X \mid = 1.3$ trillion states and for the 24 puzzle (5×5 grid) $\mid X \mid = 10^{25}$ states. X_0 is any arrangement of the tiles on the board. The sensor for this problem is just knowing where all of the tiles, and the empty tile, are on the board. The possible actions at each time step are: moving up, down, left or right the empty tile. The best solution is the one that takes the fewest steps to go from an arbitrary start to the solution.

4.3 8 Queens Problem

Another popular problem is the 8-queens problem. The goal is to place 8 queens on a chess board so that no queen threats any other. This means that any vertical, horizontal or diagonal line can have at most one queen. Figure 4

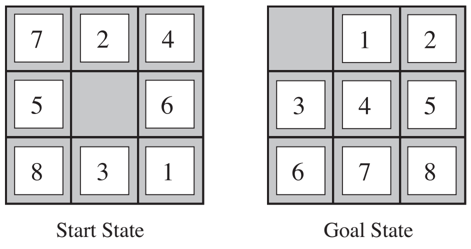

Figure 3: 8 puzzle unsolved and solved

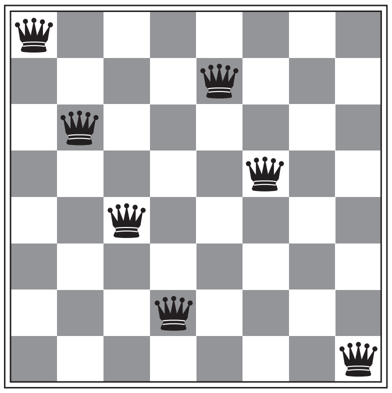

Figure 4: Sample 8 queens board, not a valid solution

shows an infeasible solution. The number of states in this problem depends on how you handle the problem. If we place all the queens randomly and check if this is a feasible solution, then we get $\binom{64}{8} = 4,426,165,368$ states which is too high. If we solve the problem incrementally by putting a queen to a valid (unthreatened) square each time, we will see that the number of states reduces significantly to only $2,057$. Making it a little bit more systematic by putting the first queen to the first line and the second to the second line and so on, it will make it easier to find the algorithm and make the problem much easier to solve. If we had 100 queens instead of 8, the problem would grow into a 10^{400}-state problem with the first method. Even the incremental method would have 10^{52} states. Consequently, the problem is infeasible for both approaches.

Another important metric affecting the running time is the number of goal states. Although there are only two goal states in 8-puzzle, the 8-queen problem has 96. This makes the 8-queen puzzle an easier problem because finding any of them will solve the problem.

4.4 Route-Finding Problems

Route-finding problems are fully observable instances (the agent can see the whole map) in which the goal is to find the shortest path from a specified start node to a specified end node on a graph (or grid). This class of problems is well suited to goal based agents because the performance measure is based on getting to the goal as efficiently as possible, requiring the agent to be able to better rationalize how to reach the goal.

The following section describes methods for addressing problems like the above toy problems and similar instances,

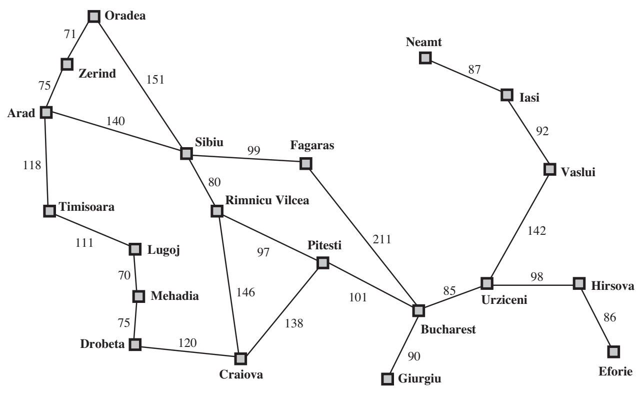

Figure 5: A map of the Romanian road system

which are appropriate for goal-based agents.

5 Tree Search

When we attempt to solve a problem with a search-based approach we start at an initial state. The various valid actions from the initial state result in more states that we can transition to. When we consider these new states, they lead to more new states and so on. This process forms a tree of states. The goal state or states are also somewhere in this tree. The process of finding the goal state is called "tree searching" and the different searching methods are called "tree search algorithms".

