

1.00  
1.00

# Graph Search

## Note

All questions on this page are due at 10:00pm on Sunday, 02 March.

## Table of Contents

- [1\) Introduction](#)
- [2\) What is a Graph?](#)
  - [2.1\) Uses](#)
  - [2.2\) Another Example: The 15 Puzzle](#)
- [3\) Pathfinding in the Abstract](#)
  - [3.1\) Order Matters!](#)
    - [3.1.1\) Example One: Replace First Path On Agenda With Its Children](#)
    - [3.1.2\) Example Two: Replace Last Path On Agenda With Its Children](#)
    - [3.1.3\) Example Three: Remove First Path On Agenda, Add Its Children to End](#)
    - [3.1.4\) Order Matters: Summary](#)
  - [3.2\) Another Example](#)
  - [3.3\) Check Yourself: USA](#)
  - [3.4\) Check Yourself: Back to Flood Fill](#)
- [4\) Summary of BFS vs. DFS](#)
  - [4.1\) So Is DFS Bad?](#)
- [5\) Pathfinding in Python](#)
- [6\) A Small Example](#)
- [7\) Revisiting the Representation of a Graph: 15-Puzzle](#)
- [8\) Another Example: Word Ladders](#)
- [9\) Summary](#)

## 1) Introduction

In last week's reading, we looked at two related problems: *flood fill* (an operation on images for recoloring a contiguous region of similarly colored cells) and *pathfinding* (finding and returning a path through a maze, which we arrived at by modifying flood fill to keep track of additional information). In this reading, we'll be building on some of those ideas and formalizing them a little bit, looking at an interesting category of algorithms called *graph search* algorithms, which can be used to solve a wide variety of different problems.

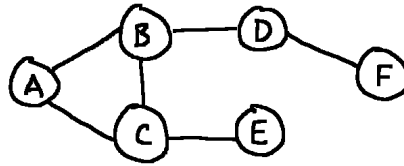
This reading will start at a high level, introducing and discussing graph search in the abstract; but, by the end, we will be talking in detail about code for graph search and about some of the work involved in bringing these approaches to bear on new problems.

## 2) What is a Graph?

Graph search algorithms are all about systematically exploring structures called *graphs*; but, before we can talk too much about exploring graphs, we will need to define what a graph is. Graphs are structures that represent sets of objects and relationships between pairs of those objects. In 6.101, we'll talk about graphs as consisting of two things in an abstract sense:

- a set of *vertices*, one for each object we're interested in; and
- a set of *edges*, which represent relationships between those objects.

We will often represent graphs using pictures, like so:



In this kind of drawing, the circles represent vertices in our graph, and each corresponds to an object (with which we label each vertex). Here, we have six objects (*A*, *B*, *C*, *D*, *E*, and *F*), each of which is represented by a single vertex.

The lines that connect the circles in the drawing represent edges, each of which corresponds to a relationship between a pair of objects. For example, *B* and *C* are related in some way according to this graph (since there is an edge connecting them), but *C* and *D* are not related in that way. In this drawing, each line represents a *bidirectional* edge: a symmetric relationship between vertices; if the relationship only goes one way, we will draw it as an arrow (a *directed edge* or *unidirectional edge*) instead.

That's still quite abstract for now, but that's our notion of a graph. Our focus in this reading will be on a specific category of algorithms called *graph search*, which involve systematically "exploring" a graph in some way, such that we can answer questions about it. There are a number of questions that we might be interested to ask about a graph, for example:

- What are all of the objects that are directly connected to *A*?
- What are all of the objects that I can reach by traversing exactly two connections?
- What if I am willing to traverse an arbitrary number of connections?

All of these questions can be solved using graph search approaches, but our focus in 6.101 is going to be on a particular subcategory of graph search algorithms, *pathfinding algorithms*, which we will use to answer the question: given a starting object and a goal object, what is a path that connects the two?

If you recall last week's reading, this idea is very similar to the last problem we tried to solve, where we were trying to find a path from one location in a maze to another. To think about a maze as a graph, we treat each location in the maze as a vertex in the graph, with an edge between adjacent locations whenever there is no wall blocking the way. So rather than focusing on images in particular, we're generalizing this notion to finding a path in a graph: a sequence of vertices (or a sequence of edges) leading from our starting vertex to our goal vertex.

## 2.1) Uses

So far, this might all seem very abstract. But the idea of a graph, and the algorithms we're going to talk about today, are not just interesting made-up things to think about (although they *are* kind of interesting in their own right); these ideas have tremendous practical utility in a broad variety of contexts. Lots of real-world domains can be thought of in terms of objects and their relationships to each other; this allows us to construct graphs corresponding to these domains, and graph search approaches allow us to answer questions about those contexts. So, it turns out that a wide variety of problems can be solved by: constructing a graph related to that problem, using some sort of graph search approach to answer some questions about that graph, and then reinterpreting those answers in terms of the original problem that we started with. There are a large number of examples, but to name a few:

- **Web search** involves relating web pages to each other and ranking them relative to each other. One of the early approaches to ranking search results, [PageRank](#), ranks pages using a graph whose vertices represent individual web pages and whose edges represent hyperlinks between them.
- **Robot navigation** is about planning actions that a robot can take to affect the world in some way (e.g., move to a different location, bake a cake, deliver a package, etc.). Determining what actions the robot should take to accomplish its goal often involves exploring a graph where vertices represent possible states of the world, and edges reflect actions that the robot can take.
- **VLSI circuit layout** involves arranging millions of transistors on a chip and connecting those components with wires, subject to constraints on the physical locations of the various components (which may affect each other), the total area of the circuit, the total length of wire, or other constraints. This is a complicated process, but it is often solved using graph search techniques on multiple graphs related to the components and the connections between them.
- **Social-network graphs** represent connections between different people (or other entities), based on relationships or past communications. Exploring these graphs can answer complicated questions about these relationships, such as finding groups of people connected by common interests for purposes of targeted advertising, discovering new relationships based on other connections, and more.
- **Route planners** like the GPS-powered systems in most modern cars find optimal paths between physical locations using common modes of transport. These systems often work using graphs of location data (locations connected by ways to move between them) and finding paths between locations that are optimal in some sense (e.g., minimizing travel time, optionally accounting for tolls and traffic, etc.).
- Many kinds of **puzzles and games** can be solved using graph search techniques, often by constructing a graph where each vertex represents the state of the game at a certain point and the edges represent moves that players can make. By finding paths through those graphs, we can often figure out the optimal (or nearly optimal) move to make based on the current state of the game. These techniques are often used when a game is played against artificial intelligence or when games otherwise contain agents that are not controlled by human players.

This is just a small subset of the kinds of problems that can be solved using graph search, but hopefully it gives a sense of the power of some of these approaches. It's also worth tempering expectations a little bit; many of the specific problems mentioned here involve humongous graphs and require more advanced techniques than we're going to be able to cover in 6.101 (we'll focus our attention on a small subset of graph search algorithms). That said, the techniques will serve as an introduction to graph search techniques and as a foundation on which you can build later on if you're interested (particularly in classes like 6.121<sup>[6.006]</sup> Introduction to Algorithms and 6.122<sup>[6.046]</sup> Design and Analysis of Algorithms).

