# Section 3: Graph Search Algorithms

Amy Zhao

February 09, 2024

## 1 Graph Basics and Definitions

In general, a **graph** $G = (V, E)$ is a collection of vertices, connected by a collection of edges. Graphs can be undirected or directed. In an undirected graph, each edge is an unordered pair $\{v, w\}$ over vertices which represent the endpoints of the edge. In a directed graph, each edge is an ordered pair $(v, w)$; $v$ points to $w$. We refer to the number of vertices as $|V|$ and the number of edges as $|E|$.

There are two ways to represent graphs: by an **adjacency matrix** and by an **adjacency list**. Although both represent the same information, we see that there are tradeoffs between each representation. These tradeoffs are, in part, dependent on the density of the graph in question. If an undirected graph with $|V|$ vertices is connected, the number of edges $|E|$ is at least linear in $|V|$ ($\Omega(|V|)$). If the graph has no parallel edges, then $|E| = O(|V|^2)$. Informally, we say a graph is **sparse** if the number of edges is "relatively" linear in the number of vertices; a graph is **dense** if its edges are "relatively" quadratic to the number of vertices. Adjacency matrices may be more efficient to encode a dense graph, although most graph algorithms take adjacency lists as inputs.

## 2 Paths, Cycles, and Connectivity

We define a **path** in an undirected graph as a sequence $P$ of vertices $v_1, v_2, \ldots, v_k$ with the property that each consecutive pair $v_i$ and $v_{i+1}$ are joined by an edge in $G$. A path is **simple** is all vertices in the path are distinct. If the first and last vertex in the path are the same, then the path is a **cycle**.

## 2.1 Length $k$ Paths

**Theorem.** Let $G = (V, E)$ be a directed graph with $n$ vertices $v_1, \ldots, v_n$. Recall that an adjacency matrix $M$ of $G$ is an $n$-by-$n$ matrix such that $M_{i,j} = 1$ if $(v_i, v_j) \in E$ and $M_{i,j} = 0$ if $(v_i, v_j) \notin E$. Note that $M_{i,j}$ denotes the entry in row $i$ and column $j$, as shown below.

$$M = \begin{bmatrix} M_{1,1} & M_{1,2} & M_{1,3} & \cdots & M_{1,n} \\ M_{2,1} & M_{2,2} & M_{2,3} & \cdots & M_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{n,1} & M_{n,2} & M_{n,3} & \cdots & M_{n,n} \end{bmatrix}$$

For all $k \in \mathbb{N}$, the entry $(M^k)_{i,j}$ is the number of paths from $v_i$ to $v_j$ using exactly $k$ steps. It may be helpful to recall that, for $k \in \mathbb{Z}^+$, a $k$-step path is a $k-1$ step path with an edge appended to the end. Note, the paths being counted are not necessarily simple paths.

Let $P(k)$ be the predicate which says that the entry $(M^k)_{i,j}$ represents the number of paths from $v_i$ to $v_j$ using exactly $k$ steps. The base case is $k = 0$. By definition, an entry $M_{i,j}$ is equal to one if there is an edge from $i$ to $j$, and $M^0$ is the identity matrix. This implies that there is a single path from some vertex $i$ to itself. This is true as this is simply the zero (empty) path; $P(0)$ holds. Suppose $P(k)$ holds; consider the product $(M^k)M$:

$$(M^k)M = M^{k+1} = \begin{bmatrix} (M^k)_{1,1} & (M^k)_{1,2} & (M^k)_{1,3} & \cdots & (M^k)_{1,n} \\ (M^k)_{2,1} & (M^k)_{2,2} & (M^k)_{2,3} & \cdots & (M^k)_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (M^k)_{n,1} & (M^k)_{n,2} & (M^k)_{n,3} & \cdots & (M^k)_{n,n} \end{bmatrix} \begin{bmatrix} M_{1,1} & M_{1,2} & M_{1,3} & \cdots & M_{1,n} \\ M_{2,1} & M_{2,2} & M_{2,3} & \cdots & M_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{n,1} & M_{n,2} & M_{n,3} & \cdots & M_{n,n} \end{bmatrix}$$

