Introductory Computer Programming: 2017-18
Shortest Path Algorithms

Soumik Purkayastha

MD-1710
## Contents

1 Project Proposal

2 Report

2.1 Introduction ......................................................... 3

2.2 Basic Definitions .................................................... 4

2.3 Representing Graphs ................................................. 8

2.4 Generating Random Graphs

2.4.1 Introduction ..................................................... 10

2.4.2 Procedure ......................................................... 10

2.5 Shortest Path Problem .............................................. 11

2.5.1 Statement of Problem ........................................... 11

2.5.2 Variants .......................................................... 11

2.5.3 Discussion on Shortest Paths .................................. 12

2.5.4 Some important routines ....................................... 13

2.5.5 Some important properties: ................................... 13

2.6 Breadth First Search (BFS) ......................................... 14

2.6.1 Introduction ..................................................... 14

2.6.2 Problem statement .............................................. 14

2.6.3 Remarks ........................................................ 14

2.6.4 Description of algorithm ...................................... 14

2.6.5 Working of algorithm .......................................... 15

2.6.6 BFS returns shortest path ..................................... 15

2.7 Bellman Ford Algorithm ............................................ 16

2.7.1 Introduction ..................................................... 16

2.7.2 Algorithm ........................................................ 16

2.7.3 Application: Arbitrage opportunities ...................... 17

2.8 Shortest Paths in DAGs ............................................. 18

2.8.1 Introduction ..................................................... 18

2.8.2 Algorithm ........................................................ 18

2.8.3 Application: PERT analysis .................................. 19

2.9 Dijkstra’s Algorithm ................................................ 20

2.9.1 Introduction ..................................................... 20

2.9.2 Application: Erdös numbers .................................. 21

2.10 All Pairs Shortest Paths Problem ................................ 22

2.10.1 Introduction ..................................................... 22
2.10.2 Application of what we have learnt ................................. 22
2.10.3 Topics for further study .................................................. 22

3 References .............................................................................. 23

4 Appendix .................................................................................. 23
1 Project Proposal

A graph is a data structure, consisting of a set of objects in which some pairs of the objects are in some sense ‘related’. Graphs are of great importance in the field of computer science - a fundamental area of interest being the algorithms used to work with such objects. In this project, we shall discuss shortest path problems. As the name suggests, we shall try to calculate the shortest path between two nodes of a given graph.

First, we shall briefly discuss the basic definitions in graph theory.

Next, we shall discuss how to represent graphs (we shall only discuss the two most common computational representations of graphs - as adjacency lists or as adjacency matrices).

The breadth-first-search (BFS) algorithm is next recognised as a shortest path finding algorithm on unweighted graphs. We shall briefly discuss this topic, not dwelling on the theoretical development, but on understanding how the BFS helps us better understand shortest path problems.

Next, we shall focus on the single source shortest path problem: given a graph \( G = (V, E) \), we wish to find the shortest path from any given source node \( s \in V \) to any other destination node \( v \in V \). We shall see that this problem may be used to solve many other problems, such as the following:

1. **Single destination shortest path problem**: finding a shortest path to a given destination node \( u \) from each node \( v \).

2. **Single-pair shortest-path problem**: Find a shortest path from \( u \) to \( v \) for given nodes \( u \) and \( v \).

Keeping in mind our interest in shortest path problems, we shall study three such algorithms in detail. These algorithms are as follows:

1. **Bellman-Ford algorithm**: which solves the single-source shortest-paths problem in the general case in which edges can have negative weight.

2. **A linear-time algorithm** for computing shortest paths from a single source in a directed acyclic graph.

3. **Dijkstra’s algorithm**: an improvement on (2), under the restriction of \( E \) having only non-negative components.

We shall also discuss the problem of all-pairs shortest-paths problem which involves searching for a shortest path from \( u \) to \( v \) for every pair of nodes \( (u, v) \in V \times V \). Instead of studying algorithms for this problem in great detail, we shall try and apply our understanding of the single source shortest path problem here and see the utility of the same.

We will finally look at an application for each of the three algorithms discussed.

\(^1\)Graphs in which each edge can be considered to have unit weight.
• For Bellman Ford algorithm, we will discuss arbitrage opportunity. It is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 46.4 Indian rupees, 1 Indian rupee buys 2.5 Japanese yen, and 1 Japanese yen buys 0.0091 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $46.4 \times 2.5 \times 0.0091 = 1.0556$ U.S. dollars, thus turning a profit of 5.56 percent. Suppose that we are given $n$ currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table $R$ of exchange rates, such that one unit of currency $c_i$ buys $R[i, j]$ units of currency $c_j$. Our aim would be to find an algorithm to determine whether or not there exists a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

• For shortest paths in DAG: we will discuss PERT analysis. PERT is an acronym for program evaluation and review technique. A PERT chart presents a graphic illustration of a project as a network diagram consisting of numbered nodes representing events, or milestones in the project linked by labelled directed edges representing tasks in the project. The direction of the arrows on the lines indicates the sequence of tasks. A path through this DAG represents a sequence of jobs that must be performed in a particular order. A critical path is a longest path through the DAG, corresponding to the longest time to perform an ordered sequence of jobs. The weight of a critical path is a lower bound on the total time to perform all the jobs. We can find a critical path by negating the edge weights (representing time needed for performing a certain task) and running the proposed algorithm.

