# Lecture 2
# Uninformed Search

Alice Gao

January 10, 2022

## Contents

Modified and distributed by Blake VanBerlo, with permission from Alice Gao.

# 1 Learning Goals

By the end of this lecture, you should be able to

- Formulate a real world problem as a search problem.
- Trace the execution of and implement uninformed search algorithms (Breadth-first search, Depth-first search, Iterative deepening search, and Lowest-cost-first search).
- Given an uninformed search algorithm, explain its space complexity, time complexity, and whether it has any guarantees on the quality of the solution found.
- Given a scenario, explain whether and why it is appropriate to use an uninformed algorithm.

# 2 Applications of Search Algorithms

In this section, I am going to describe several applications of search algorithms.

## 2.1 Propositional Satisfiability

Let me talk about a class of problems called propositional satisfiability.

We have a propositional formula. For example,

$$((((a \wedge b) \vee c) \wedge d) \vee (\neg e)).$$

Given this propositional formula, is there a way to assign truth values to the variables to make the formula true? This is an example of a broad class of problems called Satisfiability or SAT problems.

One application of this problem is the FCC spectrum auction. The purpose of the FCC spectrum auction is to buy back radio spectrums from TV broadcasters and sell them to the Telecom companies (who provide services for our phones).

When the FCC is buying back the radio spectrums, they will offer a price to the TV broadcasters. Depending on the price, some TV broadcasters will be willing to give up their spectrums and go off air. Other TV broadcasters may decide to hold on to their spectrums and stay on air.

For the TV broadcasters who decide to remain on air, FCC may need to reassign the spectrums so that the companies do not interfere with one another. This is a packing problem and it can be formulated as a propositional satisfiability problem.

The slide has an extremely small problem and it is easy to solve. For the FCC spectrum auction, solving the packing problems is computationally challenging. First, the formulas are huge and have a lot of variables in them. Second, we need to solve a huge number of such problems. Finally, we need to solve each problem instance very quickly. Perhaps up to a minute per problem. Alice's advisor Kevin Leyton-Brown and his students worked on developing efficient algorithms to solve this problem.

If you are interested, check out the news article and some papers related to this project.

## 2.2 Hua Rong Dao

This is a classic Chinese sliding puzzle called Hua Rong Dao.


In the Romance of the Three Kingdoms, there was a famous battle called the Battle of the Red Cliffs between the Wei kingdom and the Shu kingdom. The Wei kingdom is led by their general called Cao Cao. The Shu kingdom has a famous strategist called Zhuge Liang.

Cao Cao was badly defeated and he fled with a handful of soldiers. The only way for Cao Cao to escape alive was through the narrow Hua Rong Dao. Zhuge Liang anticipated this and placed his best generals at Hua Rong Dao. These generals are Zhang Fei, Ma Chao, Zhao Yun, Huang Zhong and Guan Yu.

Cao Cao was a great warrior and he defeated the first four generals. When he saw Guan Yu, Cao Cao felt a sense of despair. Guan Yu was the best, and Cao Cao knew that he had no hope of defeating Guan Yu. Luckily for Cao Cao, the two knew each other before. Guan Yu was once a guest at Cao Cao's place and Cao Cao showed him considerable kindness. By appealing to their friendship, Cao Cao persuaded Guan Yu to let him escape. This was the story of Hua Rong Dao.

This puzzle shows Cao Cao trying to escape through Hua Rong Dao and the other people trying to stop him. Our goal is to slide the pieces horizontally and vertically, until Cao Cao escapes from the bottom opening. In other words, we want to move Cao Cao to this 2x2 region.

The puzzle has many other initial configurations. You can check out other initial configurations on the Chinese Wikipedia page.

## 2.3 Sliding Puzzles

The first one is called the eight puzzle. In eight puzzles, we have numbers on tiles from 1 to 8 arranged in a 3x3 grid, and the rule is that we can slid the tiles either horizontally or vertically. The goal is to move the tiles from the initial configuration to the final configuration, where the tiles are arranged from 1 to 8 in order.

Initial State

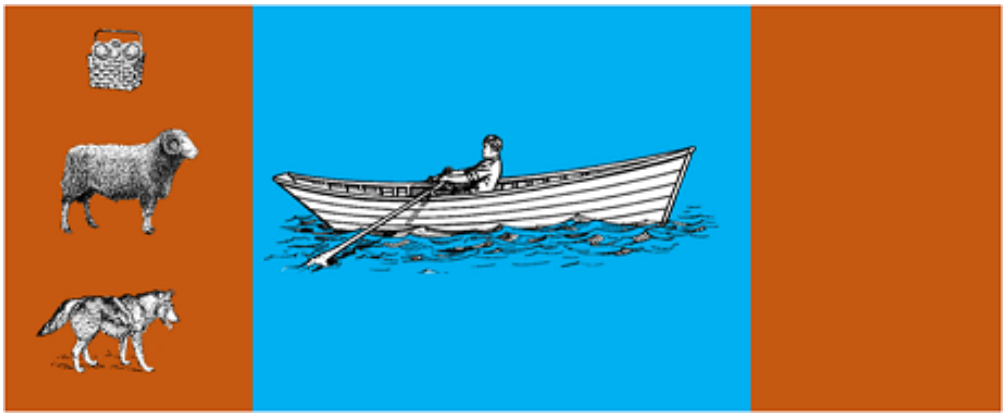
| | | |
|---|---|---|
| 5 | 3 | |
| 8 | 7 | 6 |
| 2 | 4 | 1 |

Goal State

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

## 2.4 River Crossing Problem

This next one is a river crossing problem, called the wolf, goat, and cabbage problem. Here is a description from Wikipedia.