$$= \begin{bmatrix} (M^k)_{1,1}M_{1,1} + \cdots + (M^k)_{1,n}M_{n,1} & \cdots & (M^k)_{1,1}M_{1,n} + \cdots + (M^k)_{1,n}M_{n,n} \\ (M^k)_{2,1}M_{1,1} + \cdots + (M^k)_{2,n}M_{n,1} & \cdots & (M^k)_{2,1}M_{1,n} + \cdots + (M^k)_{2,n}M_{n,n} \\ \vdots & \ddots & \vdots \\ (M^k)_{n,1}M_{1,1} + \cdots + (M^k)_{n,n}M_{n,1} & \cdots & (M^k)_{n,1}M_{1,n} + \cdots + (M^k)_{n,n}M_{n,n} \end{bmatrix}.$$

Notice that each entry $(M^{k+1})_{i,j}$ is equivalent to the sum $\sum_{\ell=1}^{n} (M^k)_{i,\ell} M_{\ell,j}$ where $\ell$ represents some "intermediate vertex", and every $M_{\ell,j}$ is only non-zero (equal to 1) if there is an edge from $\ell$ to $j$. This implies that every term in the summation is non-zero if there are a non-zero number of length $k$ paths from $i$ to $\ell$ and an edge

from $\ell$ to $j$ exists in the graph. We can take each of these length $k$ paths from $i$ to $\ell$ and append the edge from $\ell$ to $j$ to form a length $k+1$ path from $i$ to $j$. Thus, it follows that every term in the matrix for $M^{k+1}$ represents the number of length $k+1$ paths from some vertex $i$ to some vertex $j$, and $P(k+1)$ holds. By the principle of induction, the original claim is is true. □

# 3 Graph Search Algorithms

## 3.1 Breadth-First Search

The basic idea of **breadth-first search** (BFS) is to start from one vertex and explore outward in all possible directions, adding vertices one layer at a time. A vertex fails to appear in any of the layers if and only if there is no path to it.

```
   Input: Graph G = (V, E) given as an adjacency list, starting vertex s
   Post : Vertex is reachable from s if and only if is marked as "explored"
 1 Function BFS(G, s):
 2     Q ← new queue, s ← explored
 3     Q.enqueue(s)
 4     while Q is not empty do
 5         v ← Q.dequeue()
 6         for each edge (v, w) in Adj[v] do
 7             if w is unexplored then
 8                 w ← explroed
 9                 Q.enqueue(w)
10         end
11     end
```

Breadth-first search has time complexity $O(|V| + |E|)$ when the graph supplied is in the form of an adjacency matrix.

## 3.2 Depth-First Search

The main idea of **depth-first search** (DFS) is to start from one vertex and continuously follow the edge leading out until a dead end is reached. The algorithm then backtracks until it encounters an unexplored vertex and repeats the process.

```
Input: Graph G = (V, E) given as an adjacency list, starting vertex s
Post : Vertex is reachable from s if and only if is marked as "explored"
1  Function IterativeDFS(G, s):
2      S ← new stack, S.push(s)
3      while S is not empty do
4          v ← S.pop()
5          if v is unexplored then
6              v ← explored for each edge (v, w) in Adj[v] do
7                  S.push(w)
8              end
9      end
10 Function RecursiveDFS(G, s):
11     s ← explored
12     for each edge (s, v) in Adj[s] do
13         if v is unexplored then
14             RecursiveDFS(G, v)
15     end
```

Depth-first search also has time complexity $O(|V|+|E|)$ when the graph is supplied as an adjacency matrix.

### 3.3 Example: Modifying Reachability

Suppose we have an undirected graph $G = (V, E)$ with the additional property that every edge has a color **red** or **blue**, and we want to determine if there is a path from vertex $u$ to vertex $v$. However, we introduce a limitation to the path we want to find; we would like to determine if there is a path that contains only **red** edges or only **blue** edges. Is it possible to use or modify an algorithm we have seen before?