## 2.2) Another Example: The 15 Puzzle

Let's take a look at a concrete example of a puzzle that can be solved using graph search: the [15 puzzle](#). The 15 puzzle consists of a board split into a 4-by-4 grid, with each cell except one occupied by a numbered tile. The goal is to get the board from an initial scrambled state (example shown on the left below) into an ordered state (shown on the right below) by repeatedly sliding tiles into the open position. From each board position, we have between two and four options for moves; for example, considering the board on the left, we have three options for how to proceed: we could move the "5" tile down into the open space, the "12" tile to the right into the open space, or the "7" tile up into the open space.

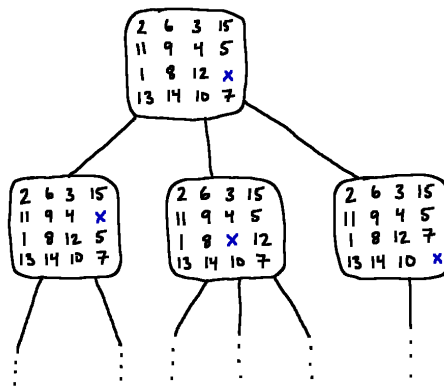
Start (Scrambled)

2	6	3	15
11	9	4	5
1	8	12	
13	14	10	7

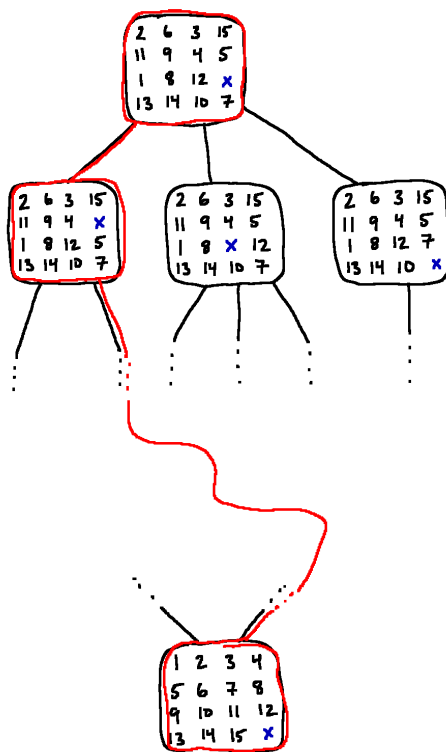
Goal (Ordered)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

The reason to mention this puzzle, aside from it just being kind of a fun puzzle, is that it is one example of a kind of problem we will be able to solve by finding a path through a related graph. Consider creating a graph where each vertex represents the state of the board, and these vertices are connected by edges representing moves that we can take to move between the states. Graphically, we might draw such a graph like so (here only showing a small portion of the graph):



Notice that each vertex here corresponds to one possible state that the game board could be in (an arrangement of the tiles) and that the edges connecting the vertices represent moves that we could make (tiles that we could slide) to move between the states. Now, we can imagine that, if we were to draw out the full graph, *somewhere* in that graph would be a state corresponding to our solved puzzle, and a path (a sequence of game states) starting from our original configuration and ending with the goal configuration would represent a solution to the puzzle and tell us what moves we would need to make in order to solve it:



Before we take a look at some results, let's think a little bit about how big of a problem this is. How many possible distinct arrangements of the tiles are there on the board?

- ☐ 15
- ☐ 16
- ☐  $15^2 = 225$
- ☐  $16^2 = 256$
- ☐  $15^4 = 50,625$
- ☐  $16^4 = 65,536$
- ☐  $15! = 1,307,674,368,000$  (around 1.3 trillion)
- ☐  $16! = 20,922,789,888,000$  (around 21 trillion)

Submit

You have submitted this assignment 0 times.

We'll return to this specific example a little later in this reading, but, for now, let's look at one solution to the problem (you can use the buttons below to step through it):

<< First Step < Previous Step Next Step > Last Step >>

CONFIGURATION 1

1	2	3	4
	5	6	7
9	10	11	8
13	14	15	12

CONFIGURATION 38

1	2	3	4
5		6	7
9	10	11	8
13	14	15	12

CONFIGURATION 39

1	2	3	4
5	6		7
9	10	11	8
13	14	15	12

CONFIGURATION 40

1	2	3	4
5	6	7	
9	10	11	8
13	14	15	12

CONFIGURATION 41

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

**CONFIGURATION 42**

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

This solution consisted of 42 board states (41 moves). Given this result, and given the scale of the problem, there are some questions we might ask ourselves:

- Is this solution optimal? That is, is there a solution out there that requires fewer than 41 moves?
- How many of the ~10 trillion possible board states do I need to consider in order to find a solution? Could we do things more efficiently?

For the rest of this reading, we're going to build on these ideas by building on the things we started talking about in last week's reading and formalizing some of those ideas to develop general-purpose pathfinding algorithms, and then we'll also spend a little bit of time thinking about how those algorithms perform.

### 3) Pathfinding in the Abstract

Our approach to pathfinding will be very similar to our code for flood fill from last week. In that example, we maintained an "agenda" (or "work queue") of pixels that we eventually needed to color in, and then we repeatedly pulled one pixel off of the agenda, colored it in, and then added neighboring pixels of the right color back to the agenda. Here, we will do something similar, but our agenda will consist of *paths* from some starting state to some other state. But, we'll still work our way through this process in the same way: we'll repeatedly pull a path out of the agenda, consider it, and add its child paths (new, longer paths!) onto the agenda. Here is our approach outlined in pseudocode:

To find a path from a state  $t$  to some other state:

- Initialize an **agenda** (list of paths to consider) containing a single path that consists of only our starting state
- Initialize a **visited set** (all states that have ever been added to the agenda) to contain only the starting state
- Repeat the following:
  - Remove one path ( $t \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_N$ ) from the agenda
  - For each "neighbor" state  $n$  of  $s_N$  (for each state  $n$  directly connected to  $s_N$  via an edge):
    - If  $n$  is in the visited set, skip it
    - Otherwise, if  $n$  satisfies our goal condition, return the path ( $t \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_N \rightarrow n$ )
    - Otherwise, add  $n$  to the visited set and add the path ( $t \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_N \rightarrow n$ ) to the agenda

until either we find a path or the agenda is empty (in which case no path exists)

So, there's our high-level approach, and it maybe doesn't sound like much: we're going to keep track of all the paths we need to consider, and then consider them one after the other until we find one that satisfies the goal condition we're looking for. But, while this approach doesn't sound too impressive on the face of it ("just keep looking at new paths until you find one that works"), it's taking advantage of two useful properties of computers: they're good at doing a lot of things really quickly, and they're good at remembering lots of things. Putting that all together, these ideas will end up giving rise to programs that feel quite smart and that can compute some impressive results.

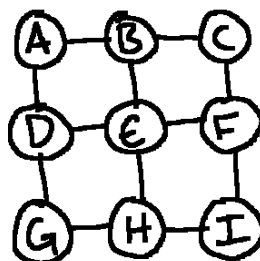
### 3.1) Order Matters!

The approach outlined above is almost complete, but it turns out that one key piece of information is missing.

As we work through this process, we'll generally have multiple paths in the agenda (often, a whole lot of paths!). But, the first step in the loop in our algorithm simply says to "remove one path from the agenda;" it says nothing about which path we should remove! And, similarly, it talks about "adding [new paths] to the agenda," but it doesn't say anything about where we should put those new paths!

It turns out that the order in which we consider the paths has a big effect on the way that this approach "explores" the graph it's given and also on the nature of the paths that we ultimately return. So, it will be worth taking some time to explore these effects.