• For Dijkstra’s algorithm we will talk about Erdős numbers and how to find the same for a given graph of authors. The Erdős number describes the "collaborative distance" between mathematician Paul Erdős and another person, as measured by authorship of mathematical papers. Given a graph with nodes representing authors of mathematical papers, and an edge between two nodes signifying that the two authors have been co-authors of at least one paper, our problem would be to find the shortest path between a given author and Paul Erdős - which is essentially a single source shortest path problem. We will implement Dijkstra’s algorithm on a collaborative graph generated from a dataset found online.
2 Report

2.1 Introduction

Graphs are mathematical structures used to model pairwise relations between objects. We shall take a graph to be made up of vertices (also called nodes or points) which are connected by edges (also called arcs or lines). A graph may be undirected, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be directed from one vertex to another. Graphs are one of the prime objects of study in discrete mathematics.

Note: To implement graphs in R, we will be making use of the igraph package. It is aimed at network analysis and and visualisation. It contains routines for simple graphs and network analysis. It can handle large graphs very well and provides functions for generating random and regular graphs, graph visualization, centrality methods and much more.

This report is organised as follows:

1. We will discuss some basic definitions involved with graphs.
2. Next, we will discuss two ways of representing graphs.
3. We shall talk a little about generating random graphs, giving special attention to the Erdős–Rényi random graph model, and directed acyclic graphs (DAG).
4. At this point, we will state the Shortest Path Problem and talk about various variants of the problem. We shall also discuss some salient features of shortest path(s) in a given graph. Some important routines will be introduced and some important properties of shortest paths will be introduced. These properties will be useful in proving correctness of the algorithms we wish to discuss.
5. We shall look at the Breadth First Search algorithm and discuss how it returns the shortest path in a graph.
6. Next, we will discuss three algorithms, establishing their correctness, analysing running time and looking at one application for each.
7. Finally, we will identify that even though all pairs shortest path problem can be solved using the algorithms we discuss in (6), we can do better by looking at more evolved algorithms.

All references have been taken from Introduction to Algorithms, Third Edition (2009) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.

---

2More information may be found here.
2.2 Basic Definitions

Definition 2.2.1 A graph $G = (V, E)$ consists of $V$, a non-empty set of vertices (or nodes) and $E$, a set of edges. Each edge has either one or two vertices associated with it, called its endpoints.

Definition 2.2.2 A directed graph or digraph $G$ is a pair $(V, E)$, where $V$ is a finite set and $E$ is a binary relation on $V$. $V$ is called the vertex set of $G$ and its elements are called vertices. The set $E$ is called the edge set of $G$ and its elements are called edges. In Figure 1 below, we see a directed graph with 8 nodes and 12 edges.

\[
V = \{1, 2, \ldots, 7, 8\}
\]
\[
E = \{(1, 2), (1, 3), (2, 1), \ldots, (7, 2), (8, 5)\}
\]

Figure 1: Directed Graph
Definition 2.2.3 In an **undirected graph**, the set $E$ consists of unordered pairs of vertices. Self loops are forbidden. $(u,v)$ and $(v,u)$ are considered to be the same edge. Every edge is hence, an ordered of exactly two distinct vertices. In Figure 2, we see a directed graph with 8 nodes and 6 edges.

\[
V = \{1, 2, \ldots, 7, 8\}
\]

\[
E = \{(1, 4), (2, 7), (2, 8), \ldots, (6, 7)\}
\]

![Figure 2: Undirected Graph](image-url)
Definition 2.2.4 If \((u, v)\) is an edge in a directed graph \(G = (V, E)\) then \((u, v)\) is incident from (or leaves) vertex \(u\) and is incident to (or enters) vertex \(v\). If \((u, v)\) is an edge in a graph \(G = (V, E)\), then vertex \(v\) is adjacent to vertex \(u\). The degree of a vertex in an undirected graph is the number of edges incident on it. Nodes in directed graphs have in and out degrees. In Figure 3, we see that

- Edges incident on node 3: \{\((3, 7), (3, 8)\)\}
- Edges incident to node 7: \{\((1, 7), (3, 7)\)\}
- Nodes adjacent to node 5: \{2, 6\}
- In/Out degree of node 2: 3/1
Definition 2.2.5  **Weighted graphs** are graphs for which each edge has an associated weight, typically given by a weight function \( w \): \( \rightarrow \mathbb{R} \). In Figure 4, we see a weighted digraph. The weights here are all lying in (0, 1), but that need not be the case in general.

