An Introduction to Graph Algorithms

Cathen Fontanilla and Zoe Guo

May 2026

Abstract

In this paper, we explore the basics of graph theory by providing a short background section on graph theory fundamentals and properties of graphs. We connect these properties to combinatorial algorithms to find special traits, such as the shortest path between two points. In particular, we cover the Gale-Shapley Algorithm, Kruskal's (Greedy) Algorithm, and Dijkstra's Algorithm (along with other algorithms), and discuss what these algorithms look like in practice. Finally, we apply graph algorithms to real-life scenarios, demonstrating how combinatorics can help us gain a better understanding of our world.

1 Introduction

The world around us is filled with graphs - and they're not just the kind used in xy planes or plots for statistics. These graphs are made of vertices and edges, and they are found in GPS navigation, trip planning, and even the job hiring process. To understand graphs and their applications, mathematicians study graph theory, which investigates the special properties of graphs like paths to improve the way graphs are applied in daily life.

In this paper, we will explore a subset of graph theory, graph algorithms, before investigating different applications of graph algorithms in daily life.

We begin in Section 2 with an introduction to graph theory and algorithms, as well as a few important graph-related terms.

In Section 3, we will investigate common graph algorithms and analyze their efficiencies. We will also explore shortest path algorithms and the different scenarios in which each is used.

Finally, we conclude with Section 4, which introduces a mundane and helpful application of the discussed algorithms.

2 Background

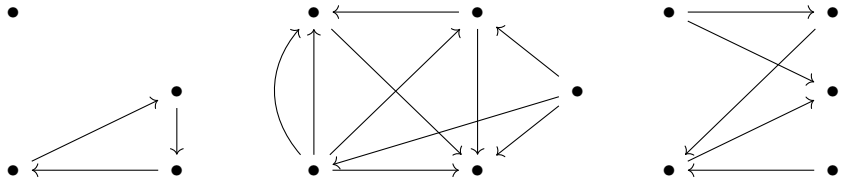
In this section, we discuss some of the most common definitions and properties in graph theory.

2.1 Graph Theory

Many real-world systems involve relationships between discrete objects. For instance, many problems in mathematics and computer science involve studying relationships between a collection of objects. These systems can be represented through graphs. These basic structures allow us to define more complex objects that will be used in later sections, such as paths, cycles, and trees.

Definition 2.1 (Graphs)**.** A *graph*, $G = (V, E)$, is defined as the set of vertices V which are connected via a set of edges E. Edges of a graph may further be denoted by their endpoints through the form $e = \{x, y\}$, which signifies that edge e begins at vertex x and ends at vertex y.

Graphs may come in a variety of shapes and sizes, some of which are demonstrated below.

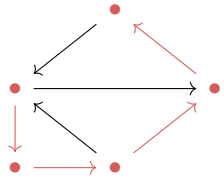


Within graphs, special orderings of vertices and edges have different names, such as paths and cycles. There are also paths with special properties, like trees.

Definition 2.2 (Path)**.** A *path* in a graph $G = (V, E)$ is a sequence of distinct edges

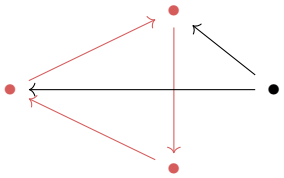
$$(v_0, v_1), (v_1, v_2), \ldots, (v_n, v_{n+1})$$

such that the endpoint of edge (v_x, v_{x+1}) is the starting point of the proceeding edge and $\{(v_i, v_{i+1})\} \in E$ (note that all edges are defined by a pair of vertices). Additionally, each edge in a path only shows up once. The *length* of the path is k, the same number as the number of edges within the path.



A graph with a path of length 4, highlighted in red.

Definition 2.3 (Cycle)**.** A *cycle* in graph G is a series of selected edges within the set E that form a closed loop which starts and ends at the same vertex.

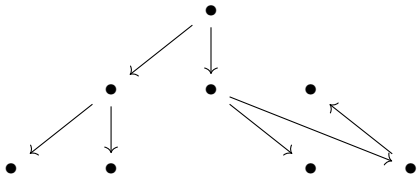


A graph with a cycle, highlighted in red.

Definition 2.4 (Tree)**.** A *tree* is a graph, G that is a connected, simple graph on n vertices with two special qualities:

(i) G is *minimally connected* (remove one edge of G and you get a disconnected graph)

(ii) G has no *cycles*; this also means there is only 1 unique path from any vertex X to a different vertex Y on a tree.