Let's see how this process plays out on a small example graph:





How many vertices does this graph contain?

Submit

You have submitted this assignment 0 times.

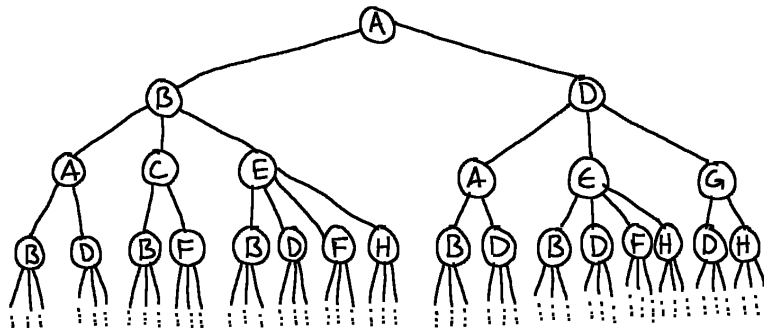
How many edges does this graph contain?

Submit

You have submitted this assignment 0 times.

Let's now consider the process of finding a path from  $A$  to  $I$  in this graph, using the approach we outlined above, and we'll try it with a few different rules about ordering. All of these processes are going to start the same way: our agenda will contain a single path (the path containing only  $A$ ); then we will remove that path and add its child paths (two paths, one representing  $AB$  and one representing  $AD$ ). After that point, we will start to see the differences that arise from different choices about the order in which we consider the paths from the agenda.

To help us with this, we'll visualize things in a slightly different way, using a structure called a "rooted tree," which we'll use to try to represent all possible paths through the little graph from above. Here is an example showing the paths starting at  $A$  (note that we're only showing a portion of the whole tree):



Unlike our original graph drawing, the diagram here represents *paths* starting at  $A$  (the "root vertex"); and we explicitly diagram the idea that paths are sequences of states, and, in theory, a path can contain the same state more than once.

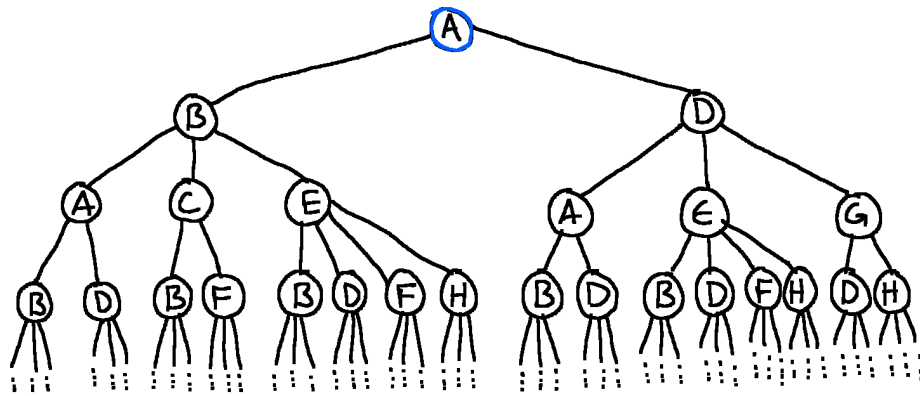
### 3.1.1) Example One: Replace First Path On Agenda With Its Children

Let's start by considering the following rule for our agenda: each time we want to remove a path from the agenda, we will remove the element from the *front* of the agenda; and, when we add neighbors on to the agenda, we'll also add them to the *front* of the agenda.

The example below shows the start of how this process evolves when searching from  $A$  to  $I$  in the graph above.

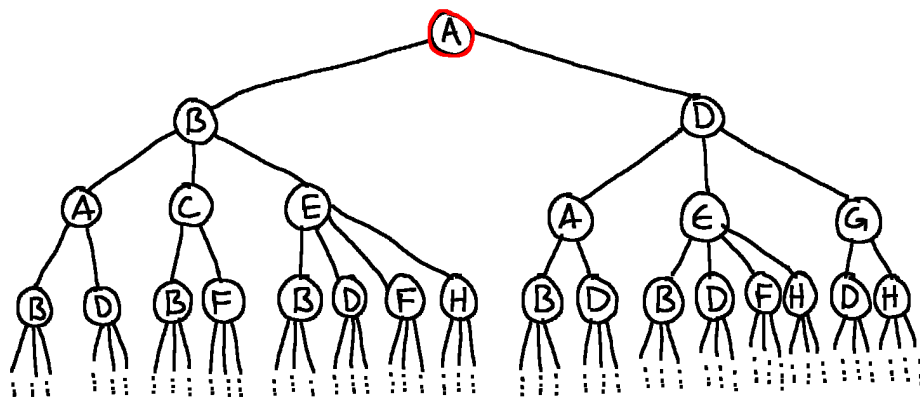
Use the buttons below to step through this example, paying careful attention to how the agenda and the visited set evolve, as well as the order in which we're considering paths. In these graphs, we will color paths that have been added to the agenda but not yet considered in blue; and the path currently being considered will be shown in red.

<< First Step < Previous Step Next Step > Last Step >>

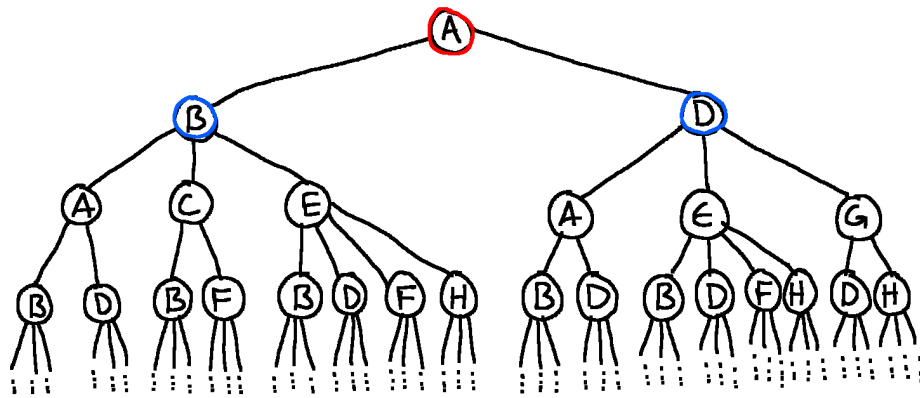
**STEP 1****Agenda:** (*A*)**Visited:** *A*

At the start of this process, we have a single path on our agenda, the path containing only our starting state *A*. Since we have added a path containing *A* to the agenda, we also mark *A* as visited.

Our next step is to remove the first path off of the agenda. In this case, we don't have much choice: we'll remove the path (*A*).

**STEP 2****Agenda:****Visited:** *A*

Now we have removed the path (*A*) from the agenda. Our next step will be to add its children to the front of the agenda. How many paths will we add to the agenda, and what will those paths be? How will our visited set change?