![Graph with 8 nodes and 14 edges](image)

Figure 4: A weighted digraph.
2.3 Representing Graphs

There are two standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. Either way is applicable to both directed and undirected graphs.

The adjacency-list representation is usually preferred, because it provides a compact way to represent sparse graphs. A potential disadvantage of the adjacency list representation is that there is no quicker way to determine if a given edge $(u, v)$ is present in the graph than to search for $v$ in the adjacency list of $u$ given by $Adj[u]$.

The adjacency list approach is preferred for dense graphs, and for when we need to be able to tell quickly if there is an edge connecting two given vertices.

As far as representing weighted graphs go, the weight $w(u, v)$ of the edge is simply stored with vertex $v$ in $Adj(u)$ - adjacency list for node $u$. For adjacency matrices, the weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored as the entry in row $u$ and column $v$ of the adjacency matrix. If an edge does not exist, a NIL value can be stored as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or $\infty$.

---

[^3]: $|E| \ll |V|^2$

[^4]: Memory intensive at times!
Figure 6: Representing a directed graph.
2.4 Generating Random Graphs

2.4.1 Introduction

A random graph is a graph in which properties such as the number of graph vertices, graph edges, and connections between them are determined in some random way. There are many models to be studied. We will focus on one such model: Erdős–Rényi (ER) random graph model. This model has two closely related models:

- \( G(n, m) \): a graph is chosen uniformly at random from the collection of all graphs which have \( n \) nodes and \( M \) edges
- \( G(n, p) \): a graph is constructed by connecting nodes randomly. Each edge is included in the graph with probability \( p \) independent from every other edge. Equivalently, all graphs with \( n \) nodes and \( M \) edges have equal probability of \( p^M (1 - p)^{(\binom{n}{2}) - M} \)

We work with the second model (due to Gilbert).

2.4.2 Procedure

We are familiar with the adjacency matrix representation of a matrix. Given an adjacency matrix, we can obtain the corresponding graph. We focus on generating the adjacency matrix of a random graph: generating a random matrix with certain properties.

- For digraph using \( G(n, p) \) model, we have a square matrix of order \( n \), with a null diagonal.
- For unweighted graphs, the entries are independent \( Bin(1, p) \) random variables.
- For weighted graphs, each entry is a \( Bin(1, p) \) random variable, with weight \( W \) following some given distribution. We will stick to \( W \sim U[0, 1] \). Weights need not necessarily be positive, as discussed before.
- For undirected graphs, we need only consider an upper (or lower triangular) matrix with null diagonal. We will later construct a symmetric adjacency matrix using this lower (upper) triangular matrix.
- The weight assignment procedure remains the same.

We will devote special attention to generating directed acyclic graphs or DAGs. The procedure on the last slide does nothing to ensure if a graph is acyclic or not. We remedy this by maintaining the following convention while generating a DAG:

The idea is that the graph is acyclic if and only if if there exists a vertex ordering which makes the adjacency matrix lower triangular. It's easy to see that if the adjacency matrix is lower triangular, then vertex \( i \) can only be pointing to vertex \( j \) if \( i < j \).

We borrow this idea to generate a lower triangular random matrix (with appropriate entries) and use it to get our graph. Once this is done, we may choose to permute the tags (numbers/names) of the nodes.
2.5 Shortest Path Problem

2.5.1 Statement of Problem.

In a shortest-paths problem, we are given a weighted, directed graph

\[ G = (V, E) \]

with weight function

\[ w : \rightarrow \mathbb{R} \]

The weight of path \( p = \{v_0, v_1, ..., v_k\} \) is:

\[ w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \]

The shortest-path weight from \( u \) to \( v \) is

\[ \delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} : \text{if there is path from } u \text{ to } v. \\ \infty : \text{otherwise.} \end{cases} \]

A shortest path from vertex \( u \) to vertex \( v \) is then defined as any path \( p \) with weight \( w(p) = \delta(u, v) \)

2.5.2 Variants

Our aim is to find shortest path in a given graph. First we shall focus on the single source shortest path problem: given a graph \( G = (V, E) \), we want to find a shortest path from a given source vertex \( s \in V \) to each vertex \( v \in V \). Variants of this problem might include the following

- **Single destination shortest paths problem:** find a shortest path to a given destination vertex \( t \) from each vertex \( v \). By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

- **Single pair shortest path problem:** find a shortest path from \( u \) to \( v \) for given vertices \( u \) and \( v \).

- **All pairs shortest path problem:** find a shortest path from \( u \) to \( v \) for every pair of vertices \( u \) and \( v \). Although this can be solved by running a single source algorithm once from each vertex, it can usually be solved faster.
2.5.3 Discussion on Shortest Paths

1. **Optimal Substructure of Shortest Path:** Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it. This optimal substructure property is a hallmark of the applicability of both dynamic programming and the greedy method.