Consider a path of all **red** edges (the algorithm to find a path with all **blue** edges is analogous). If we want to guarantee a path of a single color, we can search in a modified graph where the **blue** edges are discarded. A path of only **red** edges exist in the original graph if $u$ and $v$ belong to the same connected component in the modified graph.

Thus, the algorithm proceeds by first removing all **blue** edges, which takes $O(|E|)$ times. We run DFS on the resulting graph to determine if we can reach $v$ from $u$. This takes $O(|V| + |E|)$ time, so the overall runtime is $O(|V| + |E|)$.

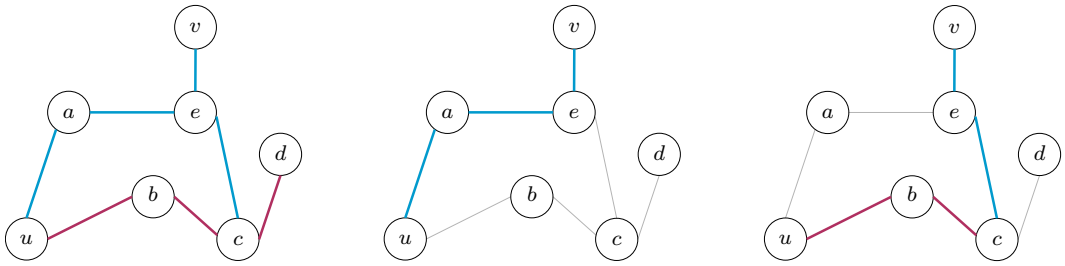



**Figure 3.1:** Example of a graph (left) where it's possible to reach $v$ from $u$ with a path of a single color (center) and a path where all one color appears before the other (right).

Now, suppose we want to decide whether there exists a path from $u$ to $v$ such that within the path, all **red** edges appear before all **blue** edges. We borrow a similar idea from the previous scenario by deleting edges of one color. This time, we need to restore the edges and repeat the deletion process for the other color.

More specifically, we first delete all **blue** edges from the graph and run DFS on the modified graph to determine which vertices are in the same connected component as $u$. We track these vertices as reachable from $u$ using only **red** edges. Then, we restore the **blue** edges and delete the **red** edges. Run DFS again, but start with vertex $v$ in order to determine which vertices are reachable from $v$ using only **blue** edges. We can conclude that a path exists from $u$ to $v$ with the conditions specified if there exists a vertex $w$ that is both reachable from $u$ using only **red** edges and reachable from $v$ using only **blue** edges.

Accounting for the runtime of each operation, we see that this algorithm also has a time complexity of $O(|V|+|E|)$: modifying the edges takes $O(|E|)$, two operations of DFS take $O(|V|+|E|)$, and iterating through the vertices to determine the existence of $w$ takes $O(|V|)$.

## 3.4 Example: Testing Bipartiteness

Recall that a **bipartite graph** is a graph whose vertices can be divided into two sets such that every edge in the graph connects one vertex in $U$ to a corresponding vertex in $V$; no two vertices in the same set share an edge.

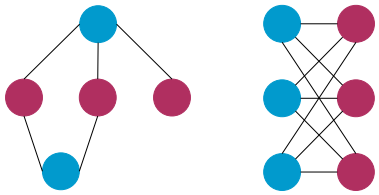


**Figure 3.2:** Examples of bipartite graphs; node colors denote the set that the vertex would belong to if split accordingly.

### 3.4.1 Cycle Parity

**Theorem.** A graph $G$ is bipartite if and only if it does not contain an odd cycle.

Suppose first that $G = (V = A \cup B, E)$ is bipartite, assume by way of contradiction that there exists a cycle $v_1, v_2, \ldots, v_k, v_1$ which is odd. Additionally, assume $v_1 \in A$ without loss of generality. Simple induction implies that $v_i \in A$ when $i$ is odd, and $v_i \in B$ when $i$ is even. However, this means that $v_k, v_1 \in E$ is an edge with both endpoints in $A$, contradicting the assumption that $G$ is bipartite. Thus, the forward direction is true.