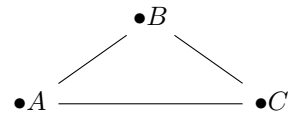
Definition 2.5 (Spanning Tree)**.** A *spanning tree* is a type of specially created tree of a graph G. Let T be a spanning tree if:

(i) G and T are connected graphs.

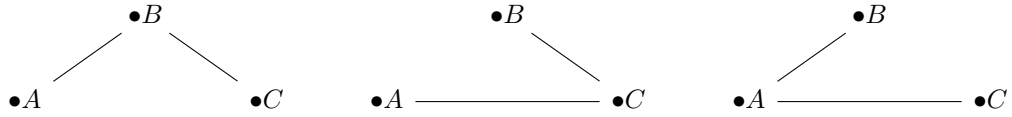
(ii) All vertices of T are vertices of G and vice versa.

(iii) Every edge in T is an edge in G, but not every edge in G is necessarily an edge in T.

Graph G



Spanning Trees of Graph G



Spanning trees of G can be created by taking G and slowly removing edges until there is no edge that can be removed that would leave G as a connected graph. Note that graphs that are trees form their own spanning trees. Spanning trees are important because they connect all vertices in a graph using the fewest possible positive edges while avoiding unnecessary cycles.

2.2 Algorithms and Efficiency

In combinatorics, graphs have a wide variety of applications, including (but not limited to) representing relationships within people or modeling road infrastructure in cities. When graphs represent these structures, some common questions that arise are whether mathematicians can use these structures to find the shortest path between two cities or if all vertices in the graph can be connected efficiently. These questions are tackled using graph algorithms.

Definition 2.6 (Graph Algorithms)**.** A graph algorithm can be defined as a series of steps or functions that are performed on a graph $G = V, E$. For example, one step in an algorithm could be choosing two $v_i \in V$ and adding the smaller v_i into an empty set as a new element. While one step may seem simple, the use of algorithms in place of humans completing these steps helps mathematicians find patterns and analyze graphs really quickly, and with computer programming, two traits that especially help when mathematicians are short on time.

All graph algorithms have a measurement called efficiency. Efficiency is the number of steps or space it takes for an algorithm to finish acting upon a graph based on the algorithm's instructions. In many real-world applications, like computer programming, algorithms that are more efficient are preferred because they will consume less energy or take up less storage.

The efficiency of a graph algorithm is defined using the asymptotic notation $O(f)$ (or $\Omega(f)$ and $\theta(f)$, but we'll explain those notations later). As an algorithm approaches an input of infinity, the efficiency or time complexity of the algorithm approaches f. To compare how efficient algorithms are, mathematicians use asymptotic notation, which describes how the running time of an algorithm grows as the input size becomes increasingly large. It can be measured in three different ways: $O(f)$, $\Omega(f)$, and $\theta(f)$. In our paper, we will be focusing on $O(n)$ only.

Note: The following are explained as "the algorithm with formula F belongs to the set of functions that grows no faster than $O(g(f))$."

Definition 2.7 ($O(n)$)**.** The notation $O(n)$, known as the time complexity, for some algorithm $f(n)$ represents a function that approximates the maximum number of steps the algorithm will take as the function $f(n)$ increases for large input size for n.

For example, if an algorithm takes

$$f(n) = n^3 + 2n^2 + 57$$

steps to run, then we say its time complexity is

$$O(n^3),$$

since $f(n)$ approaches n^3 for large values of n. This is because the highest-degree term dominates the growth of the function as n approaches infinity.

Example 2.8 (Algorithm Efficiency with the Largest Number)**.** Suppose we are given a list of n numbers and want to find the largest value. One simple algorithm checks each number in the list one at a time and keeps track of the current maximum. Since this algorithm examines each element once, the algorithm has a running time of approximately n steps, giving it a time complexity of $O(n)$.

Example 2.9 (Algorithm Efficiency with MergeSort)**.** MergeSort $M(n)$ is an algorithm that takes a list of numbers, $\{n_1, n_2, n_3, ...n_m\}$ as an input and outputs the same list, but with the numbers sorted from least to greatest. MergeSort works by taking a set of x elements, splitting that set in half (or as close to half as possible), and continuing to split subsets until each subset contains 2 or 1 elements inside it. Then, MergeSort will sort those small-element subsets before recombining them to create a fully-sorted set of n. To evaluate the efficiency of MergeSort we note:

$$M(2k) = 2M(k) + 2k - 1.$$

In other words, the amount of steps to sort a list of $2k$ elements via MergeSort is the same as the number of steps to split 2 sets of k elements (because $2k$ will be split into subsets of size k), and there are a max of $2k - 1$ steps to combine the two lists.