   - **Dijkstra’s Algorithm** is a greedy algorithm.
   - **Floyd Marshall Algorithm** is a dynamic programming algorithm.

The following lemma states the optimal-substructure property of shortest paths more precisely:

Given a weighted, directed graph $G = (V, E)$ with weight function $w : \rightarrow \mathbb{R}$ let $p = \{v_1, v_2, \ldots, v_k\}$ be a shortest path from vertex $v_1 \rightarrow v_k$ and, for any $i, j$ such that $1 \leq i \leq j \leq k$, let $p_{ij} = \{v_i, v_{i+1}, \ldots, v_j\}$ be the subpath of $p$ from vertex $v_i \rightarrow v_j$. Then, $p_{ij}$ is a shortest path from $v_i \rightarrow v_j$.

2. **Presence of negative weight edges:** Sometimes, there may be edges whose weights are negative. If the graph $G = (V, E)$ contains no negative-weight cycles reachable from the sources, then for $v \in V$, $\delta(s, v)$ remains well defined, even if it has a negative value.

   If there is a negative-weight cycle reachable from $s$, however, shortest-path weights are not well defined. No path from $s$ to a vertex on the cycle can be a shortest path—a lesser-weight path can always be found that follows the proposed “shortest” path and then traverses the negative-weight cycle. **If there is a negative-weight cycle on some path from $s$ to $v$, we define $\delta(s, v) = -\infty$.**

3. **Presence of cycles:** We will establish: a graph cannot contain a negative-weight cycle. Nor can it contain a positive weight cycle.

   If $p = \{v_0, v_1, \ldots, v_k\}$ is a path and $c = \{v_i, v_{i+1}, \ldots, v_j\}$ is a positive weight cycle on this path (so that $v_i = v_j$ and $w(c) > 0$), then the $p' = \{v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k\}$ has weight $w(p') = w(p) - w(c) < w(p)$, and so $p$ cannot be a shortest path from $v_0 \rightarrow v_k$.

That leaves only 0-weight cycles. We can remove a 0-weight cycle from any path to produce another path whose weight is the same. As long as a shortest path has 0-weight cycles, we can repeatedly remove these cycles from the path until we have a shortest path that is cycle-free.
2.5.4 Some important routines.

- **InitialiseSingleSource(G,s):** given a graph \( G \) and a source node \( s \), this procedure initialises two more data structures: \( d[u] \) to calculate distance of node \( u \in V \) from \( s \) and \( \pi[u] \) to obtain predecessor of each node \( u \in V \).

  ```
  1: function InitialiseSingleSource(G,s)
  2:     for each \( v \in V \) do
  3:         \( d[v] \leftarrow \infty \)
  4:         \( \pi[v] \leftarrow \text{NIL} \)
  5:     end for
  6:     \( d[u] \leftarrow 0 \)
  7: end function
  ```

- **Relax(G,u,v):** tests whether we can improve the shortest path to \( v \) found so far by going through \( u \) and, if so, updating \( d[v] \) and \( \pi[v] \). A relaxation step may decrease the value of the shortest path estimate \( d[v] \) and update \( v \)'s predecessor field \( \pi[v] \).

  ```
  1: function Relax(G, u, v)
  2:     if \( d[u] > d[v] + w(u, v) \) then
  3:         \( d[u] \leftarrow d[v] + w(u, v) \)
  4:         \( \pi[v] \leftarrow u \)
  5:     end if
  6: end function
  ```

2.5.5 Some important properties:

1. **Triangle inequality:** For any edge \((u,v)\in E\), we have \( \delta(s,v) \leq \delta(s,u) + w(u,v) \).

2. **Upper-bound property:** \( d[v] \geq \delta(s,v) \) for all vertices \( v \in V \), and once \( d[v] \) achieves the value \( \delta(s,v) \), it never changes.

3. **No-path property:** If there is no path \( s \rightarrow v \), \( d[v] = \delta(s,v) = \infty \).

4. **Convergence property:** If \( s \rightsquigarrow u \rightarrow v \) is a shortest path in \( G \) for some \( u,v \in V \), and if \( d[u] = \delta(s,u) \) at any time prior to relaxing edge \((u,v)\), then \( d[v] = \delta(s,v) \) at all times afterward.

5. **Path-relaxation property:** If \( p = \{v_0,v_1,\ldots,v_k\} \) is a shortest path from \( v_0 \rightarrow v_k \), and the edges of \( p \) are relaxed in the order \((v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)\), then \( d[v_k] = \delta(s,v_k) \).

6. **Predecessor subgraph property:** Once \( d[v] = \delta(s,v) \) for all \( v \in V \), the predecessor subgraph is a shortest paths tree rooted at \( s \).