However, a tree of states is not enough to find the solution. We need to store additional information together with these states. So, we use trees in which nodes include 4 more attributes besides the state information. These trees are called "search trees". The node in Figure6 is a sample node from the 8-puzzle problem search tree. A node stores the following attributes:

- a state
- a link to the parent node which stores the last state before the current one
- the last action taken before moving to this state
- the depth of the node
- the path cost from the root to the current node

Figure 5 shows the map of Romania. Lets say we want to go from Arad to Bucharest by using the shortest way. In this problem every city can be considered as a state. As we can see from the map there are many alternative paths from Arad to Bucharest and they have different distances. The length of the road Arad - Sibiu - Fagaras - Bucharest is 450 while Arad - Sibiu - Rimnicu Vilcea - Pitesti - Bucharest is 418. So, we will have two different nodes having the same state but different path costs. The node corresponding to the Bucharest state and the minimum path cost is the optimum goal.

At the beginning we have the initial node with these attributes:

- State - Arad
- Parent Node - none
- Action - none

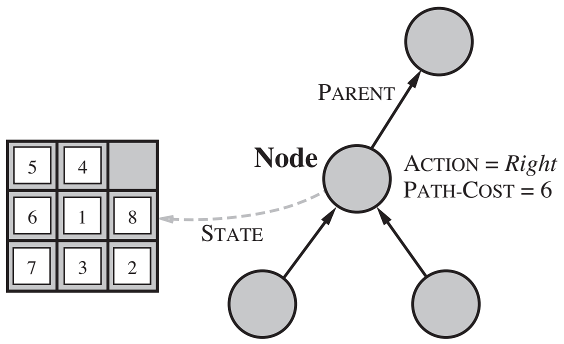

Figure 6: How a state from the 8 puzzle translates to a node

- Depth - 0
- Path Cost - 0

We start constructing our search tree by expanding the first node. From Arad we can go to only three cities. When we expand the first node we will have 3 new nodes as in Figure 7. If we do not have the goal state yet, we need to choose another node and expand that one too. When we expand Sibiu this time we will have the second tree in Figure 7.

Notice that now we have another node corresponding to the city of Arad just like the root node. But this node has depth 2 and path cost 280. If we keep going Arad - Sibiu - Arad - Sibiu - Arad ... we will never be able to reach Bucharest. It means that the search trees are not always finite.

5.1 Fringe

The set of nodes from which we can select from the next choice is called "fringe". The fringe includes the leaves of the tree which are the nodes that do not have any children. Initially just contains X_0. The basic functions needed for a queue are:

- insert(element, queue): add element and return the resulting queue
- remove(queue): remove and return the first element in the queue
- empty(queue): return true if no elements
- initialize(element): create a queue that contains element

Algorithm 1 Tree Search

```
$fringe \leftarrow make\_node(initial\_node)$
**loop**
  **if** empty(fringe) **then**
    report failure
  $node \leftarrow get\_node(fringe)$
  $children \leftarrow expand(node)$
  $fringe \leftarrow add(children)$
```