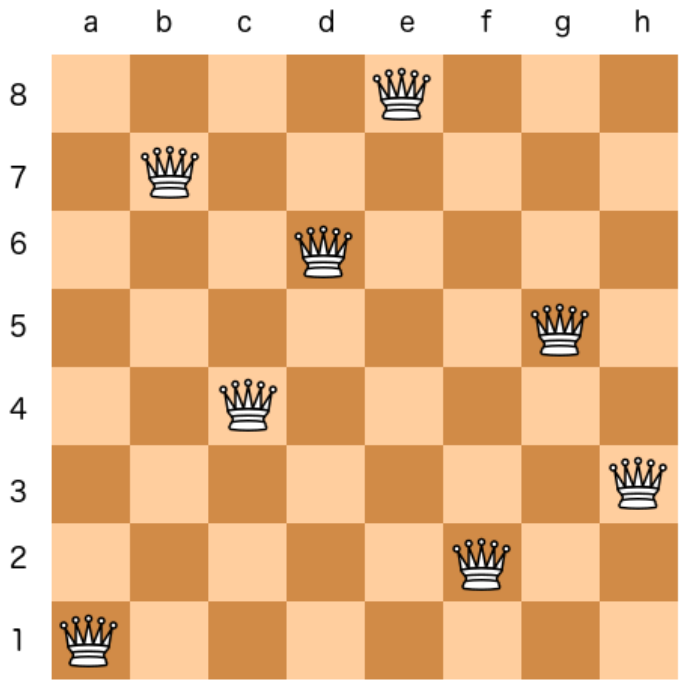
Once upon a time a farmer went to a market and purchased a wolf, a goat, and a cabbage. On their way home, the farmer came to the bank of a river and rented a boat. But crossing the river by boat, the farmer could carry only themselves and a single one of their purchases: the wolf, the goat, or the cabbage.

If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage.

The farmer's challenge was to carry themselves and the purchases to the far bank of the river, leaving each purchase intact. How did the farmer do it?

## 2.5 N-Queens Problem

Finally, let's look at the N queens problem.




We have an 8 by 8 chessboard. We want to put 8 queens on the board such that no two queens attack each other. A queen attacks anything in the same row, in the same column, and in the same diagonal.

This is also a classic constraint satisfaction problem.

# 3 Formulating a Search Problem

In this section, I will discuss how to take a real-world problem and formulate it as a search problem.

## 3.1 Motivation for using search algorithms

When is it a good idea to use search? We often use search algorithms to solve challenging problems. A problem is challenging when it is easy to recognize a solution, but there is no efficient algorithm to derive a solution. These problems are good candidates for applying search algorithms. An example is NP-hard problems.

A search algorithm mimics how we solve a complex problem in real life. For example, to solve a sliding puzzle, we can keep moving the pieces in different ways until we reach the goal state. We are essentially executing a search algorithm in a trial-and-error manner. On a computer, a search algorithm will explore all the paths systematically.

## 3.2 Components of a Search Problem

Let's describe the components of a search problem using this search graph. Each node represents a state. S is the initial state, and G is the goal state.

A search problem may have multiple goal states. The formulation can list all the goal states. Alternatively, we can specify a goal test — a Boolean function that takes a state and returns true if and only if the state is a goal state.

The directed edges represent the successor or neighbour relationships. If there is a directed edge from X to Y, then Y is a successor or neighbour of X. For example, the initial state S has three successors: D, E, and P. You can also consider the directed edges as actions. For instance, taking action allows us to move from S to D.

For some search problems, the directed edges may have costs associated with them. In this course, we assume that the costs are non-negative.

Our goal may be to find any path from the initial state to a goal state. For example, if the directed edges have costs, our goal may be to find the shortest path with the smallest total cost.

## 3.3 Formulating the 8-Puzzle as a Search Problem

**Example:** Let's look at an instance of the 8-puzzle.

A 3x3 grid has the numbers 1 to 8. We want to move the tiles horizontally or vertically until the board is the same as the goal state on the right.

Let's define the 8-puzzle as a search problem.

Initial State

| | | |
|---|---|---|
| 5 | 3 | |
| 8 | 7 | 6 |
| 2 | 4 | 1 |

Goal State

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

**State Definition**

We need to define a state. A state must contain enough information so that:

- We can distinguish two different states before and after moving a piece.
- We can verify whether a state is a goal state or not.
- We can generate the successors of a state efficiently.

For the 8-puzzle, the order of the numbers is essential. Let's use 0 to denote the empty square. Each state has 9 numbers by writing out the numbers line by line from top to bottom. Given this state definition, the initial state is 530-876-241, and the goal state is 123-456-780.

**Successor Function**

Next, let's define the successor function. To change a state, we need to move a piece adjacent to the empty square to the empty square. For the initial state, we can move 3 or 6 to the empty square.

There are many ways to describe the successor function. Here is one possibility. Let's consider the empty square as a tile. If state B is a successor of state A, we can convert A to B by moving the empty tile up, down left, or right by one step.

**Cost function**

What about the cost function?

For the 8-puzzle, our goal is to find a path from the initial state to the goal state. We may want to minimize the number of moves. There is no sense of one move being more expensive than another one. We can define every move or every directed edge to have a cost of one.

Take a look at the complete search formulation for the 8-puzzle.

- State: $x_{00}x_{01}x_{02}, x_{10}x_{11}x_{12}, x_{20}x_{21}x_{22}$

$x_{ij}$ is the number in row $i$ and column $j$. $i, j \in \{0, 1, 2\}$. $x_{ij} \in \{0, \ldots, 8\}$. $x_{ij} = 0$ denotes the empty square.

- Initial state: $530, 876, 241$.
- Goal states: $123, 456, 780$.
- Successor function: Consider the empty square as a tile. State B is a successor of state A if and only if we can convert A to B by moving the empty tile up, down, left, or right by one step.
- Cost function: Each move has a cost of 1.

**Problem:** Which of the following is a successor of 530,876,241?

(A) 350, 876, 241

(B) 536, 870, 241

(C) 537, 806, 241

(D) 538, 076, 241

**Solution:** There are only two possible ways we can move the empty tile. We can move the empty tile to where the 3 was, so we would get 503,876,241. We could also move the empty tile to where the 6 was, so we would get 536,870,241.

Below is the visual representation of the transformation from the initial state to the successor state when we move 6, which is the correct answer:

Initial State

| | | |
|---|---|---|
| 5 | 3 | |
| 8 | 7 | 6 |
| 2 | 4 | 1 |

Successor State

| | | |
|---|---|---|
| 5 | 3 | 6 |
| 8 | 7 | |
| 2 | 4 | 1 |

The correct answer is (B) 536,870,241.

## 3.4 Choosing Among Multiple Formulations

For a given problem, there are often multiple ways of formulating it as a search problem. Choosing a suitable formulation is crucial since it may affect how efficiently we can find a solution.

We can construct the search graph by creating the nodes using the state definition and creating the directed edges using the successor function.

Ideally, we want to choose a state definition that minimizes the number of nodes in the graph, potentially reducing the number of nodes we need to explore in the worst case. We also want to choose the successor function such that the graph has fewer edges if possible.