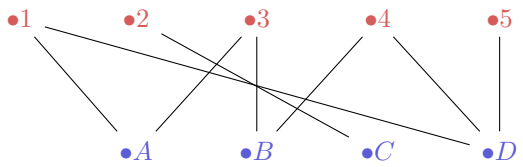
3 Main Algorithms

Many common algorithms involve shortest paths, most efficient connected paths, and so forth. As such, we have composed a list of the most common combinatorics algorithms and their efficiencies below.

3.1 Gale-Shapely Algorithm

One important problem in graph theory and combinatorics is determining stable matchings between two groups with preferences.

Definition 3.1 (Bipartite Graph)**.** A bipartite graph is a graph whose vertices can be divided into two disjoint sets, X and Y, such that every edge connects a vertex in X to a vertex in Y. No two vertices in the same set are adjacent. Vertices in X are only connected to vertices in Y and not in X while vertices in Y are only connected to vertices in X and not in Y.

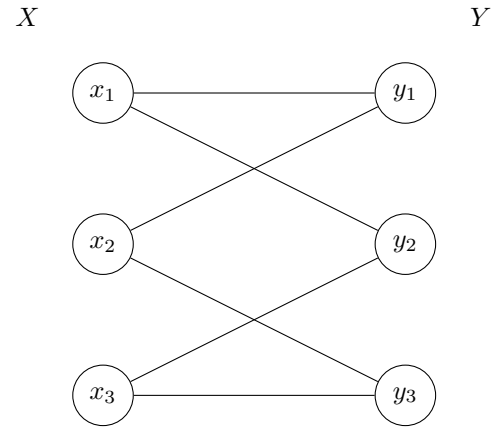


Definition 3.2. A matching is called *stable* if there does not exist an applicant and a job that would both prefer each other over their current matches.

Definition 3.3 (Gale-Shapley Algorithm)**.** The Gale-Shapley algorithm, also known as the deferred acceptance algorithm, is a method used to create a stable matching between two groups with preferences.

Let $G(X, Y)$ be a bipartite graph, where:

- $X = \{x_1, x_2, x_3, \dots\}$ represents a set of applicants,
- $Y = \{y_1, y_2, y_3, \dots\}$ represents a set of jobs.



Here, we find a stable matching of some group of vertices or "applicants" with a second group of vertices or "open jobs." In this algorithm, we assume that each object in our "applicant" group has a ranked list of "open jobs" they would prefer to be matched with (e.g., job #1, job #2, job #3, etc.). At the same time, each "open job" has a ranked list of preferred "applicants" they would like to fill that job. To view this scenario in terms of graph theory, let us establish the following:

For G, our algorithm will turn up a matching, M, for which each $x \in X$ may not get their preferred $y \in Y$ (and vice versa), but no two x would switch y based on preferences.

Algorithm (Gale-Shapely Algorithm).

(i) Each applicant, $\{x_1, x_2, x_3, ...\}$, applied to the first choice job that has not yet rejected them.

(ii) On the other end, each "job" will reserve the first candidate who applies and reject other candidates. If two candidates x_a and x_b apply for the same job, the job $(y_1, y_2, y_3, ...)$ will choose the candidate that they prefer.

(iii) If a candidate is rejected from their first choice job, they will apply to their second choice, then their third choice, and so on and so forth.

(iv) The process continues until every applicant is matched.

The resulting matching is called *stable*, meaning there's no applicant and job that would both prefer each other over their assigned match.

Example 3.4. Suppose there are two students, A and B , and two schools X and Y. The students have the following preferences:

$$A : X \succ Y$$

$$B : Y \succ X$$

The schools have the following preferences:

$$X : A \succ B$$

$$Y : B \succ A$$

First, A applies to X and B applies to Y. Since both schools accept their preferred applicant, the algorithm ends with the stable matching

$$(A, X), (B, Y).$$

Proof of Gale-Shapley algorithm. We first show that the algorithm terminates. This is easy as each applicant applies to jobs in the order of their preference list, and no applicant applies to the same job more than once. Since there are finitely many applications and jobs, only finitely many applications can happen. Thus, the algorithm must eventually come to a stop. Next, we show that the resulting matching is stable. Suppose, for contradiction, that there exists an applicant x and a job y such that:

(i) x prefers y over their assigned math, and

(ii) y prefers x over the applicant assigned to y.

Since x prefers y over their final match, applicant x must have applied to y earlier in the algorithm. When x applied, either:

- y rejected x immediately in favor of a more preferred applicant, or
- y temporarily accepted x but later replaced x with another applicant it preferred more.