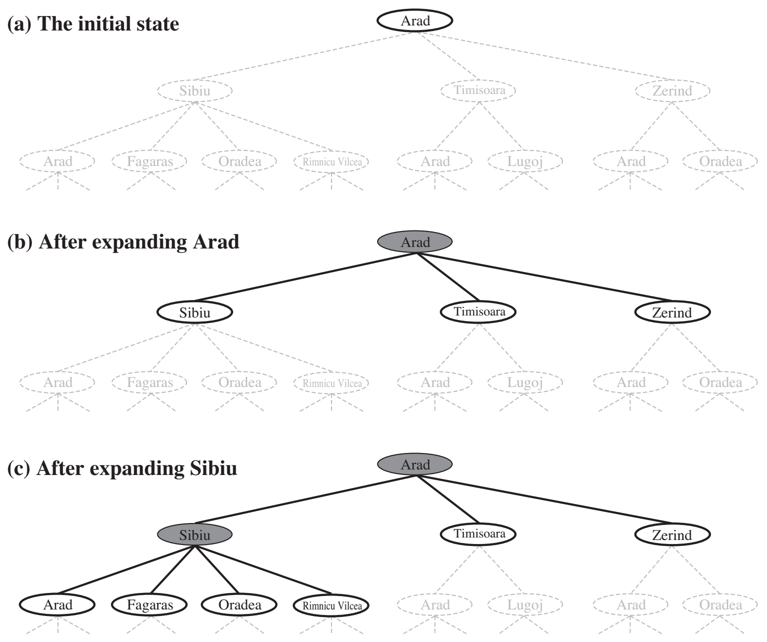

Figure 7: Example of expanding paths taken from the Romania map example

The basic idea of the general search tree algorithm is shown in Algorithm 1. Instantiations of the general algorithm differ in the type of the data structure they use to store the fringe and the method to choose which node will be expanded next.

5.2 Search Algorithm Comparison Criteria And Complexity Parameters

There are four criteria comparing the various search tree algorithms:

1. **Completeness:** Does it return a solution in finite time, if one exists?
2. **Optimality:** Does it return the optimum solution?
3. **Time Complexity:** How many nodes are generated until a solution is found?
4. **Space Complexity:** How many nodes -maximum- we have to remember to find the solution?

The following parameters are used to calculate and compare the performance of an algorithm:

- b: maximum branching factor
- d: the depth of the shallowest goal state
- m: maximum length of path in the tree

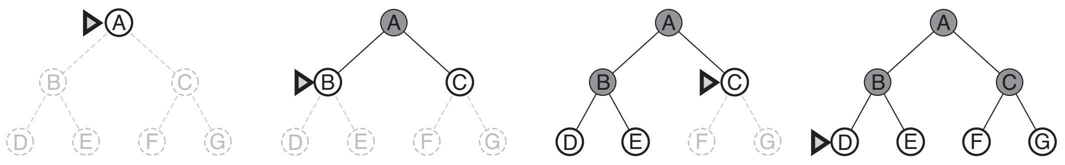

Figure 8: Demonstration of how breadth first search progresses

6 Tree Search Algorithms

6.1 Breadth First Search

Breath-First Search (BFS) expands all the nodes first before expanding the nodes at the next level. As Figure 8 shows, BFS first expands A and then B. And then, before expanding D or E it expands C because it is at a higher level, i.e., has a lower depth. Here are the properties of BFS:

- Data Structure: First In First Out (FIFO) - Queue
- Complete: if b is finite
- Suboptimal: Optimal if all the costs of the edges are equal.
- Time Complexity: $O(b^{d+1})$
- Space Complexity: $O(b^{d+1})$

The data structure is a queue - the first element inserted is the first to be removed. When we expand a node, we insert the node's children to this queue. All the nodes that are already in the queue will be served before the children. The algorithm is complete when we have a finite branching factor. In the cases of weighted edge costs, the optimum solution could be deeper than a non-optimum solution. In the unweighted case, we know that the first solution we get will be the optimum, because it will reach first the goal node with the lowest depth. If the edge costs are equal then the goal node with the smallest depth coincides with the optimal node.

The worst case scenario is having the goal at the right-most node at depth d. Let's take a look at the number of nodes generated until we expand the goal. Before expanding the goal we must have expanded all the nodes at depths from 1 to d and the total number of these nodes is $\sum_{i=0}^{d} b^i$. Moreover, we have generated the children of the nodes at the depth d except the goal node's. The number of these children is $b^{d+1} - b$. Thus, the time complexity is $O(\sum_{i=0}^{d} b^i + b^{d+1} - b) \Longrightarrow O(b^{d+1})$.

We only store the fringe in BFS. The size of the fringe includes in the worst case all the children at depth d+1 except the children of the goal node. The total number is $b^{d+1} - b$ and so the space complexity is $O(b^{d+1} - b) \Longrightarrow O(b^{d+1})$.