**STEP 3**

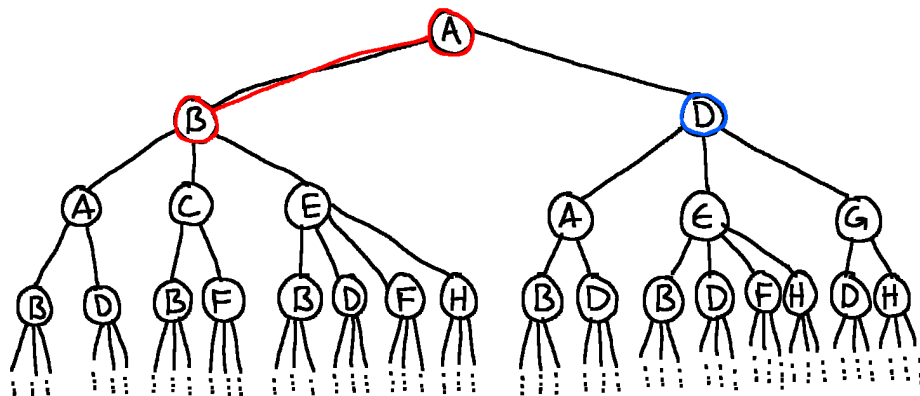
**Agenda:**  $(A \rightarrow B), (A \rightarrow D)$

**Visited:**  $A, B, D$

Here, we've added two new paths to the agenda,  $(A \rightarrow B)$  and  $(A \rightarrow D)$ . We determined these paths by looking at  $A$  in our original graph and noticing that  $B$  and  $D$  are directly connected to  $A$  by edges. There is a question of the order in which we add these child paths to the agenda, but for now, let's keep them in alphabetical order.

Since we have added paths containing  $B$  and  $D$  to the agenda, we also add  $B$  and  $D$  to our visited set.

Since we've not yet found the goal, we'll continue on by pulling the next path out of the agenda. According to our rule, which path will we select?

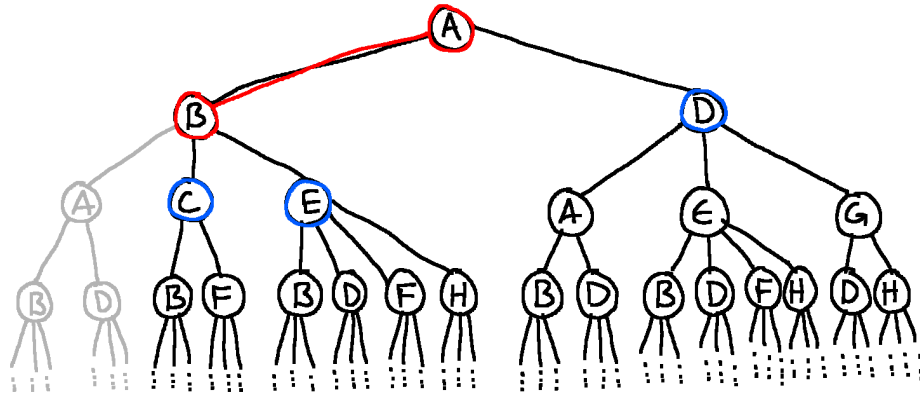
**STEP 4**

**Agenda:**  $(A \rightarrow D)$

**Visited:**  $A, B, D$

Our next path to consider is  $(A \rightarrow B)$ , since it was the first path in our agenda. We have removed it from the agenda, and our next step is to add its children to the agenda. How will the agenda and the visited set change when we do this?

### STEP 5



**Agenda:**  $(A \rightarrow B \rightarrow C)$ ,  $(A \rightarrow B \rightarrow E)$ ,  $(A \rightarrow D)$

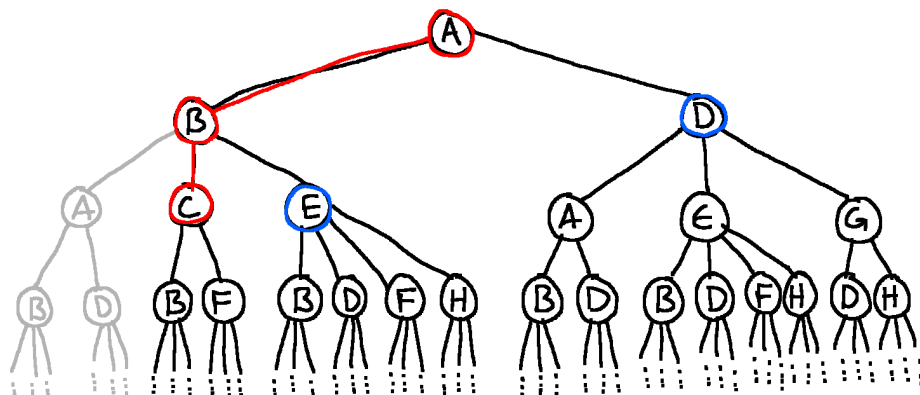
**Visited:**  $A, B, D, C, E$

Recall that our rule here is that we are adding new paths to the front of the agenda (to the same side that we are removing from). So  $(A \rightarrow D)$  stays at the end of the agenda, and we added our two new paths to the front.

Before moving on, let's take a moment to discuss what happened here.  $B$  is directly connected to three other states in our original graph, so why did we only add two paths? The key lies in the fact the  $A$  was already in our visited set before considering each of these new paths; as such, we do *not* add the path  $(A \rightarrow B \rightarrow A)$  on to the agenda. We've greyed out part of the tree structure here to show that we will never consider those paths.

But there are two neighbors of  $B$  that we've not yet considered; each of those generates a new path that we add to the agenda, and we update our visited set accordingly. Next, we'll continue by again pulling the first (left-most) path off of the agenda.

### STEP 6

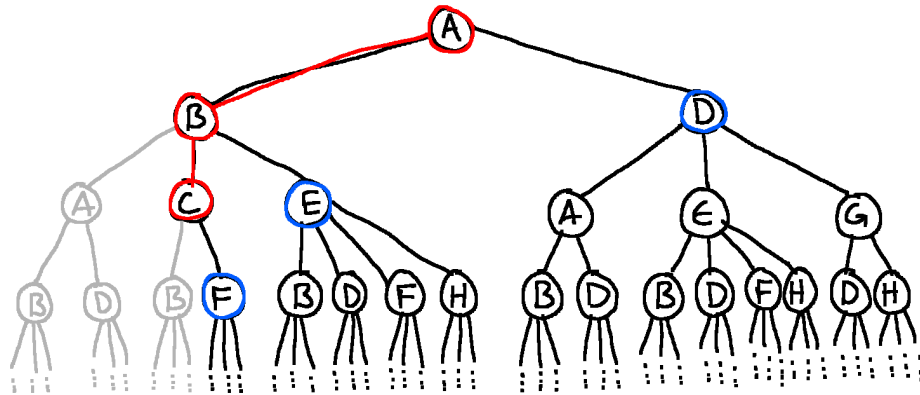


**Agenda:**  $(A \rightarrow B \rightarrow E), (A \rightarrow D)$

**Visited:**  $A, B, D, C, E$

We have pulled the first path  $(A \rightarrow B \rightarrow C)$  off the agenda. Next, we'll consider its children. What new paths will be added to the agenda, and how will the visited set be updated?