In either case, the final applicant matched with y must be preferred over x. This contradicts the initial assumption that y prefers x over its assigned match. Therefore, no such pair (x, y) exists, and the matching produced by the Gale-Shapley algorithm is stable. □

3.2 Spanning Trees

3.2.1 Combinatorics of Spanning Trees

Looking back at our definition of spanning trees in section 2, a natural question one might have is determining how many distinct spanning trees a graph can have. Especially since spanning trees can represent the least amount of roads one needs to build to connect V cities, or networks to connect V servers, knowing the number of possible spanning trees is very useful. However, even for small-sized graphs, the number of unique spanning trees can be surprisingly large.

Counting spanning trees helps determine how many different ways a network can remain connected while minimizing the number of unnecessary edges.

Theorem 3.5 (Cayley's Formula)**.** *The number of spanning trees in the complete graph K_n is n^{n-2}.*

Cayley's formula gives an exact count of spanning trees for a graph in which every pair of vertices is connected by an edge. Despite the simplicity of K_n, the number of spanning trees grows rapidly with n, showing the combinatorial complexity of graph structures.

Proof of Cayley's formula. Consider a labeled tree on the vertices

$$\{1, 2, \ldots, n\}.$$

A Prüfer sequence is formed by repeatedly removing leaves from a tree, where a leaf is a vertex connected to exactly one other vertex. We construct a Prüfer sequence from the tree using the following steps:

(i) Find the leaf with the smallest-numbered vertex.

(ii) Record the label of the vertex adjacent to that leaf.

(iii) Remove the leaf from the tree.

(iv) Repeat until only two vertices remain.

Since one vertex is removed at each step and the process stops when two vertices are left, the resulting Prüfer sequence has length $n - 2$. Conversely, given any sequence with length $n - 2$, whose entries come from

$$\{1, 2, \ldots, n\},$$

one can reconstruct a labeled tree uniquely. Thus, counting spanning trees of K_n is equivalent to counting Prüfer sequences. Each of the $n - 2$ positions in the sequence can contain any of the n vertex labels. Therefore, the total number of Prüfer sequences is

$$n \cdot n \cdots n = n^{n-2}.$$

Since there is a one-to-one correspondence or a bijection between Prüfer sequences andlabeled trees, the number of spanning trees in K_n is therefore,

$$n^{n-2}.$$

□

3.2.2 Minimal Weight Spanning Tree Algorithm

Definition 3.6 (Kruskal's Algorithm)**.** Kruskal's algorithm finds the minimum weighted spanning tree of a weighted, undirected graph.

Kruskal's algorithm is a part of a category of algorithms called "Greedy Algorithms," meaning it attempts to be more efficient by greedily selecting lower weights first. To create Kruskal's algorithm, which finds the minimally weighted spanning tree of G, first select the edge in G with the smallest weighting and store it in set T. Then edge with the second smallest weighting and store it in T, then the edge with third smallest weighting and store it in T while making sure no subset of selected edges in T form a cycle.

For the ith step, look for an unselected edge e_i such that if e_i is added to T, the obtained graph $G_T = (V, T)$ does not contain a cycle and the weight of e_i is minimal among all edges that satisfy our previous two conditions. In other words:

(i) Start with an empty set of edges T.

(ii) Repeatedly choose the edge with the smallest weight that does not form a cycle in T.

(iii) Add the chosen edge to T.

(iv) Stop when T has $|V| - 1$ edges.

(v) The resulting graph T is a minimum spanning tree

Proof of Kruskal's algorithm. Let A be the set of edges chosen so far. At each step, Kruskal's algorithm selects the minimum-weight edge that does not form a cycle with A.

Since A is always a forest, adding an edge e connects two different connected components of A. Therefore, e joins two distinct components (it connects two vertices that are currently disconnected in the forest).

Thus, e is the lightest edge that connects these two components.

Let T^* be a minimum spanning tree. If $e \in T^*$, we are done. If not, adding e to T^* creates a cycle. This cycle must contain some edge e' that connects the same two components. Since e is the smallest such edge, $w(e) \leq w(e')$. Replace e' with e in T^*. The result is still a spanning tree with no greater total weight. Thus, there exists a minimum spanning tree containing e.

Repeating this argument for every step shows that Kruskal's algorithm does produce a minimum spanning tree. □

3.3 Shortest Path Algorithms

Shortest path algorithms are a special subset of graph algorithms that find the shortest, continuous path in weighted and unweighted graphs. For weighted graphs, paths are ranked based on the sum of the edges that make up the path; for unweighted graphs, however, paths are compared by the number of edges make up that path.

In this section, we will be looking into three shortest path algorithms: Breadth First Search, Dijkstra's Algorithm, and the Bellman-Ford Algorithm. For each algorithm, we will first examine the objective and limitations of the algorithm, breaking down the algorithm process, demonstrating an example of the algorithm in action, before finishing with a short proof and efficiency analysis.