6.2 Depth First Search

Depth-First Search (DFS) always selects the left-most node and expands it. It moves to the following sibling node only after it is done with the entire subtree of the current node. As in Figure 9, it first expands A and then B. It will not expand C unless it is done with B's children. Then it expands D and H. After H is done it expands I. And expanding I means that we are done with the whole subtree of D and it expands E and so on. Here are the properties of Depth-First Search:

- Data Structure: Last In First Out (LIFO) - Stack
- Incomplete
- Not optimal

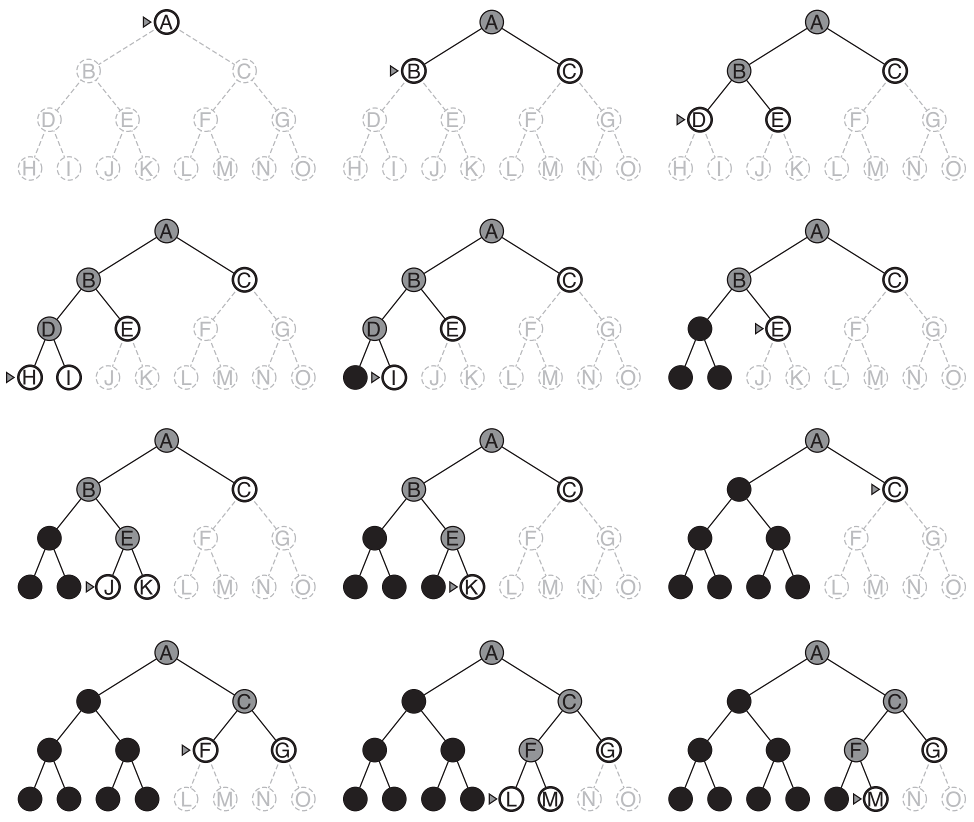

Figure 9: Depth-first Search.

- Time Complexity $O(b^m)$
- Space Complexity $O(bm)$

In DFS, when we expand a node we insert all the children at the top of the stack. If we look at the second tree in Figure 9, B is above C in the stack for this condition. After we expand B, its children will be also above C in the stack, and their children etc. Hence, we have to wait for the whole subtree of B to be expanded before expanding C. If that subtree is not finite e.g. having an infinite loop, then we will never be able to expand C. If the goal is in C's subtree, then we will never reach the goal. So, it is incomplete and also not optimal.

The worst case scenario for depth-first search is that the algorithm has to search the entire tree. This means that the goal node is the right-most node of the tree. Since m the maximum length along the tree, the worst case is that the tree is complete up to level m. In this case we have to generate $\sum_{i=0}^{m} b^i$ node, which results in a time complexity of $O(\sum_{i=0}^{m} b^i) \Longrightarrow O(b^m)$.

Let's take a look at the tree in row 2, column 1 of Figure 9. When we expand a node, what we need to remember are the right siblings of the node's ancestors and of the node. For a node at depth l there are l sets of siblings and at most $b-1$ siblings per set. The worst case is when $l = m$. Consequently, the maximum number of nodes we have to remember is $m(b-1)$ and the space complexity is $O(bm)$.

Summary of Basic Search Algorithms:
Breadth-first search has the advantage of being complete for most reasonable discrete problems and even being optimal in some cases. On the other hand, it has high time and space complexity. Depth-first search is not even complete or optimal. Moreover, its time complexity is worse than BFS. But, it has the best space complexity compared to the alternatives.