### STEP 7



**Agenda:**  $(A \rightarrow B \rightarrow C \rightarrow F), (A \rightarrow B \rightarrow E), (A \rightarrow D)$

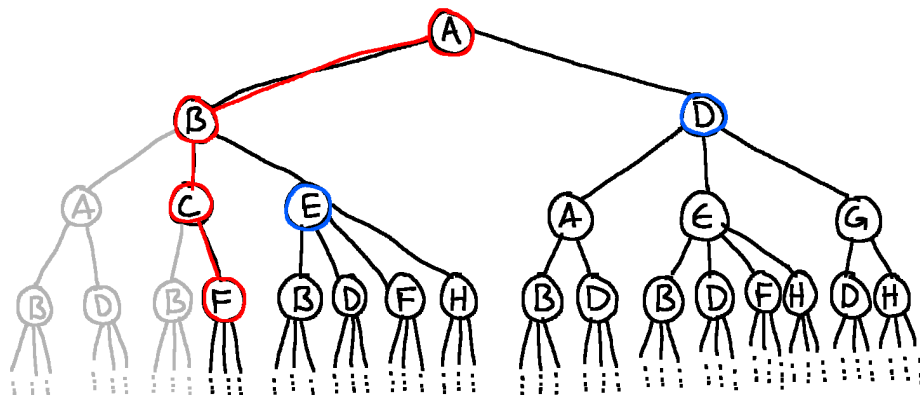
**Visited:**  $A, B, D, C, E, F$

Importantly, we only get one new path added to the agenda here,  $(A \rightarrow B \rightarrow C \rightarrow F)$ . Note that  $(A \rightarrow B \rightarrow C \rightarrow B)$  was another possibility, but we did not add it because we have already visited  $B$  (and our algorithm says that we shouldn't add any new paths ending in a state that we've already visited).

Because we added a path containing  $F$  to the agenda, we also add  $F$  to our visited set.

Let's carry out one more step here, pulling one more path off the agenda.

### STEP 8



**Agenda:**  $(A \rightarrow B \rightarrow E), (A \rightarrow D)$

**Visited:**  $A, B, D, C, E, F$

According to our rule, we pull the first (left-most) path off of the agenda each time. In this case, we pull ( $A \rightarrow B \rightarrow C \rightarrow F$ ). We'll stop the updates here, but our next step would be to loop over  $F$ 's neighbors; what will happen then, following the algorithm from above?

What path from  $A$  to  $I$  will ultimately be found by this approach when it finishes? Enter your answer as a sequence of state names with no punctuation, e.g. DCBA:



You have submitted this assignment 0 times.

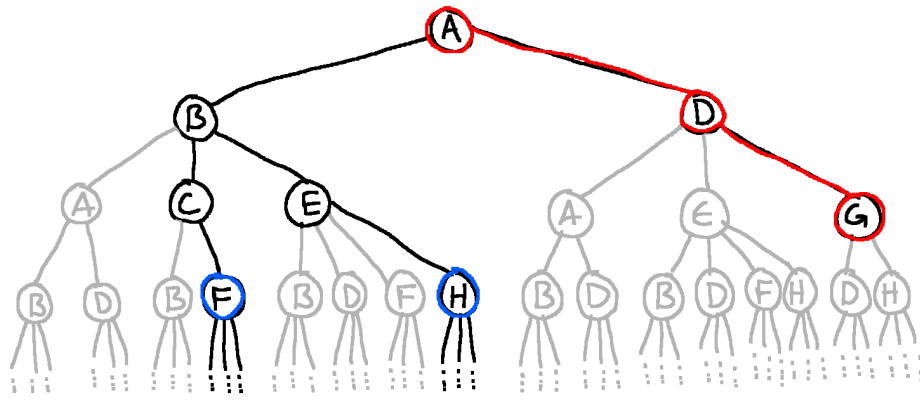
But, let's think not only about the solution here; let's look also at the way in which this algorithm explored the graph. This approach seemed to be going all the way down one path as far as it could, before coming back to consider other paths. Graphically, this approach is growing a single path as deeply as it can before going back and looking at other paths. This idea (of exploring the entire depth of the tree before really considering any of the breadth of the tree) is referred to as a **depth-first search** (or **DFS** for short).

### 3.1.2) Example Two: Replace Last Path On Agenda With Its Children

Now, let's take a look at another approach. Here, we are again looking for a path from  $A$  to  $I$ , but, instead of replacing the *first* (left-most) path in the agenda with its child paths, we're instead going to replace the *last* (right-most) path in the agenda with its children. Before clicking through the examples below, try walking through this example on paper in its entirety. You don't need to draw the whole tree structure, but try to work through how the agenda and the visited set will evolve as we perform this search, as well as the path we ultimately return using this approach. Once you have an idea of what you're expecting, click through the example below to verify:

<< First Step < Previous Step Next Step > Last Step >>

**STEP 1**



**Agenda:**  $(A \rightarrow B \rightarrow C \rightarrow F), (A \rightarrow B \rightarrow E \rightarrow H)$

**Visited:** A, B, D, C, E, G, F, H

What path will ultimately be found by this approach? Enter your answer as a sequence of state names with no punctuation, e.g. DCBA:

Submit

You have submitted this assignment 0 times.

Once again, we cut off the example a little bit before the end, but, hopefully, we can see that this approach is characteristically different from the first two orderings we tried. This approach is called **breadth-first search** (or **BFS** for short) because we are exploring the entire breadth of the tree before considering paths deeper down the tree. Said another way, this approach considers all paths of a given length  $n$  before considering *any* paths of length  $n + 1$ .

### 3.1.4) Order Matters: Summary

In this section, we've looked at a couple of *very slight* variations of the algorithm we presented above. The only thing we changed was how we decide which path to consider next when there are multiple paths on our agenda. But, this slight difference in ordering had a big effect on the way in which our algorithm explored the graph; and, ultimately, it will have a big effect on the kinds of paths we return in the end.

The big takeaway here is that, if we always consider the *newest* path that was added to the agenda (i.e., the path that was added to the agenda most recently), we end up performing a *depth-first search*. If we instead consider the *oldest* path (i.e., the one that was added earliest), we instead perform a *breadth-first search*.

Another place that order matters is the order we loop over the neighbors of a vertex. In these examples, we always loop over the neighbors in alphabetical order. But different neighbor ordering may also produce different paths -- particularly for depth-first search, where if we are unlucky in what order we add neighbors to the agenda, we may end up exploring much more of the graph before reaching the goal.

## 3.2) Another Example

In the examples above, we saw these differences play out in a lot of detail. Next, let's take a step back and look at a related example to see if we can get more of a sense of how DFS and BFS differ in terms of the way they explore a graph in a broader sense.

In each of the examples below, we are considering the problem of searching for a path through a grid. In each, we're starting at the yellow square in the middle and trying to find a path to the green spot in the top right.

In this setup, our vertices represent different locations, as  $(r, c)$  coordinates, in this spaces. The edges are such that every vertex is connected to the 4 cells directly adjacent to it -- up, down, left, and right (though, of course, the cells on the edge don't have all four of those edges since we can't go out of bounds).

The following videos show how the various search methods we've developed above explore this space. In each, the yellow square represents the starting location for our search and the green location represents the goal. As each new path is added to the agenda, it is briefly colored in red, and the final path discovered by the search is shown in blue at the end. Throughout the video, locations that have been visited are shown in cyan.

## BFS

In our first example, we'll consider performing a breadth-first search, adding children in right / left / up / down order:

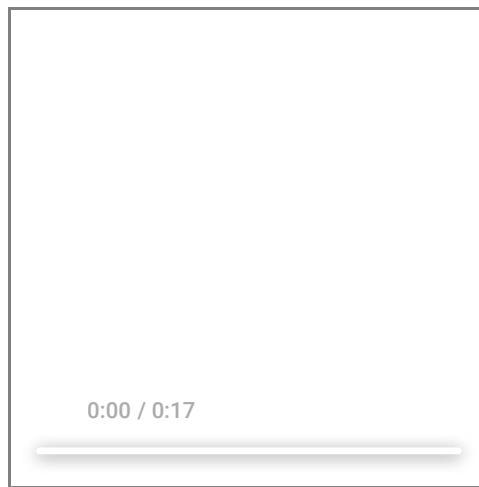


Notice how the search sort of "bubbles" outward from the starting location, gradually considering longer and longer paths until it reaches the goal.