3.3.1 Breadth First Search

Breadth First Search (BFS) is an algorithm used to explore a graph step by step. It's especially useful for finding the shortest path in an *unweighted graph*, where all edges are treated equally. Let $G = (V, E)$ be a directed graph and let $s \in V$ be a starting vertex. The goal of BFS is to find the shortest distance from s to every other vertex in the graph.

For each vertex $v \in V$, let $d(v)$ represent the number of edges in the shortest path from s to v. If v cannot be be reached, then $d(v) = \infty$.

Definition 3.7. BSF works by exploring the graph in layers. Firstly, it visits all vertices directly connected to the starting point. Then, it visits vertices that are two steps away, then three steps away, and so on. The BSF process can be broken down into the following steps:

Algorithm (Breadth First Search).

(i) Set $d(s) = 0$ and $d(v) = \infty$ for all $v \neq s$

(ii) Create an empty queue, denoted Q

(iii) Add s to Q

(iv) While Q isn't empty:

  (a) Remove a vertex u from the front of Q

  (b) For each neighbor v of u:

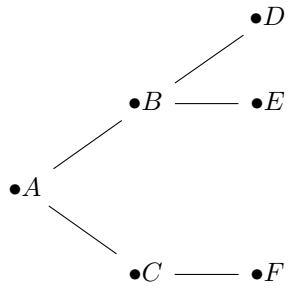
    i. If $d(v) = \infty$

      A. Set $d(v) = d(u) + 1$

      B. Add v to Q

Example. Breadth First Search Consider the graph with vertices $V = \{A, B, C, D, E, F\}$ and edges:

$$E = \{(A, B), (A, C), (B, D), (B, E), (C, F)\}.$$



We demonstrate BFS applied to G starting from vertex A.

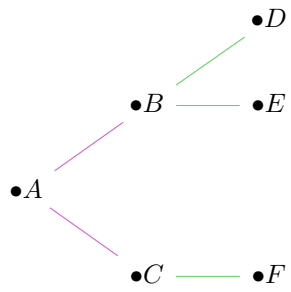
(i) To start, set $d(A) = 0$ and all other distances to ∞.
Queue: $[A]$

(ii) Next, let us process A. The neighbors of vertex A are B and C.
Set $d(B) = 1$, $d(C) = 1$, and add them to the queue.
Queue: $[A, B, C]$

(iii) Moving onwards, we now process B. The neighbors of vertex B are A, D, and E. However, since A is already visited, so we skip it.
Set $d(D) = 2$, $d(E) = 2$. and add them to the queue.
Queue: $[A, B, C, D, E]$

(iv) Then, we process C. The neighbor of C is F.
Set $d(F) = 2$ and add it to the queue.
Queue: $[D, E, F]$

(v) Finally, we process D, E, and F. These vertices do not lead to any new unvisited vertices.



Final distances:

$$d(A) = 0, \quad d(B) = 1, \quad d(C) = 1, \quad d(D) = 2, \quad d(E) = 2, \quad d(F) = 2.$$

This example shows how BFS explores the graph level by level. All vertices at distance 1 from A are visited before any at distance 2, guaranteeing that the distances found are the shortest possible.

Proof of Breadth First Search. Because BFS explores vertices in order of distance from the starting point, the first time we reach a vertex, we have found the shortest path to it. In other words, the algorithm will record a vertex by its shortest distance to the starting point before it can record a vertex by a longer distance to the starting point, and by that time, the shortest distance has already been recorded. □

Time Complexity. BFS runs in $O(|V| + |E|)$ time, since each vertex and edge is considered at most once.
Remark. BFS only works for unweighted graphs because it treats every edge as if it were weighted 1. For graphs with different edge weights, different algorithms are needed, such as Dijkstra's algorithm.

3.3.2 Dijkstra's Algorithm

Unlike the Breadth First Search, which applies to unweighted graphs, many applications require handling graphs with weighted edges. When modeling roads between cities or flight plans, for example, distance, travel time and costs may be recorded as edge weights between two destinations. In such cases, the goal is to compute the shortest paths where edges may have different weights or costs.

Definition 3.8 (Length of Path on a Weighted Graph)**.** The length of a path P on a graph $G = (V, E)$, where every edge $e \in E$ is assigned a numerical weight $e_1 = i_1, e_2 = i_2, ...e_n = i_n$ is the sum of i_x for all $e_x \in P$. For example, if $P = \{e_1, e_5, e_6\}$, the length of P is $i_1 + i_5 + i_6$.