6.3 Uniform Cost Search

Uniform Cost Search (`UCS`) calculates the path cost of all the nodes in the fringe and expands the one that has the minimum cost. Figure 10 describes the operation of the algorithm. After the initial node S is expanded, three nodes are generated: A, B and C. Among them node A has the smallest path cost and is selected for expansion, generating node G, a goal node. Then node B is the node with the minimum path cost and is expanded first, also resulting in a goal node. The goal node produced by B is the optimal solution and the node that will be visited first by `UCS`.

There are two parameters used to express time and space complexity for `UCS`:

- C^*: optimal cost
- ϵ: smallest edge cost, $\epsilon > 0$

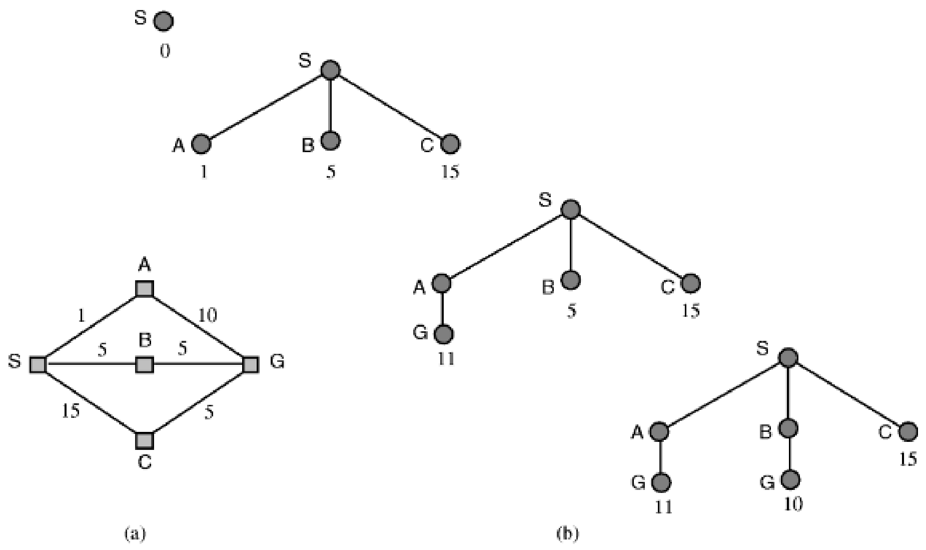

Figure 10: Uniform-First Search.

These are the properties of Uniform Cost Search:

- Complete and Optimal
- Time Complexity $O(b^{\lfloor C^*/\epsilon \rfloor + 1})$
- Space Complexity $O(b^{\lfloor C^*/\epsilon \rfloor + 1})$

It is complete if b is finite and also if the smallest edge cost is $\epsilon > 0$. In the latter case the algorithm will not search infinite paths. At some point along every path the path cost will be greater than other alternatives along the tree due to the minimum edge cost ϵ. Thus, the algorithm is complete in this case. It is optimal because it expands the goal node with the minimum path cost first. The maximum depth we can go is $\lfloor C^*/\epsilon \rfloor$. In the worst case the goal node is at depth $\lfloor C^*/\epsilon \rfloor$ and the algorithm has generated all the nodes down to one level below the goal (as in breadth-first search). So, time complexity is $O(b^{\lfloor C^*/\epsilon \rfloor + 1})$. We only store the nodes in the fringe and the maximum number will be if we have to store all the nodes at the lowest level possible (one level lower than the goal) and which is $\lfloor C^*/\epsilon \rfloor + 1$. So, the space complexity is also $O(b^{\lfloor C^*/\epsilon \rfloor + 1})$.

Overall, uniform cost search is always optimum and complete given certain conditions. Its drawback is that it has even worse time and space complexity than BFS.