Moreover, there are interactions between the state definition and the successor function. Choosing one state definition may make it easier or harder to implement the successor function. Consider the 8-puzzle again. An alternative way of defining a state is to keep track of each tile's x and y coordinates. We will have eight coordinates, one for each of the eight tiles. Given this state definition, how would you implement the successor function? Which state definition makes it easier to implement the successor function? I will leave this as a thought question for you.

# 4 Generic Search Algorithm

## 4.1 The Search Graph

Given the formulation of a search problem, we can generate the search graph.

The nodes in the graph are the states. Each state appears once in the search graph. There are no duplicates. There is a directed edge from state A to B if and only if B is a successor of A. If there is a cost function, we can label each directed edge with its cost.

When solving a search problem, we often do not work with the search graph directly. For many real-world search problems, the search graph may be extremely large or infinite. It may be impossible or infeasible to generate and store the complete search graph in memory. Instead we will search for a solution in the search graph by generating a search tree.

## 4.2 The Search Tree

Given a search graph, an initial state, and a goal test, we will search for a solution by generating a search tree.

The following is an example of a search tree:

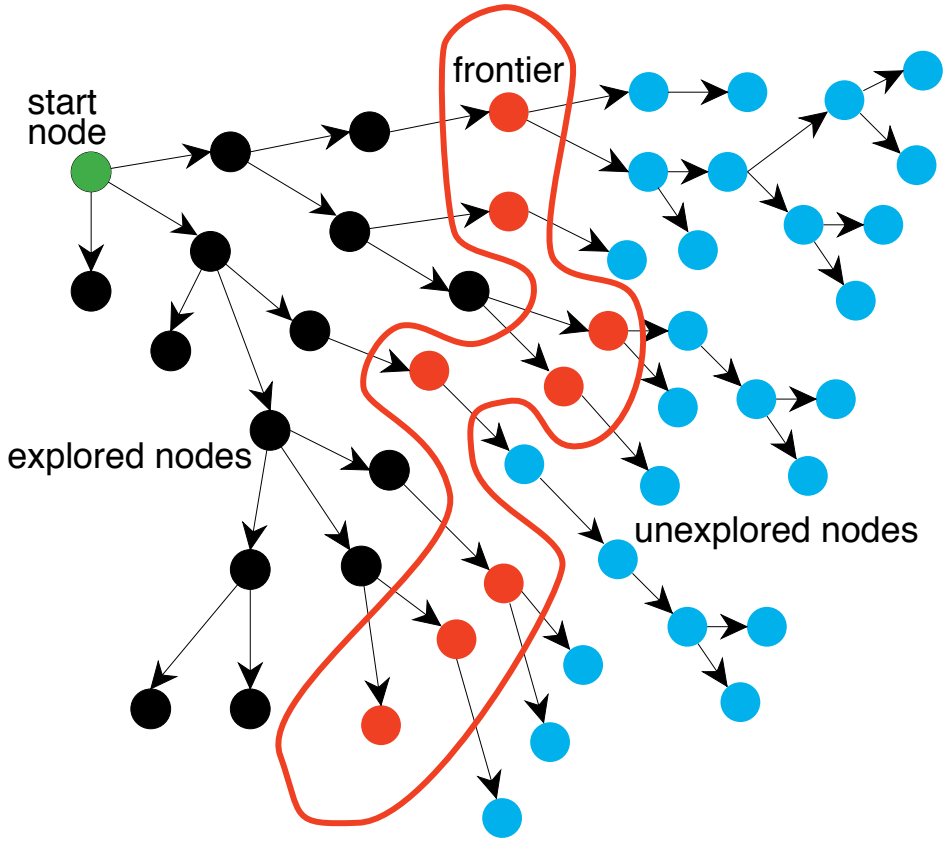



Figure 1: A Generic Search Tree

We do this by maintaining a frontier of paths from the start node. The frontier contains all of the paths that are available for exploration. At the beginning, the frontier starts with having the trivial path containing the initial state and no edges. As the search proceeds, the frontier expands into the unexplored region until a goal node is encountered.

At each step, we select one path from the frontier, check whether it is a goal state. If it is not a goal state, we will expand it. Expanding the state means applying the successor function and generating all of its successors. We will add its successors to the frontier and the process repeats.

It is important to distinguish the search graph and the search tree. Here are a few points: We can construct the search graph given a problem formulation, whereas constructing the search tree requires executing an algorithm. Each state appears once in the search graph but can appear multiple times in the search tree. The search graph is static, whereas the search tree grows in one direction. Even if the search graph is finite, a corresponding search tree may be infinite.

## 4.3 Generic Search Algorithm

Let's look at the pseudocode for the generic search algorithm.

**Algorithm 1** A Generic Search Algorithm

```
1: procedure SEARCH(Graph, Start node s, Goal test goal(n))
2:     frontier := {⟨s⟩};
3:     while frontier is not empty do
4:         select and remove path ⟨n_0, ..., n_k⟩ from frontier;
5:         if goal(n_k) then
6:             return ⟨n_0, ..., n_k⟩;
7:         for every neighbour n of n_k do
8:             add ⟨n_0, ..., n_k, n⟩ to frontier;
9:     return no solution
```

Initially, the frontier contains a path consisting of the initial state only. We will continue exploring the space as long as the frontier is not empty.

At each step, we remove a path from the frontier. If the last node $n_k$ on the path is a goal, we can return the path as a solution. Otherwise, we expand the path. That is, find the neighbours of the last node $n_k$. For every neighbour, we construct a path with one more edge than before, and add the path to the frontier.

Let's make a note of two lines in the algorithm.

First, on line 4, we need to select a path and remove it from the frontier. How we select this path from the frontier determines our search strategy. For instance, depth-first search selects the newest path added to the frontier, whereas breadth-first search selects the oldest path added to the frontier.

Second, on line 5, we perform the goal test when the path is removed from the frontier, not when the path is added to the frontier. There are two reasons for doing this. First, the goal test could be costly to perform, so we should delay this computation in case it is not necessary. Second, the first path to a goal node added to the frontier may not be the path with the least cost. If we want to find the shortest path, we should wait for all possible paths to be added to the frontier before selecting the best among them.