Let $G = (V, E)$ be a weighted graph, where $w : E \to \mathbb{R}_{\geq 0}$ assigns a non-negative weight to each edge. Given a source vertex $s \in V$, the shortest path problem is to determine the minimum-weight path from s to every vertex $v \in V$.

Assumption. All edge weights are non-negative.

For each vertex $v \in V$, let $d(v)$ denote the length of the shortest path from s to v. If no such path exists, $d(v) = \infty$.

Algorithm (Dijkstra's Algorithm).

(i) Begin with $d(s) = 0$ and $d(v) = \infty$ for all $v \neq s$

(ii) Let $S = \emptyset$

(iii) While $S \neq V$:

(a) Select $u \notin S$ with minimum $d(u)$

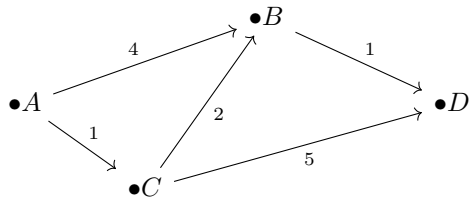
(b) Add u to S

(c) For each edge $(u, v) \in E$, update

$$d(v) = \min(d(v), d(u) + w(u, v))$$

At each step, the algorithm picks the vertex with the smallest known distance from the source and updates the distances of its neighbors. This greedy strategy works because once a vertex is selected, its shortest path distance is finalized.

Example. Consider the weighted graph below with source vertex A.



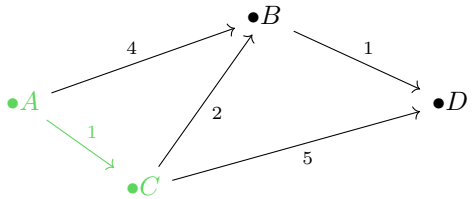
(i) Initially,

$$d(A) = 0, \qquad d(B) = d(C) = d(D) = \infty.$$

The algorithm first selects A and updates:

$$d(B) = 4, \qquad d(C) = 1.$$

(ii)



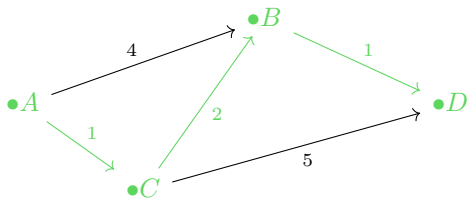
Next, the algorithm selects C since it has the smallest temporary distance. Then, it updates adjacent vertices, giving:

$$d(B) = \min(4, 1 + 2) = 3,$$

and

$$d(D) = \min(\infty, 1 + 5) = 6.$$

(iii)



The algorithm then selects B, updating:

$$d(D) = \min(6, 3 + 1) = 4.$$

(iv) Finally, the algorithm selects D. The shortest distances from A are then:

$$d(A) = 0, \quad d(C) = 1 \quad d(B) = 3, \quad d(D) = 4.$$

Proof of Dijkstra's Algorithm. Whenever a vertex u is added to the set A, the value of $d(u)$ is the guaranteed shortest-path distance from s to u. This is a greedy choice. This works because all edges are positive. If there were negative edges, a shorter path could later appear, violating the greedy choice. We proceed by induction on the number of vertices added to S. Initially, $S = \emptyset$, and the source vertex s is selected first with distance $d(s) = 0$.

Assume that every vertex already in S has the correct shortest-path distance. Let u be the next vertex selected with minimum temporary distance. Suppose, for the sake of contradiction, that there exists a shorter path to u. Such a path must pass through some vertex not yet in S. Since all edge weights are non-negative, any alternative path through an unvisited vertex cannot produce a distance smaller than $d(u)$. Therefore, $d(u)$ is the true shortest-path distance when u is selected. By induction, the algorithm correctly computes the shortest-path distances for all vertices. □

Time Complexity. The time complexity of Dijkstra's Algorithm depends on the data structure used to select the minimum-distance vertex. Using a simple array, selecting the minimum vertex requires $O(|V|)$ time at each step. The total running time, then, is

$$O(|V|^2).$$

Using a priority queue or binary heap, however, improves the running time to

$$O((|V| + |E|) \log |V|),$$

making it more efficient. Thus, Dijkstra's Algorithm is efficient even for relatively complex graphs.

3.3.3 Bellman-Ford Algorithm

Bellman-Ford operates similarly to Dijkstra's algorithm, where it returns a shortest path given a directed, weighted graph. However, the Bellman-Ford algorithm can process negative edge weights, whereas Dijkstra's cannot, due to its greedy choice - if Dijkstra's chooses the least weighted vertex and shortest path at the start, it may miss a longer sequence of edges that include an extremely negative weight and output a shorter path length.