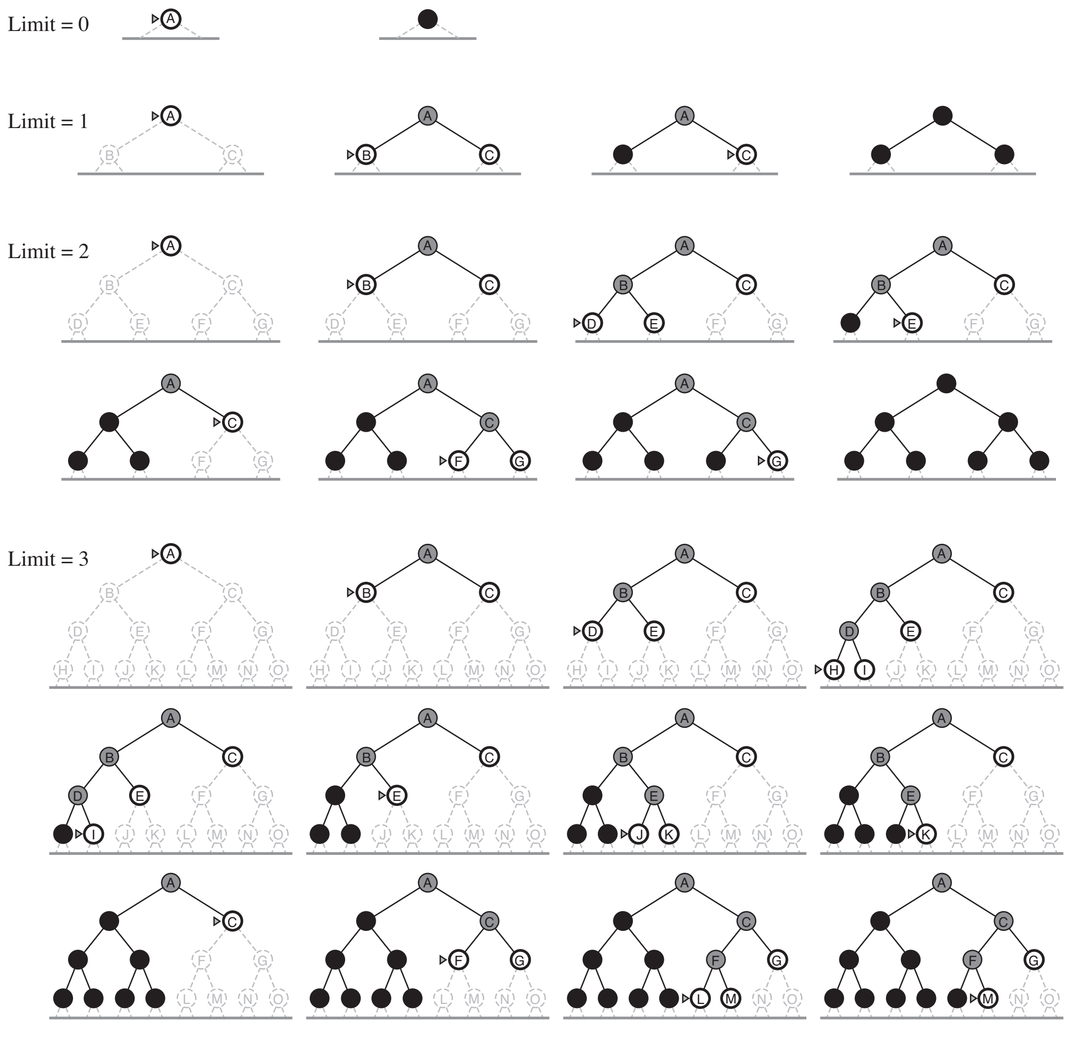

Figure 11: Iterative Deepening Depth-First Search

6.4 Iterative Deepening Depth-First Search

Iterative Deepening Depth-First Search (`IDDFS`) is a depth-limited search, meaning that it traverses the tree the same way as `DFS`, but it stops at a certain depth. It runs repeatedly through the tree, increasing the depth limit with each iteration until it reaches the depth of the shallowest goal. Figure 11 provides an example. At "Limit=2" in the figure, the algorithm goes through the tree until all the nodes are expanded. If no goal node is found then it increases the limit to three and restarts the algorithm. The motivation behind `IDDFS` is to maintain the good space complexity properties of `DFS`, while avoiding the incompleteness properties that arise when the tree contains infinite paths. It can also be seen as a combination of `DFS` and `BFS`: it has the space complexity of `DFS` and the completeness properties of `BFS` . It also has the same optimality properties like `BFS`.