## BFS, Different Order

Here we still perform a BFS, but now we add children in a different order (down / up / right / left):

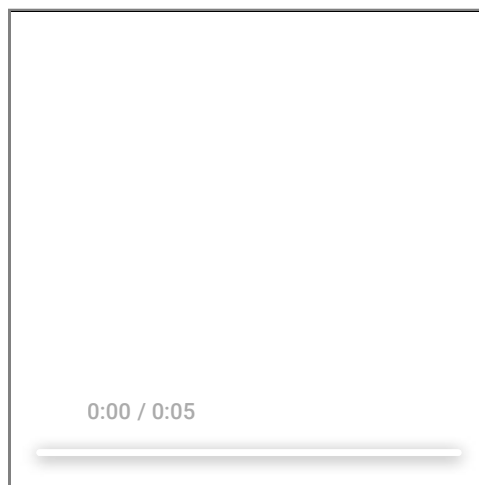




Notice that this search considers the paths in a slightly different order, but the way that it explores the space is largely the same: bubbling outward from the center, considering shorter paths before longer ones.

## DFS

Now let's try a depth-first search instead. Here we add paths in left / down / right / up order:

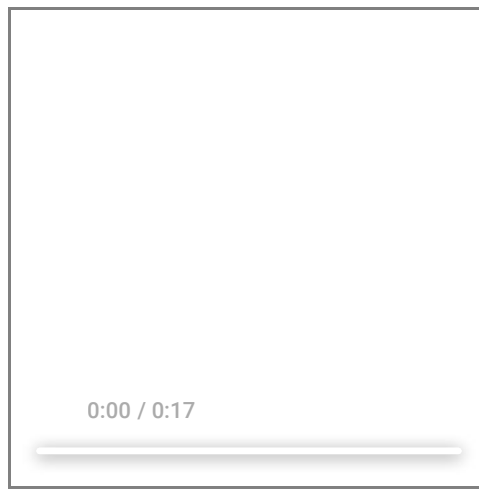


This explored the space in a very different way, snaking a single path outward from the starting position rather than considering lots of paths all over the grid.

Another thing stands out: that was fast compared to BFS! DFS not only found a great path here, but it did it while only visiting a small fraction of the cells visited by a BFS on the same problem. So what conclusion should we draw from this? DFS is just better? Well, maybe in some situations, but let's tweak things a little bit...

## DFS, Different Order

Here, we again run a DFS, but now we add children in right / up / down / left order:



We can still notice the same general pattern: the DFS works by growing one single path outward, and only goes back to consider alternative paths if it reaches a point where it can no longer grow that path (or it finds the goal).

But...yuck! Not only did this DFS visit almost all of the cells in the grid, but it also found a pretty inefficient path to the goal. Indeed, as we mentioned above

**Check Yourself:**

Why exactly was it that the first DFS so great, and this second one was so bad?

Show/Hide

### 3.3) Check Yourself: USA

So hopefully these little animations have given us a little bit of a sense of how the micro-level details of each of the different search methods manifest in terms of a macro-level exploration of the graph. Now let's consider observing similar kinds of patterns on a different graph search problem:

In the video below, each light-grey dot represents an intersection of two or more highways in the United States. The video shows various paths as they are considered as part of planning a path from Smith Center, Kansas (the geographic center of the US) to Cambridge, Massachusetts. Watch the video below to answer: which kind of search is being performed here?

0:00 / 0:51



Submit

You have submitted this assignment 0 times.

### 3.4) Check Yourself: Back to Flood Fill

We can also see these kinds of differences play out in our flood-fill example from last week. Even though we weren't finding paths to start with last week, we were still keeping an agenda of pixels that we eventually wanted to color in, and, by changing the order in which we consider elements from the agenda, we can affect the order in which we color the pixels in. This doesn't affect the end result of the flood fill, but it does affect the intermediate results.

Consider the following three videos, each of which shows the evolution of a flood-fill process:

**Method 1****Method 2****Method 3**

0:00 / 0:43

0:00 / 0:43

0:00 / 0:43

One of the flood fill implementations used a breadth-first approach: the pixel we chose to color in at any given point was the one that had been in the agenda the longest. Which of the videos corresponds to this approach?

You have submitted this assignment 0 times.

One of the flood fill implementations used a depth-first approach: the pixel we chose to color in at any given point was the one that had been added to the agenda most recently. Which of the videos corresponds to this approach?

You have submitted this assignment 0 times.

One of the flood fill implementations used a different approach: choosing a pixel from the agenda at random each time. Which of the videos corresponds to this approach?

You have submitted this assignment 0 times.

## 4) Summary of BFS vs. DFS

So far, we have been thinking fairly abstractly, but we have introduced the notion of graph search, with a specific focus on pathfinding. And we've also seen several examples of the difference between **breadth-first** and **depth-first** approaches. Before we move on to turning these abstract ideas into working Python code, let's take a moment to review some of the differences between BFS and DFS when applied to pathfinding.

### BFS

- To implement a breadth-first search, we add and remove elements from *opposite sides of the agenda*. This approach is known as "first-in, first-out" (commonly written/pronounced as "FIFO") since the element we remove is always the one that was added first.
- BFS is guaranteed to return a shortest path to a goal vertex if such a path exists, regardless of the structure of our graph. Because we consider all paths of length  $n$  before considering any paths of length  $n + 1$ , we know that, when we first encounter a state that satisfies our goal condition, the path we're considering must be optimal (in the sense that there is no shorter path to the goal).
- BFS can run forever if it is being applied to an infinite graph with no solution, but it will always terminate in a finite graph or in an infinite graph where a solution exists.

### DFS

- To implement a depth-first search, we add and remove elements from *the same side of the agenda*. This approach is known as "last-in, first-out" (commonly written/pronounced as "LIFO") since the element we remove is always the last one that was added.

- DFS is guaranteed to find a path to the goal (but not necessarily an optimal one) if such a path exists and if the graph is finite.
- DFS may run forever on an infinite graph, even if a solution exists.

## 4.1) So Is DFS Bad?

In all of the examples we have seen so far, DFS gives us a pretty gnarly path in the end, something that is kind of ugly and far from optimal. And, in the section above, we've stated that the guarantees that DFS offers us are weaker than those that BFS offers us. So, at this point, you may be thinking: why did we even bother introducing DFS? When would you ever want to use it?

The answer is a little bit nuanced, but the short version is that DFS tends to use less memory than BFS. If we consider a graph with a "branching factor" of  $b$  (i.e., every state is connected to  $b$  other states), then there will be around  $b^n$  paths of length  $n$ . This means that there will be around  $b^n$  elements in the agenda when we're considering the last path of length  $n - 1$ . By contrast, the agenda for a DFS will have around  $b \times n$  elements in the agenda when considering a path of length  $n$ ; and, as  $b$  and  $n$  increase,  $b \times n$  is substantially smaller than  $b^n$ . This is especially useful in cases where we don't care about finding the optimal path to a state (sometimes any path will do, or sometimes we're just looking for a particular state and don't care about the path at all!), and, in those cases, DFS can sometimes be the right choice.