Proof of these lemmas may be found in Secn. 24.5, *Cormen et al (2009)*, pp 607 onwards.
2.6 Breadth First Search (BFS)

2.6.1 Introduction

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. As we shall see later, Dijkstra’s single-source shortest-paths algorithm use ideas similar to those in breadth-first search.

2.6.2 Problem statement

Given a graph \( G = (V,E) \) and a distinguished source vertex \( s \), BFS systematically explores the edges of \( G \) to “discover” every vertex that is reachable from \( s \).

2.6.3 Remarks

- It computes the distance (in terms of the smallest number of edges) from \( s \) to each reachable vertex.
- It also produces a “breadth-first tree” with roots that contains all reachable vertices.
- For any vertex \( v \) reachable from \( s \), the path in the breadth-first tree from \( s \) to \( v \) corresponds to a “shortest path” from \( s \) to \( v \) in \( G \), that is, a path containing the smallest number of edges.
- The algorithm works on both directed and undirected graphs.
- We talk about BFS on unweighted graphs only.
- It expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- BFS discovers all vertices at distance \( k \) from \( s \) before discovering any vertices at distance \( k + 1 \).

2.6.4 Description of algorithm.

To keep track of progress, BFS colors each vertex white, grey, or black.

\[
WHITE \rightarrow GREY \rightarrow BLACK
\]

A vertex is discovered the first time it is encountered during the search, at which time it becomes non white.

Grey and black vertices, therefore, have been discovered, but BFS distinguishes between them to ensure that the search proceeds in a breadth-first manner.

If \( (u,v) \in E \) and vertex \( u \) is black, then vertex \( v \) is either grey or black; that is, all vertices adjacent to black vertices have been discovered. Grey vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

For completeness, the BFS algorithm has been included as a part of Appendix A.
2.6.5 Working of algorithm.

In addition to the graph \( G \) itself, BFS requires three additional data structures:

- The colour of each vertex \( u \in V \) is stored in the variable \( \text{colour}[u] \)
- The predecessor of \( u \) is stored in the variable \( \pi[u] \). If \( u \) has no predecessor (for example, if \( u = s \) or \( u \) has not been discovered), then \( \pi[u] = \text{NIL} \).
- The distance from the source \( s \) to vertex \( u \) computed by the algorithm is stored in \( d[u] \).

The algorithm also uses a queue - \( Q \) to manage the set of grey vertices.

2.6.6 BFS returns shortest path.

Define: \( \delta(s,v) \) from \( s \) to \( v \) as the minimum number of edges in any path from vertex \( s \) to vertex \( v \). If there is no path from \( s \) to \( v \), then \( \delta(s,v) = \infty \).

We require the following results:

1: For any edge \((u,v) \in E\), \( \delta(s,v) \leq \delta(s,u) + 1 \)

2: Upon termination, for each vertex \( v \in V \), the value \( d[v] \) computed by BFS satisfies \( d[v] \geq \delta(s,v) \)

3: During the execution of BFS on a graph \( G = (V,E) \), the queue \( Q \) contains the vertices \( \{v_1,v_2,...,v_r\} \), where \( v_1 \) is the head of \( Q \) and \( v_r \) is the tail. Then, \( d[v_r] \leq d[v_1] + 1 \) and \( d[v_i] \leq d[v_{i+1}] \) for \( i = 1,2,...,r-1 \).

4: Suppose that vertices \( v_i \) and \( v_j \) are enqueued during the execution of BFS, and that \( v_i \) is enqueued before \( v_j \). Then \( d[v_i] \leq d[v_j] \) at the time that \( v_j \) is enqueued.

\( \infty \): Finally, using these results, we will now demonstrate correctness of the BFS algorithm, and justify our claim of BFS yielding shortest path (in terms of number of edges.)

\[ \delta[s,v] = d[v] \]

Details may be found in Chapter 22, Cormen et al. (2009) pp 535 onwards.
2.7 Bellman Ford Algorithm

2.7.1 Introduction

The Bellman-Ford algorithm solves the single source shortest paths problem when edge weights may be negative.

Given a weighted, digraph \( G = (V, E) \) with source \( s \) and weight function \( w : \rightarrow \mathbb{R} \), the algorithm returns a boolean value indicating if there is no negative-weight cycle that is reachable from the source.

- If FALSE, the algorithm indicates that no solution exists.
- If TRUE, the algorithm produces the shortest paths and their weights.

The algorithm uses \( \text{Relax}(G, u, v) \), progressively decreasing an estimate \( d[v] \) on the weight of a shortest path from the source \( s \) to each vertex \( v \in V \) until it achieves the actual shortest-path weight \( \delta(s, v) \).