Note, however, that the Bellman-Ford algorithm cannot process graphs with cycles whose edges sum to a negative value (also referred to as a negative-weight cycle or negative cycle). This is because an object traveling a path on the graph could walk endlessly along the negative cycle, forever lowering the value of the path length. Instead of finding the shortest path, in this case, the Bellman-Ford algorithm will flag negative cycles for detection.

To set up the Bellman-Ford algorithm, let $G = (V, E)$ be a weighted graph, where $w : E \to \mathbb{R}$ assigns a weight to each edge. Given a source vertex $s \in V$, the shortest path problem is to determine the minimum-weight path from s to every vertex $v \in V$.

Algorithm (Bellman-Ford Algorithm).

(i) Begin with $d(s) = 0$ and $d(v) = \infty$ for all $v \neq s$

(ii) Repeat the following step $V - 1$ times:

(a) For each edge $(u, v) \in E$, update

$$d(v) = \min(d(v), d(u) + w(u, v))$$

(iii) Lastly, to check for negative cycles:

(a) For each edge $(u, v) \in E$, if $d(u) + w(u, v) < d(v)$, there exists a negative cycle and the shortest path cannot be found

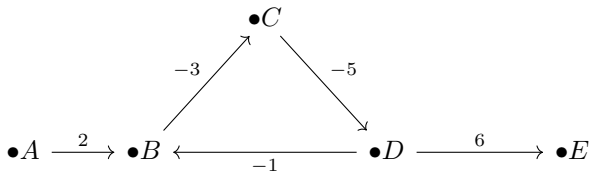
Note that in Bellman-Ford, there is no queue or empty set present; this is because the Bellman-Ford algorithm operates on a brute-force updating scheme that selects vertices to update based on adjacency to vertices with known distances. Bellman-Ford does not implement greedy choices, like Dijkstra's, and it also does not permanently fix distances to vertices, which is also used in Dijkstra's, so it has no use for recording a set of vertices.

Also, the algorithm can check whether there exists a negative cycle by seeing if any edges can still be updated because of a special graph property: in a graph with V vertices, the longest path that does not form a cycle contains a maximum of $|V| - 1$ edges.

Theorem 3.9. *Let $G = (V, E)$ be a weighted graph with no negative-weight cycles. Then, any shortest path between two vertices contains at most $|V| - 1$ edges.*

Proof. A path with at least $|V|$ edges must repeat a vertex, so it contains a cycle. If the graph contains no negative-weight cycles, removing the cycle cannot make the path longer. If the loop in our algorithm has run $|V| - 1$ times, on the Vth iteration (our checking step), no edges should update since all edges have been considered, and positive cycles will only output a longer path than what has already been recorded, which will not change estimated distances. Therefore, if any estimated distances change on our checking step, a negative cycle must be present. Thus, any shortest path can be written without cycles, and thus has at most $|V| - 1$ edges. □

Example 1. Consider the weighted graph below with source vertex A.



(i) To begin,

$$d(A) = 0, \qquad d(B) = d(C) = d(D) = d(E) = \infty.$$

The algorithm then selects A, and processes the edges in the order $(A, B), (B, C), (C, D), (D, B), (D, E)$, updating the distances to:

$$d(B) = \min(2, -7) = -7, \qquad d(C) = \min(\infty, -1) = -1, \qquad d(D) = \min(\infty, -6) = -6, \qquad d(E) = \min(\infty, 0) = 0.$$

Note that $d(B)$ is first updated to 2 since $\min(\infty, w(A, B)) = 2$. This value changes to -6 because the edge from D to B was considered when $d(D)$ was estimated to equal 2, and $d(D) + w(D, B) = 2 + (-9) = -7$. Thus, it can be said that the final value of $d(B)$ is fixed after that of $d(D)$.

(ii) Our algorithm will then loop onto a second pass of the edges, still in the same order. This will update the distances to:

$$d(B) = \min(-7, -15 + (-1)) = -16, \qquad d(C) = \min(-1, -7 - (-3)) = -10,$$
$$d(D) = \min(-6, -10 + (-5)) = -15, \qquad d(E) = \min(0, -15 + 6) = -9.$$

(iii) On the third looping of our algorithm, the distances are updated to:

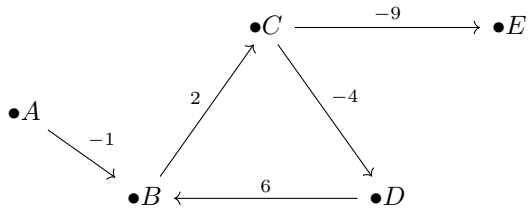
$$d(B) = \min(-16, -24 + (-1)) = -25, \qquad d(C) = \min(-10, -16 + (-3)) = -19,$$
$$d(D) = \min(-15, -19 + (-5)) = -24, \qquad d(E) = \min(-9, -24 + 6) = -18.$$