Conversely, suppose $G$ is a graph with no odd cycles. Let $d(u, v)$ represent the shortest path between two vertices in $G$; with an arbitrary $u \in V$, we define one set $A$ to be $A = \{u\} \cup \{w \mid d(u, w) \bmod 2 = 0\}$ and the other set $B$ to be any vertex not in $A$ (i.e., $B = V \setminus A$). Now, by way of contradiction, assume there exists an edge $\{w, v\} \in E$ where both vertices belong to the same set. This means that $d(u, w)$ and $d(u, v)$ are both even or both odd. Let $P_{uw}$ and $P_{vu}$ be the shortest paths connecting $u$ to $v$ and $v$ to $u$. If we combine the two paths along the edge $\{w, v\}$ to form the cycle $P_{uw}$—$\{w, v\}$—$P_{vu}$; it's easy to see this cycle has length $1 + d(u, w) + d(u, v)$ which is odd, and thus a contradiction. Therefore, $G$ is bipartite. □

### 3.4.2 Algorithm

Suppose we want to design an algorithm that determines whether a graph is bipartite. It it isn't, we would like the algorithm to return an odd cycle which justifies that the graph is indeed not bipartite. At a high level, the algorithm that accomplishes this task modifies DFS.

1. Initialize arrays to track the parent and color of each vertex.
2. Run DFS an mark each vertex in a new component as **blue**.
3. Each time a vertex $v$ is explored, take note of its parent, and set the color of $v$ to the opposite color of its parent.
4. When iterating over any neighbor of $v$, denoted $w$, we consider if $w$ has been explored and its color. If $w$ has been explored and its color matches $v$, we construct a list of vertices by backtracking in the parent array to construct a cycle before outputting an indication that the graph is not bipartite.
5. If the previous step never occurs, the algorithm outputs an indication that the graph is bipartite.

**Input :** Graph $G = (V, E)$ given as an adjacency list
**Output:** *True* if $G$ is bipartite; *False* and odd cycle $C$ if $G$ is not bipartite

```
1  Function DFS(G):
2      for v ∈ V do
3          visited[v] ← 0
4          parent[v] ←⊥
5          color[v] ←⊥
6      end
7      for v ∈ V do
8          if visited[u] = 0 then
9              color[v] ← 0
10             Explore(G, v)
11         return True
12     end
13 Function Explore(G, v):
14     visited[u] ← 1
15     for w ∈ Adj[v] do
16         if visited[w] = 1 and color[w] = color[v] then
17             construct C by backtracking through parent array from v to u
18             return False, C
19         if visited[u] = 0 then
20             parent[w] ← v
21             color[w] ← (color[v] +1) mod 2
22             Explore(G, w)
23     end
```

### 3.4.3 Approaching Correctness

It's very easy to see that if the algorithm returns *False* and an odd cycle $C$, then $G$ is not bipartite by using the proof from the discussion on cycle parity. However, it is less clear that the algorithm is correct when it returns *True*. This is because DFS does not consider all cycles during its execution, so we cannot immediately conclude that there are simply no odd cycles in $G$.

There are two portions to the proof of correctness: (1) proving that $G$ is two-colorable if and only if $G$ does not contain an odd cycle, and (2) if the algorithm returns *True*, then $G$ is two-colorable. Together, this implies that if the algorithm returns *True*, then $G$ has no odd cycles and is thus bipartite. We'll prove (1) first.

Suppose first that $G$ does not contain any odd cycles; this means that $G$ is bipartite. Let $B$ and $R$ denote the sets in the partition of vertices. We color the vertices in $B$ **blue**, and color the vertices in $R$ **red**. For any edge $\{u, v\} \in G$, we know that $u$ and $v$ cannot belong to the same set, as $G$ is bipartite. So, no adjacent vertices share a color, and it follows that $G$ is two-colorable.