2.7.2 Algorithm

```plaintext
1: function BELLMANFORD(G, s) 
2:    InitialiseSingleSource(G, s) 
3:    for i ← 1 to \( |V| - 1 \) do 
4:       for edge \((u, v) \in E(G)\) do 
5:          Relax(u, v) 
6:       end for 
7:    end for 
8:    for each edge in \( E(G) \) do 
9:       if \( d[v] > d[v] + w(u, v) \) then 
10:          return FALSE 
11:    end if 
12: end for 
13: return TRUE 
14: end function
```

Steps:
1. Initializing the \( d \) and \( \pi \) values for each node \( v \in V \).
2. Make \( |V(G)| - 1 \) passes over the edges of the graph. Each pass is one iteration of the for loop of lines 4-6 and consists of relaxing each edge of the graph.
3. Check for a negative weight cycle and return the appropriate boolean value in lines 8-13.

First, we will see how the algorithm works in absence of negative weight cycles. Let \( G = (V, E) \) be a weighted, digraph with source \( s \) and weight function \( w : \rightarrow \mathbb{R} \), and assume that \( G \) contains no negative weight cycles that are reachable from \( s \). Then, after the \( |V(G)| - 1 \) iterations of the for loop of lines 4-6 of the algorithm, we have \( d[v] = \delta(s, v) \) for all vertices \( v \) that are reachable from \( s \).

We still have to take care of the negative cycles: we shall prove that if \( G \) does contain a negative-weight cycle reachable from \( s \), then the algorithm returns FALSE.

Proof of correctness of Bellman Ford algorithm is discussed in detail in Secn. 24.1, Cormen (2009), pg 588 onward.

As described in the algorithm makes \( |E| \) relaxations for every iteration, and there are \( |V| - 1 \) iterations. The worst-case scenario is that Bellman-Ford runs in time \( O(|V| \cdot |E|) \). For certain graphs, only one iteration is needed, and so in the best case scenario, only \( O(|E|) \) time is needed.

16
2.7.3 Application: Arbitrage opportunities

Finally, we will discuss an application of the Bellman Ford algorithm: **arbitrage opportunity**. It is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency.

For example, suppose that 1 U.S. dollar buys 46.4 Indian rupees, 1 Indian rupee buys 2.5 Japanese yen, and 1 Japanese yen buys 0.0091 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $46.4 \times 2.5 \times 0.0091 = 1.0556$ U.S. dollars, thus turning a profit of 5.56 percent.

Suppose that we are given $n$ currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table $R$ of exchange rates, such that one unit of currency $c_i$ buys $R[i,j]$ units of currency $c_j$.

Our aim would be to find an algorithm to determine whether or not there exists a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

By taking negative of logarithm of $R[i, j]$ in the expression above, we observe equivalence between the arbitrage problem and detection of negative weight cycles in a given graph.
2.8 Shortest Paths in DAGs

2.8.1 Introduction

We have learnt how to tackle shortest path problems given a weighted digraph using Bellman Ford algorithm. This algorithm has time complexity given by $O(|V| \cdot |E|)$. However, if we are further told that the graph is acyclic, we can employ a more efficient algorithm which has time complexity given by $O(|E| + |V|)$, i.e. a linear time algorithm for computing shortest paths from a single source in a directed acyclic graph.

Before we discuss the algorithm, we need to introduce the concept of topological sorting in a digraph.

Topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge $(u, v)$ from $u \rightarrow v$, $u$ comes before $v$ in the ordering.

We will not concern ourselves with how topological sorting is implemented\footnote{Topological sorting is closely associated with depth first search (DFS).}. We assume we are equipped with a method $\text{topoSort}(G)$ to do the job for us. We also know that it has time complexity $O(|E| + |V|)$.

2.8.2 Algorithm

```plaintext
1: function DagShortestPath(G, s)
2:     topoSort(G)
3:     InitialiseSingleSource(G, s)
4:     for each $u \in V$ in sorted order do
5:         for each $v$ adjacent to $u$ do
6:             Relax(G, u, v)
7:         end for
8:     end for
9: end function
```

Steps:
1. The algorithm starts by topologically sorting the DAG to impose a linear ordering on the vertices in line 2.
2. Makes just one pass over the vertices in the topologically sorted order (lines 4-9).
3. As we process each vertex, we relax each edge incident from it in line 6.

To prove the correctness of this algorithm, we will need to prove the following claim.

If a weighted, directed graph $G = (V, E)$ has source vertex $s$ and no cycles, then at the termination of the $\text{DagShortestPath}$ procedure $d[v] = \delta(s, v)$ for all vertices $v \in V$.

The correctness of this algorithm may be established by following the arguments presented in Secn. 24.2, Cormen (2009), pg 593 onward.

Runtime analysis of this algorithm is as follows.