## 5) Pathfinding in Python

Now that we've spent some time thinking about graph search as an abstract process, let's think about how to turn it into Python code so that we can realize the power of these techniques.

As always, when we think about turning an abstract idea into something concrete in Python, an important question has to do with our choice of data structures. That is to say, we have several things that we're going to need to represent in Python, as well as operations that we'll need to perform on those things. In some sense, we have infinitely many choices for how to represent each of those things concretely within our program, but, as we've seen in earlier readings and recitations, the data structures we choose to represent those things can have a dramatic effect on the code that we end up writing in terms of clarity, the ease with which we can avoid common bugs, and efficiency. So, let's spend some time here thinking about representation. In particular, there are at least 5 distinct things we need to be able to represent in order to write code to solve paths using the approach outlined above:

- the graph itself (the various states and the connections between them)
- the start state and goal condition
- a candidate path (a sequence of states)
- the agenda
- the visited set

### Check Yourself:

Take a moment to think about different ways that we could represent each of these things. There are multiple options for each; what are some of the tradeoffs?

Show/Hide

**Check Yourself:**

With those decisions in mind, try your hand at writing `find_path`. Note that the structure is very similar to what we saw with flood fill last week, so you may wish to use that as inspiration.

Is the code example in the box above performing a BFS, a DFS, or something else?

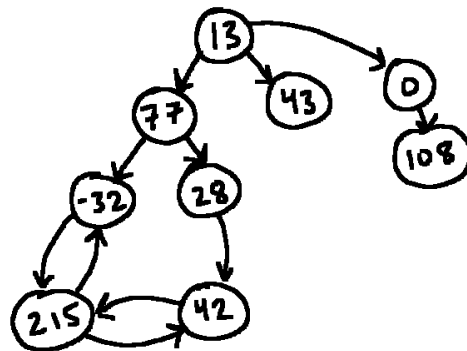


You have submitted this assignment 0 times.

It's perhaps also worth noting that the last line of this function, `return None`, is not strictly necessary in terms of correctness. In Python, if we reach the bottom of a function without seeing a `return` statement, Python will automatically return `None`. Despite this, though, there is something quite nice about keeping that line there even if it's not strictly necessary: it signals to someone reading the code that the behavior of returning `None` at that point is intentional, not just a mistake that accidentally returned `None`.

## 6) A Small Example

With this code in hand, let's walk through a few example programs. Here is a small, contrived example graph that we can start with:



Given the choice of representation we discussed above, how would we represent this graph in Python? Store your answer in a variable called `test_graph`. Note that the states here are integers (and should be represented by Python `int` objects); note also that the edges in this graph are *directed* (one-way), and make sure that your representation accurately reflects this.

```
1 test_graph =
```

You have submitted this assignment 0 times.

Try to answer the questions below without using a Python interpreter.

Consider using the `find_path` function from above *exactly as it is written*. How many states will be in the path we return from calling `find_path(test_graph, 13, lambda state: state == -32)`?

Submit

You have submitted this assignment 0 times.

What would happen if we instead called `find_path(test_graph, 13, lambda state: -32)`? If the function would return a value, type the resulting value in the box below. If it would produce an exception, write `error`. If it would enter an infinite loop, write `infinite`.

Submit

You have submitted this assignment 0 times.

What would happen if we instead called `find_path(test_graph, 13, -32)`? If the function would return a value, type the resulting value in the box below. If it would produce an exception, write `error`. If it would enter an infinite loop, write `infinite`.

Submit

You have submitted this assignment 0 times.

What would happen if we instead called `find_path(test_graph, 0, lambda x: x == 215)`? If the function would return a value, type the resulting value in the box below. If it would produce an exception, write `error`. If it would enter an infinite loop, write `infinite`.

Submit

You have submitted this assignment 0 times.

In the example from the previous question, `find_path(test_graph, 0, lambda x: x == 215)`, what elements were in the visited set by the end of the function call? Enter your result as a Python `set` below.

Submit

You have submitted this assignment 0 times.

**Check Yourself:**

Even though this function currently implements a BFS, it is possible to change a single line such that the function performs a DFS instead. What line could we change to make that happen, and what would we need to change it to?

[Show/Hide](#)

## 7) Revisiting the Representation of a Graph: 15-Puzzle

Let's return now to a problem we discussed earlier in the reading and try to bring this code to bear on the 15-puzzle. To start with, any time we are thinking about using search to solve a new problem, an important question is: how do I represent the *states* (the vertex labels) for this problem, such that my code can be made to work with them? So, let's start there.

**Check Yourself:**

Take a moment and think about how you might represent the states in this graph, keeping in mind that, in the graph we drew all the way back in section 2.1, each vertex was labeled with a representation of a possible layout of the game board.

[Show/Hide](#)



The state representation we just chose for the 15-puzzle is not the only way to do it. A state representation can also have multiple pieces that are glued together (e.g., as different elements of a tuple). But keep in mind that, because we have to store the states in a visited set, a state must be *hashable*.

Regardless of our choice of state representation, in order to use the pathfinding code we've written above to actually solve this problem, we must be able to do two things:

- given a state in our chosen representation, compute all of the states connected to it in the graph (each represented in that same way); and
- given a state in our chosen representation, determine whether it satisfies the goal condition or not.

Given these constraints, which of the following would be a valid hashable state representation for solving the 15-puzzle?

- ☐ the position of the empty square as a (row,column) pair, e.g. (3,4)
- ☐ the tiles as a single flat tuple instead of a tuple-of-tuples, e.g. (2, 6, ..., 10, 7)
- ☐ a set of (row, column, tile) triples, e.g. {(0,0,2), (0,1,6), ...} means location (0,0) contains tile 2, location (0,1) contains tile 6, etc.
- ☐ a frozenset of (row, column, tile) triples, e.g. {(0,0,2), (0,1,6), ...} means location (0,0) contains tile 2, location (0,1) contains tile 6, etc.
- ☐ a string representation of a list-of-lists, e.g. "[[2,6,3,15],...]"
- ☐ a dictionary mapping each numbered tile to a (row,column) pair representing its location, e.g., {1: (1, 2), 2: (1, 3), ..., None: (0, 0)}

Submit

You have submitted this assignment 0 times.

From here, it's worth asking ourselves: can we imagine using our `find_path` function from above to solve this puzzle? Here, we're confronted with a limitation of an early design choice: for the 15-puzzle, generating the dictionary to represent the graph would take a *long* time, and, even worse, the resulting structure would be far too big to fit in memory! It doesn't make sense to spend the time building up that representation just to then go *back* through the whole structure looking for a path. What we'll look at next is an approach to making this whole process more feasible.

Before we dig in, it's worth mentioning that this kind of revision (which often requires deleting and rewriting code!) is completely normal. It takes a lot of practice to make informed decisions about these kinds of choices, and, even with that experience, it's common not to notice until later that we should have structured things differently. So, the process of going back and changing code to use a better internal representation is not an uncommon occurrence.

### Check Yourself:

A place to start when thinking about different ways to represent graphs is to look at our search code and ask ourselves: what questions are we asking about the input graph?

Show/Hide

**Check Yourself:**

Now, try your hand at making the necessary changes to `find_path` to handle this new design, before looking at our code below.