# 5 Uninformed Search Algorithms

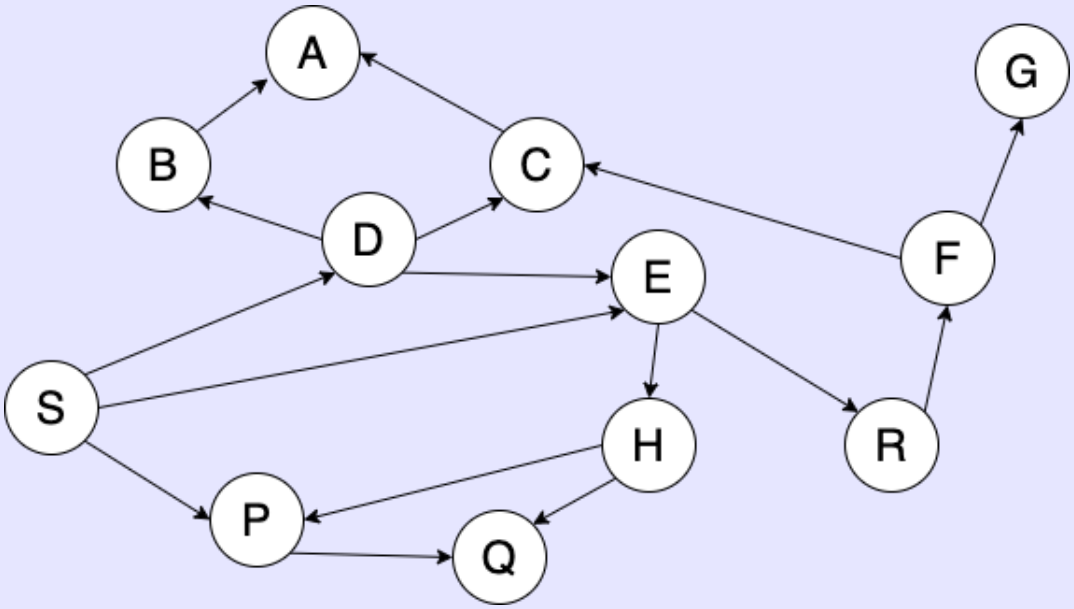
## 5.1 Depth-First Search

Recall the generic graph search algorithm. Our strategy for selecting and removing a path from the frontier determines the search algorithm.

Depth-first search treats the frontier as a stack. A stack is last-in, first-out. Think of a stack of plates. We add and remove items at the same end of the stack. Since the frontier is a stack, we choose to remove the most recent path added to the frontier at every step.

### 5.1.1 Tracing DFS on a Search Graph

**Example:** Let's trace DFS on the search graph below.




We will add nodes to the frontier in alphabetical order. This order is important. The order in which we remove the nodes from the frontier depends on how we add nodes to the frontier. While tracing the algorithm, you should draw the search tree, keep track of the nodes added to and removed from the frontier, and label the nodes in the order of expansion.

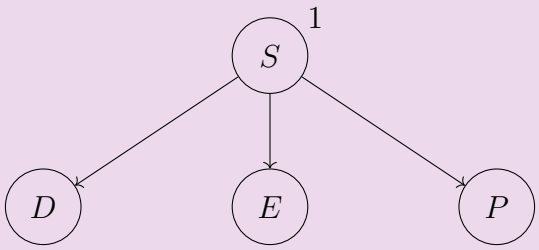
**Solution:**

Let's add the initial state S to the frontier and to the search tree.

frontier: S

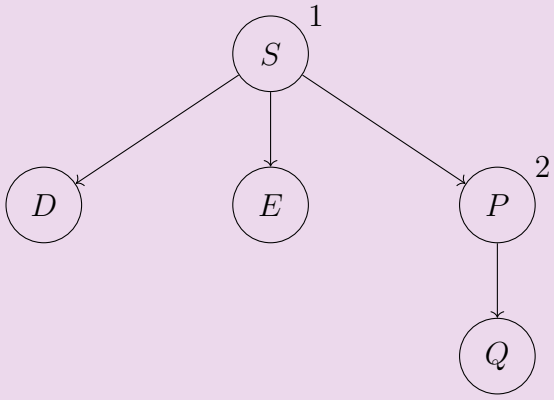

Step 1: The most recent node added to the frontier was S. S is the first node expanded. Remove S from the frontier. S is not a goal. Let's expand S. S has three successors: D, E and P. Add them to the frontier in alphabetical order. Also add them to the search tree.

frontier: ~~S~~ D E P



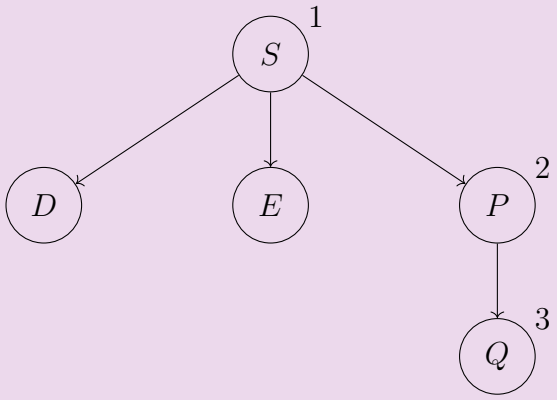

Step 2: Expand P. Remove P from the frontier and add its successors in alphabetical order (Q).

frontier: ~~S~~ D E ~~P~~ Q



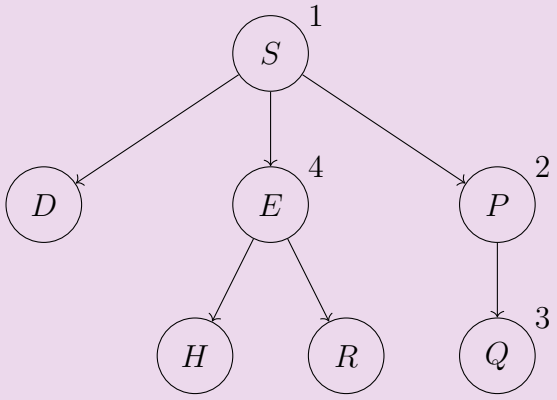

Step 3: The most recent node added to the frontier was Q. Q is the third node expanded. Remove Q from the frontier. Q is not a goal. Let's expand Q. Q has no successor. There's nothing to add to the frontier and to the search tree.