(iv) On the fourth loop, our algorithm updates the distances as:

$$d(B) = \min(-25, -33 + (-1)) = -34, \qquad d(C) = \min(-19, -25 + (-3)) = -28,$$
$$d(D) = \min(-24, -28 + (-5)) = -33), \qquad d(E) = \min(-18, -33 + 6) = -27.$$

(v) Finally, for our negative cycle checking step, we see if any values $d(A)$, $d(B)$, $d(C)$ or $d(D)$ can still be updated. Sure enough, $d(C)$ can be updated to $-34 + (-3) = -37$, which is a lower value than -28. Thus, a negative cycle is detected.

Example 2. Alternatively, consider the weighted graph below with source vertex A which has no negative cycles.



(i) Once again, our algorithm begins like so:

$$d(A) = 0, \qquad d(B) = d(C) = d(D) = d(E) = \infty.$$

The algorithm then selects A, and processes the edges in the order $(A, B), (B, C), (C, D), (D, B), (C, E)$, updating the distances to:

$$d(B) = \min(-1, 3) = -1, \qquad d(C) = \min(\infty, 1) = 1,$$
$$d(D) = \min(\infty, -3) = -3, \qquad d(E) = \min(\infty, -8) = -8.$$

(ii) Our algorithm will loop a second time, updating the distances to:

$$d(B) = \min(-1, -1 + 4) = -1, \qquad d(C) = \min(1, 1) = 1,$$
$$d(D) = \min(-3, -3) = -3, \qquad d(E) = \min(-8, -8) = -8.$$

Looking at our update values, we can see that the value of $d(B)$ will never update to anything less than its current value because traveling around the cycle another time will only increase the path length to B. For vertices C, E, and D, since there is only one way to reach these vertices through B, and the value of $d(B)$ does not change, $d(C)$, $d(D)$, and $d(E)$ will also not change. Thus, by our checking step, the distances will still update to:

$$d(B) = \min(-1, -1 + 4) = -1, \qquad d(C) = \min(1, 1) = 1,$$
$$d(D) = \min(-3, -3) = -3, \qquad d(E) = \min(-8, -8) = -8.$$

Therefore, the Bellman-Ford algorithm has finished processing shortest paths for this graph.

Theorem 3.10 (Correctness of the Bellman-Ford Algorithm)**.** *If a weighted graph contains no negative-weight cycles reachable from the source vertex s then the Bellman-Ford algorithm correctly computes the shortest path distances from s to every other vertex.*

Proof. The Bellman-Ford algorithm repeatedly relaxes every edge in the graph. Relaxing an edge (u, v) means checking whether the path from s to u, followed by the edge (u, v), gives a shorter path to v than the current estimate.

Suppose the shortest path from s to some vertex v contains k edges. After the first pass through all edges, all shortest paths of one edge are correctly computed. After the second pass, all shortest paths of two edges are correctly computed. After k passes, the shortest path to v will have been determined.

By Theorem 3.9, any shortest path in a graph with no negative-weight cycle contains at most $|V| - 1$ edges. Therefore, after $|V| - 1$ passes, all shortest path distances must be correct.

Finally, the algorithm performs one additional pass through all edges. If any distance can still be decreased,

then there exists a negative-weight cycle reachable from s. In such a case, repeatedly traversing the cycle would decrease the path length indefinitely, leading to a distance of $d(v) = -\infty$, so no shortest path exists. □

Time Complexity. The time complexity of the Bellman-Ford Algorithm is

$$O(|V||E|),$$

since the algorithm performs a total of $|V| - 1$ iterations, and during each iteration, every edge in the graph is relaxed once. Although the Bellman-Ford is slower than Dijkstra's, it is able to process graphs containing negative edge weights.

4 Applications

Graph algorithms have a wide variety of applications, from path-planning programs in GPS systems to programs that designate paths for autonomous vehicles. In Boston, one notable application of graph algorithms is trip planning through the Massachusetts Bay Transit Authority (MBTA) subway.

By treating every stop as a vertex, and subway routes connecting stops as edges, one can plan the quickest way to travel from one stop to another with the help of Dijkstra's.

5 Acknowledgments

We would like to thank our mentor, Rosa Paten, for guiding us through the program and our combinatorics research. We would also like to thank the PRIMES Circle coordinators, Paige Bright and Mary Stelow, for providing us with this opportunity and for all of their support and advice throughout the program.