- The $\text{topoSort}(g)$ of line 2 can be performed in $O(|V| + |E|)$ time.
- The call of $\text{InitialiseSingleSource}$ in line 2 takes $O(|V|)$ time.
- There is one iteration per vertex in the for loop of lines 4-8. For each vertex, the edges that leave the vertex are each examined exactly once. Thus, there are a total of $|E|$ iterations of the inner for loop of lines 5-7.
As each iteration of the inner for loop takes \( O(1) \) time, the total running time is \( O(V + E) \), which is linear in the size of an adjacency-list representation of the graph.

2.8.3 Application: PERT analysis.

Here we will discuss an application: **PERT analysis**. PERT is an acronym for program evaluation and review technique. A PERT chart presents a graphic illustration of a project as a network diagram consisting of numbered nodes representing events, or milestones in the project linked by labelled directed edges representing tasks in the project.

The direction of the arrows on the lines indicates the sequence of tasks. A path through this DAG represents a sequence of jobs that must be performed in a particular order. A critical path is a longest path through the dag, corresponding to the longest time to perform an ordered sequence of jobs.

The weight of a critical path is a lower bound on the total time to perform all the jobs. We can find a critical path by negating the edge weights and running \texttt{DagShortestPath}. 

2.9 Dijkstra’s Algorithm

2.9.1 Introduction

Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V,E)$ for the case in which all edge weights are non-negative. In this section, therefore, we assume that $w(u,v) \geq 0$ for each edge $(u,v) \in E$.

The algorithm maintains a set $S$ of vertices whose final shortest path weights from the source $s$ have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest path estimate, adds $u$ to $S$, and relaxes all edges leaving $u$. In the following implementation, we use a minimum priority queue $Q$ of vertices, keyed by their $d$ values.

1: function Dijkstra(G,s)
2: InitialiseSingleSource(G, s)
3: $S \leftarrow 0$
4: $Q \leftarrow V[G]$
5: while $Q$ not EMPTY do
6: $u \leftarrow \text{ExtractMin}(Q)$
7: $S \leftarrow S \cup u$
8: for $v$ adjacent to $u$ do
9: Relax(G, u, v)
10: end for
11: end while
12: end function

Steps:
1. Perform the usual initialization of $d$ and values in line 2.
2. Line 3 initializes the set $S$ to the empty set.
3. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the while loop of lines 5-11.
4. Line 4 initialises the min-priority queue $Q$ to contain all the vertices in $V$; since $S = \emptyset$ at that time, the invariant is true after line 3.
5. Each time through the while loop of lines 5-11, a vertex $u$ is extracted from $Q = VS$ and added to set $S$, thereby maintaining the invariant. (The first time through this loop, $u = s$.)
6. Vertex $u$, therefore, has the smallest shortest path estimate of any vertex in $VS$.
7. Then, lines 8-10 relax each edge $(u,v)$ leaving $u$, thus updating the estimate $d[v]$ and the predecessor $\pi[v]$ if the shortest path to $v$ can be improved by going through $u$.

Observe that vertices are never inserted into $Q$ after line 4 and that each vertex is extracted from $Q$ and added to $S$ exactly once, so that the while loop of lines 5-11 iterates exactly $|V(G)|$ times.

Before we go into the correctness of the algorithm, we should note that Dijkstra’s algorithm always chooses the lightest or closest vertex in $V - S$ to add to set $S$, we say that it uses a greedy strategy. Greedy strategies do not always yield optimal results in general, but as the following theorem and its corollary show, Dijkstra’s algorithm does indeed compute shortest paths

We will be trying to prove that the algorithm, when run on a weighted digraph $G$ with non-negative weights, terminates with $d[u] = \delta(s,u)$ for all vertices $u \in V$.

The key to proving this is to notice that each time a vertex $u$ is added to set $S$, we have $d[u] = \delta(s,u)$. To study the correctness of the algorithm, we will follow the proof presented in Secn. 24.3, Cormen (2009), pg 597 onward.
Bounds of the running time of Dijkstra’s algorithm on a graph with edges E and vertices V can be expressed as a function of $|E(G)|$ and $|V(G)|$ using big-O notation, as we have done this far. How tight a bound is possible depends on the way the vertex set Q is implemented.

Three priority-queue operations are called: INSERT (line 4), EXTRACT-MIN (line 6), and DECREASE-KEY (implicit in RELAX, line 9). INSERT is invoked once per vertex, as is EXTRACT-MIN. There are at most $|E|$ DECREASE-KEY operations.