Conversely, suppose $G$ is two-colorable, but by way of contradiction, $G$ contains an odd cycle. Denote this cycle $v_1 v_2 \dots v_k v_1$, where $k$ is odd. Without loss of generality, assume $v_1$ is colored **blue**. Because the subsequent vertices in the cycle are connected by an edge, they must alternate in color. However, this means that $v_k$ and $v_1$, which share an edge, are both colored **blue**. Thus, we have a contradiction, and the original claim is proven. □

Why does the algorithm presented above necessarily tell us that $G$ is two-colorable when it outputs *True*? Consider an edge $\{v, w\} \in E$, and suppose without loss of generality that $w$ is visited by the algorithm before $v$. When the algorithm calls `Explore`($G$, $v$), visited[$w$] = 1. If the algorithm returned *True*, then we must also have that color[$w$] $\neq$ color[$v$]. Additionally, the algorithm only colors a vertex if and only if the vertex was previously unvisited, which implies that the colors of $v$ and $w$ do not change; rather, they remain different colors upon the algorithm's termination. Thus, no adjacent vertices share a color, and $G$ is two-colorable. □

From the previous two proofs, we can finally conclude that the algorithm will correctly identify bipartite graphs or output an odd cycle if $G$ is not bipartite. The

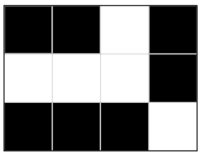
runtime analysis is fairly simple: populating the arrays which track the parent vertices and the colors adds at most $O(|V| + |E|)$, and standard DFS is $O(|V| + |E|)$ which implies that the overall runtime is $O(|V| + |E|)$.

### 3.5 Example: Maze Searching

Suppose we are given a maze in the form of an $m \times n$ matrix where empty cells are represented as '$\cdot$' in the matrix, and walls are represented as '+' in the matrix. Given an entrance (as an element of the matrix), we want to find the shortest path from the starting point to the nearest exit. In every step, we can move up, down, left, or right, and an exit is defined as an empty cell that is located along the border of the maze.

As an example, consider the maze given by:

$$m = \begin{bmatrix} + & + & \cdot & + \\ \cdot & \cdot & \cdot & + \\ + & + & + & \cdot \end{bmatrix}$$



If the starting point is $(1, 2)$, then the nearest exit, located at $(0, 2)$, is one step away. At a high level, this problem asks us to find the shortest path in a matrix, so we should consider breadth-first search which explores cells in order by their distance from the starting point. Every time we explore a cell, we'll check if it's a possible exit, and keep track of its distance from the starting point. If it is, we can terminate the search algorithm and output the distance. A slightly expanded explanation of the algorithm follows:

1. Initialize an empty queue to store explored nodes.
2. Add the starting point to the queue with distance 0, and mark it as explored.
3. While we have not yet reached an exit, and the queue is non-empty, we pop from the queue a vertex $u$ and explore its neighbors (in the four directions specified previously). If the neighbor is an exit, we can terminate the loop and return its distance from the starting location. Otherwise, we mark the neighbor as explored and add it to the queue with its corresponding distance.
4. If we fail to find an exit, we return $-1$.

```
Input  : Matrix m, starting point s = (row, column)
Output: Distance of nearest exit; −1 if no exit is possible
Function FindNearestExit(m, s):
    rows ← len(m)
    cols ← len(m[0])

    start_row ← s[0]
    start_col ← s[1]

    dirs ← [(1, 0), (−1, 0), (0, 1), (0, −1)]
    m[start_row][start_col] ← '+'

    q ← new queue
    q.enqueue([start_row, start_col, 0])

    while q is not empty do
        u ← q.dequeue()
        u_row ← u[0]
        u_col ← u[1]
        u_dist ← u[2]

        for each d in dirs do
            next_row ← u_row +d[0]
            next_col ← u_col +d[1]

            if next cell in direction d is a valid unvisited location then
                if empty cell is a valid exit then
                    return u_dist +1

                m[next_row][next_col] ← '+'
                q.enqueue([next_row, next_col, u_dist +1])
        end
    end
    return −1
```

This example illustrates a key difference between graph representations. Although the algorithm relies on BFS, the input "graph" is given as an adjacency matrix. Adding to and popping from the queue are $O(1)$ operations; checking the neighbors and marking them as explored also takes $O(1)$. However, in the worst-case, it is possible that the algorithm runs for $O(m \times n)$ nodes before the algorithm terminates. Thus, the runtime is $O(m \times n)$.