frontier: ~~S~~ D E ~~P~~ ~~Q~~



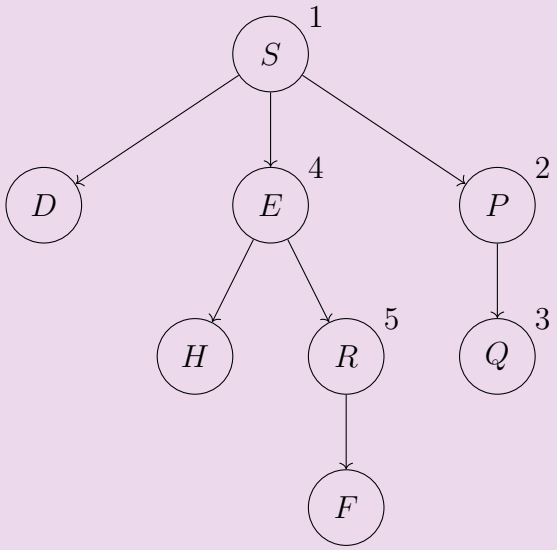

Step 4: The most recent node added to the frontier was E. E is the fourth node expanded. Remove E from the frontier. E is not a goal. Let's expand E. E has two successors: H and R. Add them to the frontier in alphabetical order and add them to the search tree.

frontier: ~~S~~ D ~~E~~ ~~P~~ ~~Q~~ H R



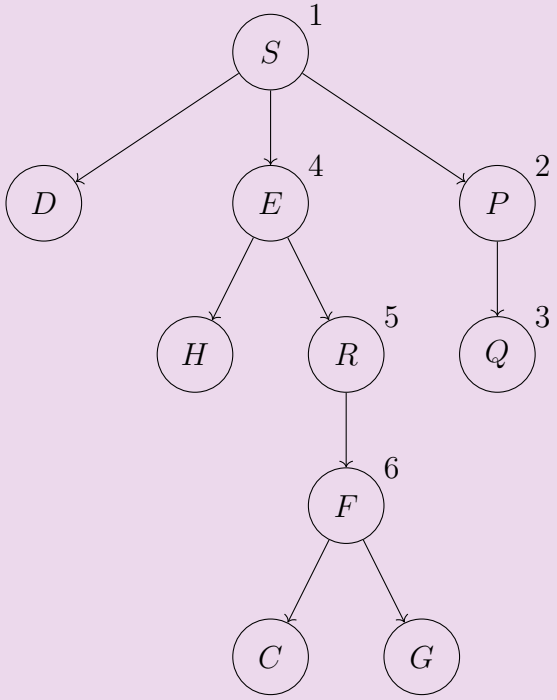

Step 5: The most recent node added to the frontier was R. R is the fifth node expanded. Remove R from the frontier. R is not a goal. Let's expand R. R has one successor F. Add F to the frontier and to the search tree.

frontier: ~~S~~ D ~~E~~ ~~P~~ ~~Q~~ H ~~R~~ F




Step 6: The most recent node added to the frontier was F. F is the sixth node expanded. Remove F from the frontier. F is not a goal. Let's expand F. F has two successors: C and G. Add them to the frontier and to the search tree.

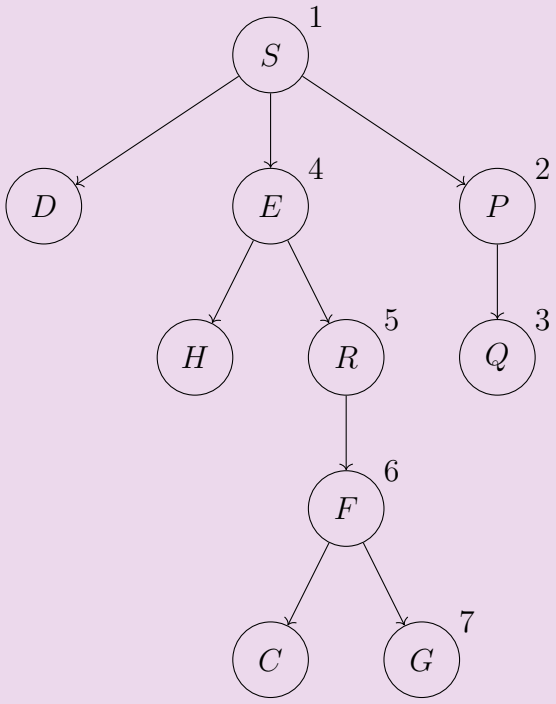
frontier: ~~S~~ D ~~E~~ ~~P~~ ~~Q~~ H ~~R~~ ~~F~~ C G




Step 7: The most recent node added to the frontier was G. G is the seventh node expanded. Remove G from the frontier. G is a goal. Return the path S, E, R, F, G

as a solution.

frontier: ~~S~~ D ~~E~~ ~~P~~ ~~Q~~ H ~~R~~ ~~F~~ C ~~G~~




Let me make some observations.

Note that DFS goes down one path until completion. If it doesn't find a goal on the path, it will backtrack and go down the next path.

The order of adding nodes to the frontier is important. Because we added nodes to the frontier in alphabetical order, we ended up removing them in reverse alphabetical order. That's why that DFS started by exploring the rightmost path in the search tree.

If we added nodes to the frontier in reverse alphabetical order, then DFS would start by exploring the leftmost path in the search tree. I encourage you to try tracing DFS for this case.

### 5.1.2 Properties of DFS

Let me discuss the properties of depth-first search. I will discuss space complexity, time complexity, completeness, and optimality.

**Useful quantities**

I will use the three quantities to characterize the complexities.

$b$ is the branching factor or the maximum number of successors or neighbours that any node has. Intuitively, $b$ determines how fat or how wide the search tree is.

$m$ is the maximum depth of the search tree, and $d$ is the depth of the shallowest goal node.

$m$ and $d$ are typically not the same. Let this triangle represent the search tree. The big G's represent goal nodes in different locations. For this tree, $d$ is the depth of the shallowest G in the search tree, whereas $m$ is the depth of the entire search tree. $m$ is always greater than or equal to $d$. For some search trees, $m$ and $d$ are the same.

**Space complexity**

Let's look at space complexity. What is the size of the frontier for DFS in the worst case?

Before I discuss the space complexity, why is it sufficient to store the frontier in memory only instead of storing the search graph or the search tree? When executing a search algorithm, we never need to store the search graph or the search tree. For most problems, the search graph and search tree are too large to be stored in memory. Instead, it is sufficient to store the frontier only to execute the algorithm.

Let's come back to space complexity. What is the size of the frontier in the worst case? DFS explores one path at a time, so it needs to remember the current path it is exploring. In the worst case, the current path has $m$ nodes, where $m$ is the maximum depth of the search tree. Suppose that this line in the triangle represents the current path we are exploring.

If the current path does not lead to a goal, DFS must backtrack and try another path. To do this, DFS needs to remember the alternative nodes at every level. These alternative nodes are the siblings of the nodes on the current path. How many siblings does each node have? The maximum number of siblings is $b$, the branching factor.