For any implementation of the vertex set Q, the running time is in

$$O(|E| \cdot T_{dk} + |V| \cdot T_{em})$$

$T_{dk}$ and $T_{em}$ are the complexities of the DECREASE-KEY and EXTRACT-MIN operations in Q. INSERT in Q is a $O(1)$ complexity task. The simplest implementation of Dijkstra’s algorithm stores the vertex set Q as an ordinary linked list or array, and extract-minimum is simply a linear search through all vertices in Q. In this case, the running time is

$$O(|E| + |V|^2) = O(|V|^2)$$

2.9.2 Application: Erdős numbers.

We will finally look at an application on Dijkstra’s algorithm: Erdős numbers. The Erdős number describes the ”collaborative distance” between mathematician Paul Erdős and another person, as measured by authorship of mathematical papers. Given a graph with nodes representing authors of mathematical papers, and an edge between two nodes signifying that the two authors have been co-authors of at least one paper, our problem would be to find the shortest path between a given author and Paul Erdős - which is essentially a single source shortest path problem. We will implement Dijkstra’s algorithm on a collaborative graph generated from a dataset found online.
2.10 All Pairs Shortest Paths Problem

2.10.1 Introduction

Finally, we consider the problem of finding shortest paths between all pairs of vertices in a graph. Typically, we will be given a weighted digraph $G = (V, E)$ with a weight function $w : \rightarrow \mathbb{R}$ that maps edges to real valued weights. We wish to find, for every pair of vertices $u, v \in V$, a least-weight path from $u$ to $v$. We typically want the output in tabular form: the entry in $u$’s row and $v$’s column should be the weight of a shortest path from $u$ to $v$.

2.10.2 Application of what we have learnt

We can solve an all-pairs shortest-paths problem by running a single source shortest paths algorithm $|V|$ times, once for each vertex as the source. Then, run time would be:

- Using Dijkstra’s algorithm: if we use the linear-array implementation of the min-priority queue, the running time is
  \[ O(V^3 + V \cdot E) = O(V^3) \]
  Alternatively, for sparse graphs, we can implement the min-priority queue with a Fibonacci heap, yielding a running time of
  \[ O(V^2 \cdot \log(V) + V \cdot E) \]

- If negative edge weights are allowed, we have to use the slower Bellman Ford’s algorithm, thereby increasing the running time to
  \[ O(V^2 \cdot E) \]
  which on a dense graph is $O(V^4)$

2.10.3 Topics for further study.

We could choose to do the following instead.

- Implement a dynamic programming algorithm based on matrix multiplication to solve the all-pairs shortest-paths problem in $O(V^3 \cdot \log(v))$. (Reference: Section 25.1, Cormen et. al. (2001), pp 622.)

- If no negative weight cycles are present, we can look at another dynamic programming algorithm, the Floyd Marshall algorithm, which has run time of $O(V^3)$. (Reference: Section 25.2, Cormen et. al. (2001), pp 629.)

- For sparse graphs, we can study about Johnson’s algorithm which has a run time of $O(V^2 \cdot \log(V) + V \cdot E)$, which outperforms the previous two algorithms when implemented on sparse graphs. (Reference: Section 25.2, Cormen et. al. (2001), pp 636.)
3 References


- The collaborative graph for calculating Erdos number has been generated from the dataset sourced from http://vlado.fmf.uni-lj.si/pub/networks/data/Erdos/Erdos02.net

4 Appendix

Appendix A: Algorithm for BFS is as follows.

```plaintext
BFS(G, s)
1   for each vertex u ∈ V[G] − {s}
2       do color[u] ← WHITE
3       d[u] ← ∞
4       π[u] ← NIL
5       color[s] ← GRAY
6       d[s] ← 0
7       π[s] ← NIL
8       Q ← Ø
9       ENQUEUE(Q, s)
10      while Q ≠ Ø
11         do u ← DEQUEUE(Q)
12             for each v ∈ Adj[u]
13                 do if color[v] = WHITE
14                     then color[v] ← GRAY
15                     d[v] ← d[u] + 1
16                     π[v] ← u
17                     ENQUEUE(Q, v)
18         color[u] ← BLACK
```


Appendix B: List of functions used is as follows.

- `plotGraph(g)`: to plot a graph $g$ with numbering of nodes and edge weights (if applicable).
- `makeGraph(n, p)`: to generate a random graph according to Erdos Renyi $G(n,p)$ model.
- `makeDAG(n, p)`: to generate a DAG of size $n$.
- `getPath(g, p, sourceNode, destNode)`: to generate a path using the predecessor array $p$ from `sourceNode` to `destNode`.
- `getWeight(g, path)`: to get weight of a `path` in graph $g$. 

• BellmanFord\((g, \text{sourceNode}, \text{destNode})\) to implement Bellman Ford algorithm on graph \(g\) from \text{sourceNode} to \text{destNode}.

• DagShortestPath\((g, \text{sourceNode}, \text{destNode})\) to find shortest path on DAG \(g\) from \text{sourceNode} to \text{destNode}.

• Dijkstra\((g, \text{sourceNode}, \text{destNode})\) to implement Dijkstra’s algorithm on graph \(g\) from \text{sourceNode} to \text{destNode}.

• A code to generate the collaboration graph to find Erdos number from a given text file containing names of authors and a list of collaborations between them.

The code files may be found \[\text{here}].\]