[Show/Hide Our Code](#)

This change allows us to solve things like the 15-puzzle, where constructing the whole graph preemptively would be prohibitively expensive. We'll leave the task of writing the code to solve the 15-puzzle as something to do on the weekend if you're bored and looking for something to do; but let's see if we can make use of this new structure to solve a different problem.

## 8) Another Example: Word Ladders

Let's use our new function-based pathfinding implementation to solve a different kind of puzzle, known as a [Word Ladder](#). In this puzzle, we are given two words, and our task is to find a sequence of words that connect those two words to each other, such that everything in the sequence is a word, and each word in the sequence differs from the one before it and the one after it by exactly one letter.

For example, a puzzle might be given as something like the following (it's common for the starting and ending words to be opposites of each other):

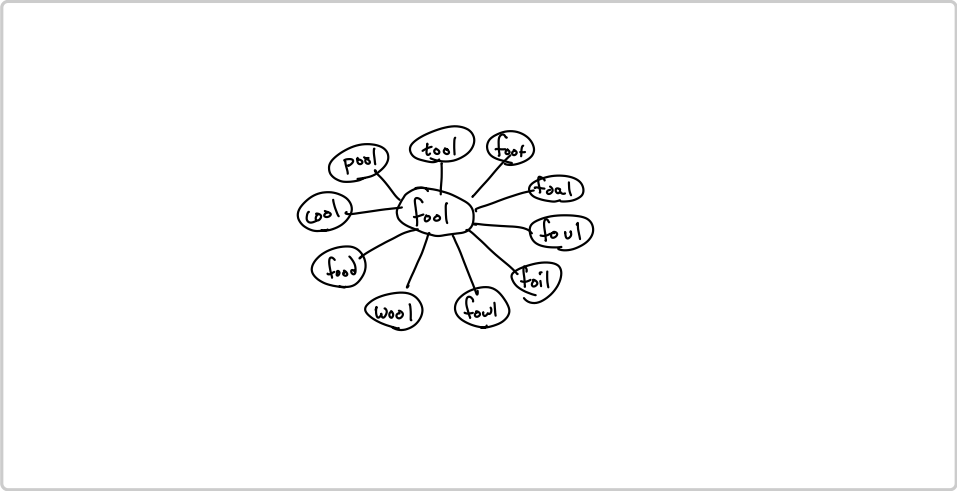
cold  
...  
warm

And a valid answer to the puzzle is:

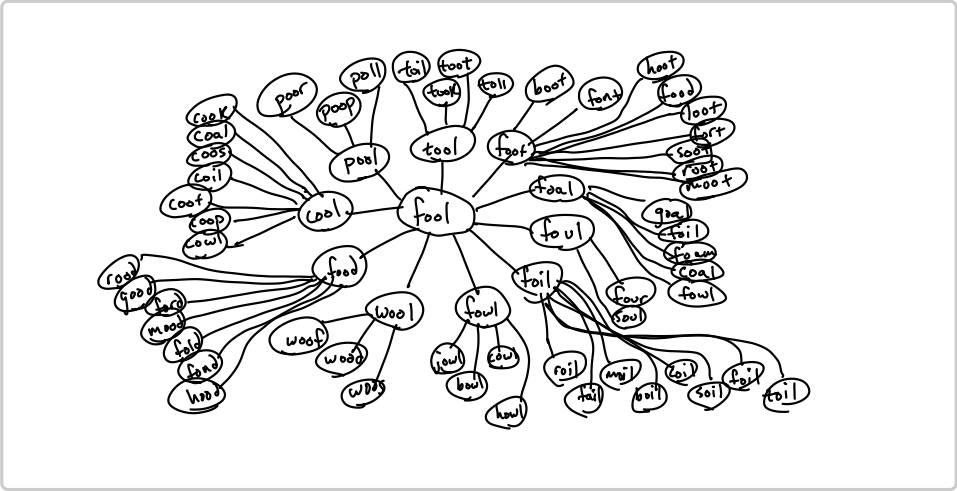
[Show/Hide Solution](#)

This problem is a prime candidate for our pathfinding approaches, and a natural representation is to use the words themselves as states and to have edges connect words that differ only by a single letter. But, like the 15-puzzle, this has a problem. Let's take a look at just a portion of the graph for solving a common puzzle where the starting word is "fool" and the ending word is "sage". Let's start by looking at just the single vertex representing the word "fool":

Isn't it beautiful? But, it's not all that exciting. Let's take a look at a slightly larger piece of the graph, showing all of the vertices that are one step away from "foo1":



and we can carry things further by looking at all vertices that are up to two steps away:



Argh! And as you can imagine, the graph gets ever bigger and more complex the farther we move away, so this is a great example for testing out our new implementation of `find_path`! Try your hand at setting up this problem by filling in the code box below, to set up a search from "patties" to "foaming" in the graph we've been talking about.

You can replicate the `ALL_WORDS` variable on your own machine (if you want to test things out there first) by copying the relevant code from the box below and downloading `words.txt` to the same directory where you're writing your code.

Note that in order to pass the tests on time, your code for `word_ladder_neighbors` below will probably not be able to loop over all of the words in the `ALL_WORDS` set each time it is called, so you'll likely need to find a way to limit your attention only to a smaller subset of possibilities.

```

1  # ALL_WORDS is a set containing all strings that should be considered valid
2  # words (all in lower-case)
3  with open('words.txt') as f:
4      ALL_WORDS = {i.strip() for i in f}
5
6  # replace the following with the starting state
7  start_state = ":"
8
9  # replace this neighbors function:
10 def word_ladder_neighbors(state):
11     """
12     takes a state as input
13     returns all neighboring states (valid words that differ in one letter)
14     """
15     return []
16
17 # replace this goal test function:
18 def goal_test_function(state):
19     """
20     takes a state as input
21     returns True if and only if state matches the goal (the target word)
22     """
23     return False
24
25 # ultimately, these variables will be passed as arguments to the find_path
26 # function to solve for the path between "patties" and "foaming"
27 #
28 #     output = find_path(word_ladder_neighbors, start_state, goal_test_function)
29

```

You have submitted this assignment 0 times.

## 9) Summary

Once again, it feels like we have come a long way. We started by thinking back on last week's example problem of flood fill, and then we expanded and formalized it, arriving at the idea of *graph search*.

Throughout today's reading, we explored several facets of graph search algorithms. We started by talking about graph search in an abstract sense, including discussing the dramatic differences in behavior that arise from changing the order in which we consider elements in the agenda. We also saw several examples of taking problems from the real world and representing them as graphs.

Which of the following statements are true?

- ☐ In order to be a BFS, it's important that new paths are added to (and removed from) the *front* of the agenda, rather than the *end* of the agenda.
- ☐ If BFS finds a path, that path is guaranteed to be optimal (in terms of length).
- ☐ DFS might enter an infinite loop even if the search domain is finite.
- ☐ It is possible that DFS and BFS could return the same path for some problem.
- ☐ It is possible that DFS and BFS could return different paths for some problem.
- ☐ BFS is guaranteed to find a path if one exists, even in an infinite domain.

Submit

You have submitted this assignment 0 times.

We then moved to implementing a pathfinding algorithm in Python, and we spent a lot of time thinking about how to represent different structures related to graph search in Python, eventually ending up at a 20-line implementation of a pathfinding algorithm. There is a lot of interesting stuff going on in those 20 lines and a lot of power from relatively little code! We also worked through examples of using that code to solve a few different kinds of problems, and we used those problems as a guide as we sought to make improvements to the code.