In summary, DFS needs to remember at most m nodes on the current path. Furthermore, for each of the $m$ nodes, DFS needs to remember at most $b$ siblings. Therefore, the space complexity is $O(m * b)$. Note that the space complexity is linear in $m$, the maximum depth of the search tree.

**Time complexity**

Let's think about time complexity. How many nodes does DFS need to visit in the worst case?

In the worst case, DFS will not find a goal node until it has explored the entire search tree. How many nodes are there in the search tree? The tree has m levels. As we move down in the tree, the number of nodes on each level increases exponentially. Level 0 has 1 node, level 1 has at most $b$ nodes, level 2 has at most $b^2$ nodes, and the bottom level has at most $b^m$ nodes. The size of the bottom level is the dominating term in the summation. Thus, the time complexity is $O(b^m)$. The time complexity is exponential in $m$, the maximum depth of the search tree.

**Interpreting the Complexities**

How should we interpret these complexities? Are they good or bad? Are they surprising or

expected? Our search problem is challenging. The search tree has an exponential number of nodes, and an algorithm may have to visit all of them in the worst case. Given this, having an exponential time complexity is expected. Most search algorithms will have exponential space and time complexities. However, DFS stands out by having a linear space complexity.

**Completeness**

Let's discuss completeness. A search algorithm is complete if it is guaranteed to find a solution if a solution exists.

Unfortunately, DFS is not complete. For example, if the search tree has an infinite path, then DFS will get stuck in the path and will not terminate.

For example, if a search graph has a cycle, the search tree will have an infinite path. Alternatively, the search tree may have an infinite path, not because of a cycle. Try to find an example where a search graph has no cycles but is infinite.

**Optimality**

Finally, is DFS optimal? For example, if DFS terminates on a problem, is it guaranteed to return the path with the least total cost?

Unfortunately, the answer is no. DFS is oblivious of the costs. Even if DFS finds a solution, it has no guarantee on the quality of the solution found.

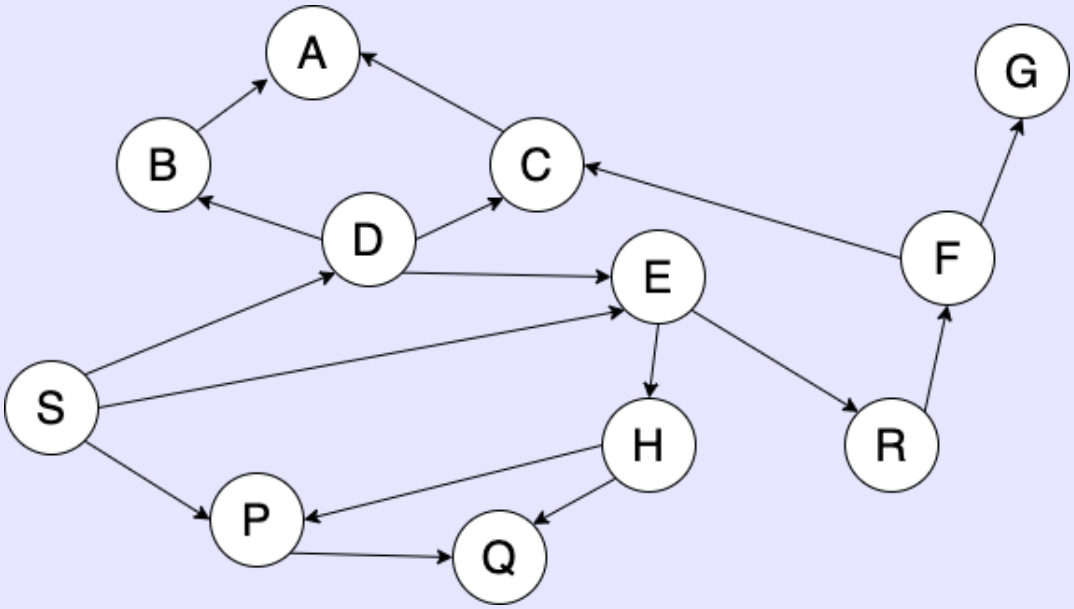
## 5.2 Breadth-First Search

Recall the generic search algorithm. Our strategy for selecting and removing a path from the frontier determines the search algorithm.

Breadth-First Search treats the frontier as a queue. A queue is first in first out. Imagine a line-up at the ticket office. We add items to the queue on one end and remove items at the other end. If the frontier is a queue, we remove the oldest node added to the frontier at each step.

### 5.2.1 Tracing BFS on a Search Graph

**Example:** Let's trace BFS on the search graph below. We will add nodes to the frontier in alphabetical order.




**Solution:**

Let's add the initial state S to the frontier and to the search tree.

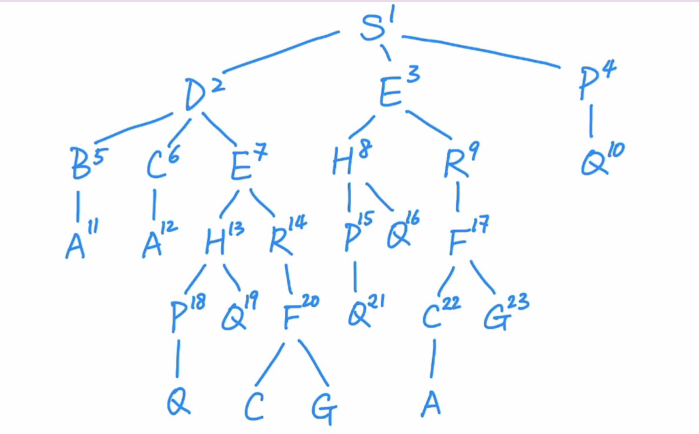
Step 1: The oldest node added to the frontier was S. S is the first node expanded. Remove S from the frontier. S is not a goal. Let's expand S. S has three successors: D, E and P. Add them to the frontier in alphabetical order. Also add them to the search tree.

Step 2: The oldest node added to the frontier was D. D is the second node expanded. Remove D from the frontier. D is not a goal. Let's expand D. D has three successors: B, C, E. Add them to the frontier in alphabetical order and add them to the search tree.

Step 3: The oldest node added to the frontier was E. E is the third node expanded. Remove E from the frontier. E is not a goal. Let's expand E. E has two successors H and R. Add them to the frontier in alphabetical order and add them to the search tree.

Step 4: The oldest node added to the frontier was P. P is the fourth node expanded. Remove P from the frontier. P is not a goal. Let's expand P. P has one successor Q. Add Q to the frontier and to the search tree.