Criteria:

- Complete? Yes, because by iteratively increasing the limit, at some point the algorithm reaches a goal node and the algorithm never searches an infinite path.
- Time: $O(b^{(d)})$. Notice that the time complexity of `IDDFS` is better than `BFS`'s. When the shallowest goal node is at level d, `BFS` will generate nodes at level $d+1$. The number of nodes at level $d+1$ is almost as many nodes on all the levels up to d. On the other hand, `IDDFS` is limited to generate nodes up to level d and consequently expands much less nodes than `BFS`.
- Space: $O(bm)$. Same with the space complexity of `DFS`, since at each point in time `IDDFS` is executing a depth-first search and has only to remember the left siblings of the currently expanded node and of its ancestors. For example in Figure 11, when the node H is expanded at "Limit-3", the fringe includes also the nodes: I, E, C.
- Optimal? Yes, if all the edges have the same cost.

6.5 Bidirectional Search

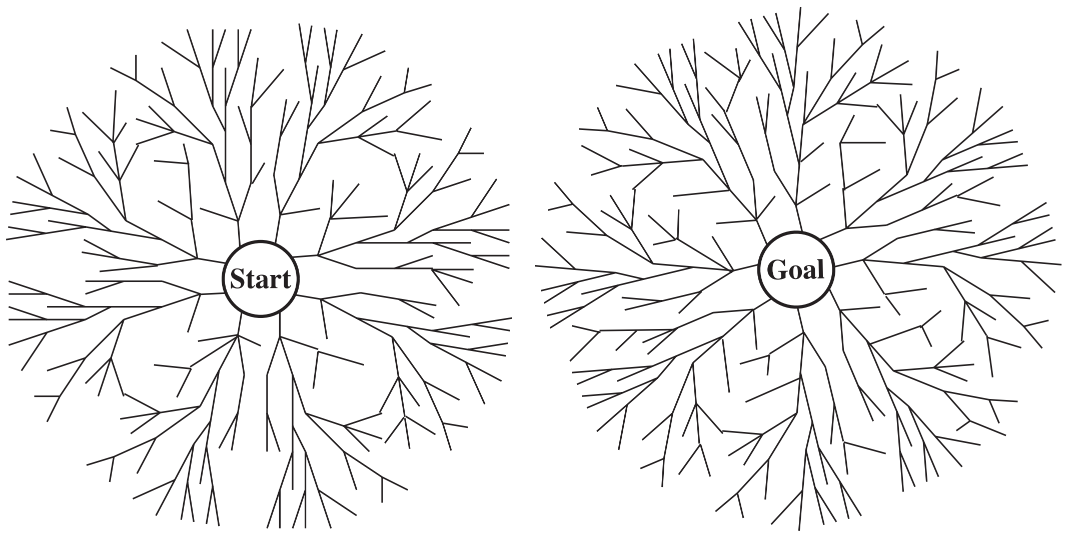

Figure 12: The graph of a bidirectional search tree.

Bidirectional search is a graph search algorithm that runs two simultaneous breadth-first searches: one forward from the initial state, and one backward from the goal. It stops when the two meet in the middle. See Figure 12. The advantage of bidirectional search is that it executes two BFS runs up to level $\frac{d}{2}$, which is much faster than running one BFS up to level d. The approach can also be advantageous when it is much easier to find a solution from the goal to the start. Nevertheless, bidirectional search cannot be always applied because it is not always possible to inverse the problem, i.e., in chess there is an infinite number of goal states and it is infeasible to search backwards from all of them.

Criteria:

- Complete? Yes, in the BFS sense.
- Time: $O(b^{d/2})$, because each BFS algorithm has to search only half the number of levels that the entire solution contains.
- Space: $O(b^{d/2})$.
- Optimal? Yes, if the edges have the same cost, as in BFS.

7 Avoiding Previous States

In many cases it is appropriate to avoid repeated states. Re-expanding already visited states can have detrimental effects in the performance of an algorithm. For instance, a depth-first search could get stuck searching repetitively Node A, Node B, and then again Node A.

By using the principle of **dynamic programming** we can avoid this problem. A dynamic programming algorithm solves every subproblem just once and then saves the solution to the subproblem, thereby avoiding the work of recomputing the answer every time the subproblem is encountered.

When dynamic programming is applied in search the resulting data structure for the search is no longer a tree but a graph, since a state corresponds to a single node and there may be many ways to reach that node from the initial state. Figure 13 shows that an additional data structure beyond the fringe is also needed in this case. This data structure, referred to as the **closed list**, stores all the nodes that have been already expanded.