So far, BFS has explored the first two levels and has not found a goal yet. I will skip the rest of the steps and show you the final search tree.




Let me make some observations.

BFS explores the search tree level by level until it finds a goal.

The order of adding nodes to the frontier determines the order of exploring the nodes in each level. Since we added nodes to the frontier in alphabetical order, BFS explores each level in alphabetical order as well.

As a practice question, try tracing BFS on this search graph by adding nodes to the frontier in reverse alphabetical order.

BFS explores the search tree level by level. When BFS terminates, we found the shallowest goal node in the search tree. In other words, the path we found has the fewest edges to any goal node among all possible solutions.

### 5.2.2 Properties of BFS

Let me discuss the properties of BFS: space complexity, time complexity, completeness, and optimality.

When characterizing complexities, I'll make use of the three useful quantities, b is the branching factor, m is the maximum depth of the search tree, and d is the depth of the shallowest

goal node.

**Space Complexity**

Let's look at space complexity. What is the size of the frontier for BFS in the worst case?

Since BFS explores the tree level by level, it will terminate at depth $d$, the depth of the shallowest goal node. The size of the frontier is at most the size of level $d$. How many nodes are there at depth d? At level 0, we have one node. At level 1, we have at most $b$ nodes. At level 2, there are at most $b^2$ nodes. At level d, we have at most $b^d$ nodes. So, the space complexity is $O(b^d)$. This is exponential in $d$.

**Time Complexity**

What about time complexity? How many nodes does BFS need to visit in the worst case?

In the worst case, BFS will visit all the nodes up to and including depth d and terminates at depth d. The time complexity is dominated by the size of level d. The total number of nodes at level d is $b^d$. So, the time complexity is $O(b^d)$. This is exponential in $d$.

So far, the complexities of BFS are not great. Both space and time complexity are exponential. However, these complexities are not surprising, since the total number of nodes in the search tree is exponential.

**Completeness**

What about completeness? Is BFS guaranteed to find a solution if one exists?

Unlike DFS, BFS won't follow an infinite path. BFS explores the nodes level by level and will terminate when it finds the shallowest goal node.

If a solution exists at a finite depth and if the branching factor is finite, then BFS is guaranteed to find the solution.

**Optimality**

Finally, let's look at optimality. If BFS terminates on a problem, is it guaranteed to return the optimal solution?

Unfortunately, the answer is no. BFS pays no attention to the edge costs. Therefore, it doesn't have any guarantee on the total cost of the path returned.

However, we can achieve a slightly weaker property. BFS is guaranteed to find the shallowest goal node since it explores the search tree level by level. In the special case when every edge has the same cost, the solution returned by BFS is the optimal solution.

## 5.3 Iterative Deepening Search

Let's look at a comparison between BFS and DFS. There is a clear trade-off between them. On one hand, DFS has better space complexity. DFS requires linear space whereas BFS requires exponential space. On the other hand, BFS has a better guarantee on completeness. BFS is guaranteed to find a solution whereas DFS may get stuck in a cycle.

| **BFS** | **DFS** |
|---|---|
| $O(b^d)$ exponential space | $O(bm)$ linear space |
| Guaranteed to find a solution if one exists | May get stuck on infinite paths |

I have a question for you. Do you think it is possible to design a search algorithm that combines the best of BFS and DFS? This may sound too good to be true, but it turns out that it is possible.

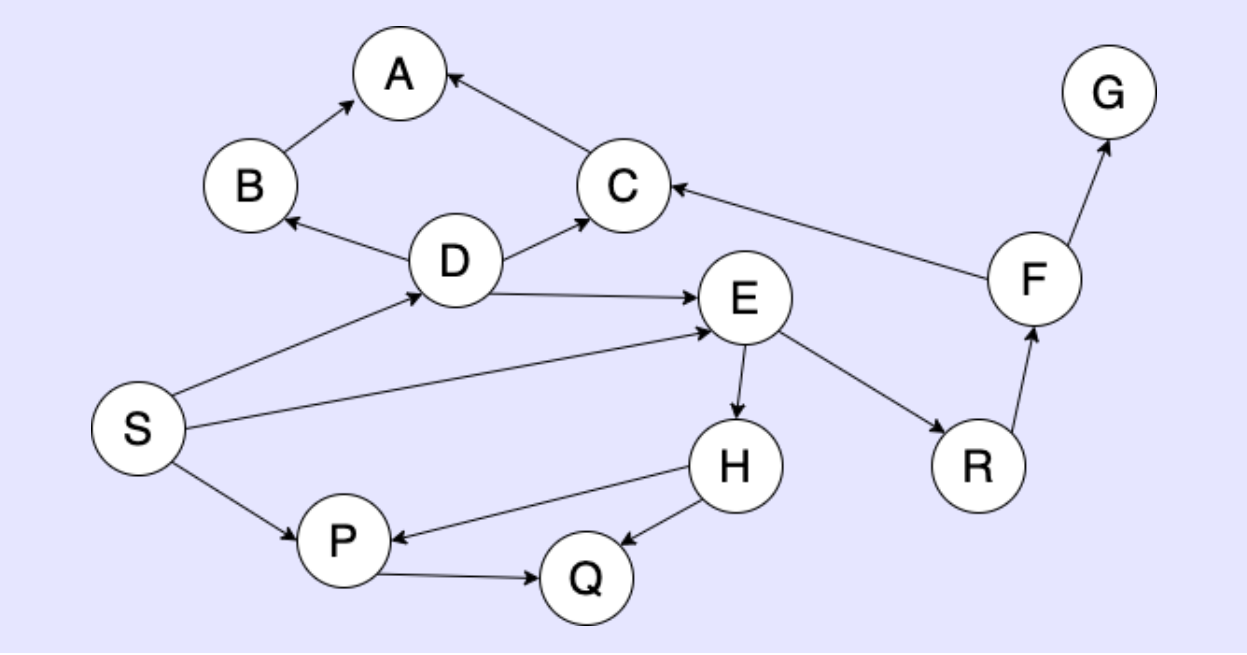
Iterative-Deepening Search combines BFS and DFS in an interesting way. IDS starts with a depth limit of 0 and increment it as long as it hasn't found a goal node. For every depth limit, IDS performs DFS up to the depth limit. This means that, if the current node is at the depth limit, IDS will not generate and add its successors to the frontier — essentially, IDS backtracks when it reaches the depth limit. Every time we increase the depth limit, IDS starts the depth-first search all over again.