Algorithm 2 shows the general way to execute search and avoid repeated states by making use of the closed list. The closed list is often implemented with the aid of a hash table. Note that it is not always possible to avoid repeated states. For problems that have huge state spaces it is impossible to remember all the states we have already visited even with a hash table. For such state spaces, the TREE-SEARCH algorithm is the preferred approach.

Algorithm 2 GRAPH-SEARCH

```
$closed \leftarrow$ an empty set
$fringe \leftarrow make\_node(initial\_node)$
loop
  if empty(fringe) then
    report failure
  $node \leftarrow get\_node(fringe)$
  if $is\_goal(node)$ then
    return $node$
  if $node$ not in $closed$ then
    add $node$ in $closed$
    $children \leftarrow expand(node)$
    $fringe \leftarrow add(children)$
```

Examples of cases where we need to use GRAPH-SEARCH is shown in Figure 14. These are problems where if we keep revisiting repeated states, we observe an exponential explosion in time complexity. For instance, if there are

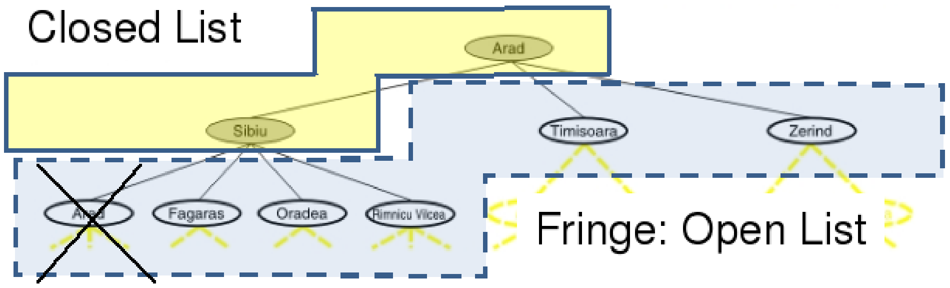

Figure 13: The closed list (already visited nodes) and the fringe.

multiple paths to reach states as in Figure 14-a, then the resulting search tree looks like the one in Figure 14-b, where a node expands unnecessarily to the same node twice.

Grids are used in many problems as a way to discretize continuous spaces. Grid-based search problems, however, as the one shown in Figure 14-c, exhibit the same issue with repeated states. For example, when we generate the neighbors of node A, if we do not avoid repeated states, A will be unnecessarily generated four more times when its children are expanded. This issue renders searching grids impossible if we do not avoid repeated states. This is a very important case, because grids are used in many applications (e.g., computer games, physical systems such as robots) in order to discretize a continuous space.

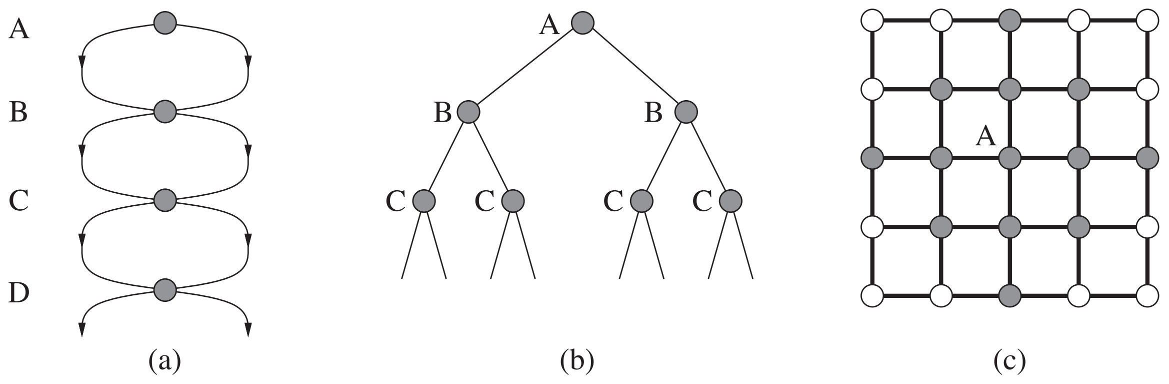

Figure 14: Examples of Exponentially Large Space Trees