IDS is an interesting hybrid of BFS and DFS. It is similar to DFS since it performs DFS for each depth limit. IDS is similar to BFS since it explores the search graph level by level by increasing the depth limit by one every time.

### 5.3.1 Tracing IDS on a Search Graph

**Example:** Let's trace IDS on the search graph below.

I will add nodes to the frontier in alphabetical order. While executing the algorithm, I will keep track of the depth limit, the frontier, and the search tree.




**Solution:**

Start with a depth limit of 0. Add the initial state S to the frontier. The most recent node added to the frontier was S. S is the first node expanded. Remove S from the frontier. S is at level 0, which is the depth limit. Therefore, we do not generate its successors.

The frontier is empty. Let's increase the depth limit to 1.

Add the initial state S to the frontier.

The most recent node added to the frontier was S. S is the second node expanded. Remove S from the frontier. S is not at the depth limit. S has three successors: D, E, P. Add them to the frontier and add them to the search tree.

The most recent node added to the frontier was P. P is the third node expanded. Remove P from the frontier. P is at the depth limit. We do not generate P's successors.

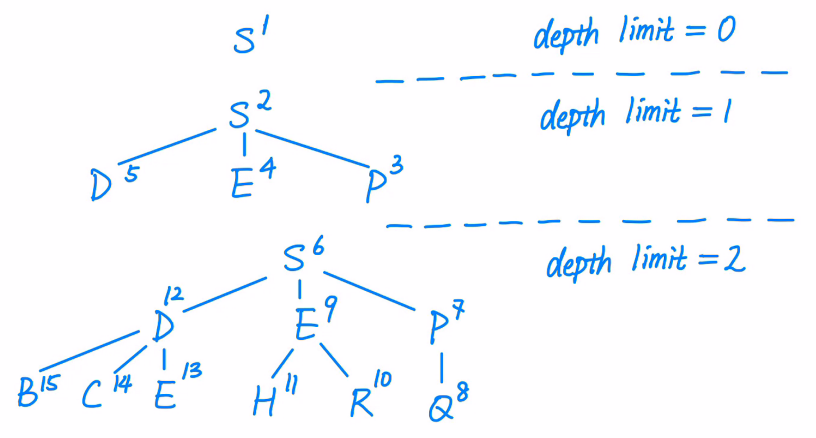
The most recent node added to the frontier was E. E is the fourth node expanded. Remove E from the frontier. E is at the depth limit. We do not generate E's successors.

The most recent node added to the frontier was D. D is the fifth node expanded. Remove D from the frontier. D is at the depth limit. We do not generate D's successors.

The frontier is empty. Let's increase the depth limit to 2.

By now, you should have a sense of how IDS works. Let me show you the search tree after we are done with depth limit = 2. IDS still has not terminated. We need to

increase the depth limit and perform DFS again.




As you can see, IDS is mimicking the behaviour of BFS in the sense that it is increasing the depth limit by 1 each time. Doing this ensures that IDS does not miss the shallowest goal node. For each depth limit, IDS tries to save space by performing DFS.

### 5.3.2 Properties of IDS

Let's look at the properties of IDS: space complexity, time complexity, completeness, and optimality.

**Space Complexity**

For space complexity, IDS performs DFS for every depth limit. Since IDS increases the depth limit by 1 each time, it will terminate at depth $d$, the depth of the shallowest goal node. The maximum length of the current path is $d$ and each node on the path has at most $b$ siblings.

Thus, the space complexity is $O(bd)$. This is linear in $d$, the depth of the shallowest goal node. The space complexity is similar to DFS.

**Time Complexity**

What about time complexity?

In the worst case, IDS will visit all the nodes in the top d levels. Thus, IDS's time complexity is similar to that of BFS. The number of nodes up to depth d is dominated by the number of nodes at depth $d$, which is $b^d$.

Thus, the time complexity is $O(b^d)$. This is exponential in $d$, the depth of the shallowest goal node.

The asymptotic time complexity of IDS is the same as that of BFS. However, if we calculate the exact number of nodes visited, the number for IDS is larger than that of BFS. The reason is that IDS performs some repeated computation. Every time IDS increases the depth limit,

it performs DFS from scratch. This means that, when IDS terminates at depth d, it visited level d 1 time, level (d-1) 2 times, level (d-2) 3 times, and so on.

The total number of nodes visited by BFS is roughly $b^d * \frac{b}{b-1}$, and the total number of nodes visited by IDS is approximately $b^d * (\frac{b}{b-1})^2$. The difference is a multiplicative factor of $\frac{b}{b-1}$. This factor is largest when $b$ is small. As $b$ becomes larger, the factor becomes close to 1.

**Completeness**

Let's look at completeness. Is IDS guaranteed to find the solution if one exists?

The answer is yes. Since IDS only performs DFS until a depth limit, it won't follow an infinite path forever. Thus, it doesn't have the same problem as DFS. IDS is complete and is guaranteed to find a solution if one exists.

**Optimality**

Finally, let's look at optimality.

Similar to other uninformed search algorithms, IDS does not pay any attention to the edge costs. Therefore, it makes no guarantee on the quality of the solution found.

However, since IDS increases the depth limit by one each time, it can achieve the same weaker property as BFS. IDS is guaranteed to find the shallowest goal node. If every edge has the same cost, IDS is guaranteed to return the optimal solution.

**Recap of IDS Properties**

Here is a recap of the IDS properties.

IDS is an amazing example of a win-win situation.

Similar to DFS, IDS requires linear space only.

Similar to BFS, IDS is complete and is guaranteed to find the shallowest goal node.

IDS also has the same time complexity as BFS, although the exact number of nodes visited by IDS is larger than that of BFS.

# 6 Practice Problems

Feel free to discuss these questions. The answers will not be provided.

1. What is the difference between the search graph and the search tree?
2. For the search algorithms that we consider, we only store the frontier, not the search graph nor the search tree. If we are able to store the search graph or the search tree, would our search algorithm be better? Can we avoid certain problems for storing the search graph or the search tree?
3. For IDS, can we increase the depth limit by more than 1 each time?
4. For IDS, whenever we increase the depth limit, we can re-use the results of the previous depth-first search instead of performing the depth-first search from scratch?
5. I learned BFS and DFS in CS 341. However, the complexity analyses of these algorithms in this course look quite different from those in CS 341. Why is this the case?
6. How do we derive the goal path from the expansion? We know we reached the final node but do we also keep track of how we got there?