Sample Proof Activity Report

Comp251, Winter 2022

Claim 1

The Bellman-Ford shortest path algorithm takes time $\Theta(E \cdot V)$ where V is the number of vertices and E is the number of edges.

Proof

This proof is adapted from Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms* [3].

Proof. There are 3 components of the Bellman-Ford Algorithm that we can reason about individually.

1. Initialize-Single-Source(G,s) In this function we iterate through each vertex of the graph once doing constant time work each time as we are only initializing two fields for each vertex: the distance from the source vertex s and the predecessor vertex on the shortest path. These are initialized as infinity and null respectively. The only exception is the distance from s to s is set at 0. This takes $\Theta(V)$ time.

2. After initialization we do $|V|-1$ iterations, each time relaxing every edge in the graph. The relax method takes $O(1)$ time as we are doing comparisons of numerical values, and updating fields. We update the shortest path and predecessor of a vertex if we find a shorter path by taking a particular edge. Since on each of the $\Theta(V)$ iterations, we iterate through every edge exactly once, doing constant time operations, the total time of this step is $\Theta(V \cdot E)$. There is no early stopping condition in this loop, therefore it is a tight bound. Since we are looking for a shortest path, we only iterate up to $|V|-1$ as a shortest path can only contain $|V|-1$ edges before it contains a cycle.

3. After the main loop, we examine our final values for negative weight cycles. This process involves iterating again through every edge in the graph, doing constant time comparisons. We check if the distance to the end vertex of an edge has a larger distance than the start vertex of the edge plus the value of taking the edge, and in this case return false immediately as this corresponds to a negative cycle. Therefore, this takes $O(E)$ time. It is not $\Omega(E)$ as in the best case we find a negative weight cycle quickly and can return that one exists immediately.

So, the total running time is then $\Theta(V \cdot E)$. Note again that this is a tight bound as in component 2, we always iterate through the E edges, $\Theta(V)$ times.

$\square$

Proof Summary

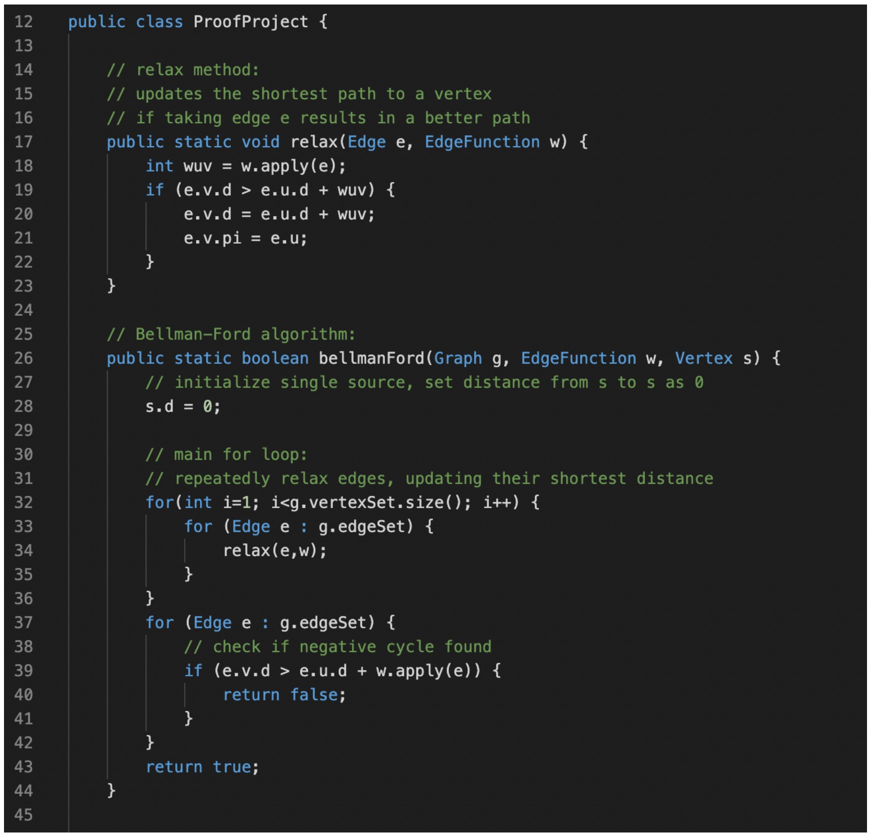
We argue the running time of each of the three main components in the algorithm. Initializing the graph vertices takes linear time as we visit each vertex exactly one once. The main for loop iterates through the number of vertices, and each iteration it iterates through all the edges in the graph, seeing if using that edge leads to a shorter cost path. Note, that we iterate through the number of vertices because the longest, non-cyclic path can visit at most all the vertices in the graph. Finally, we iterate over the edges once again to check for a cycle. Therefore, the total run time is dominated by the iteration that repeats for the number vertices and each time iterates through all the edges.

Algorithm

Below is the Java code for the Bellman-Ford shortest path method (Figure 1, Lines 24-42). I created the data structures needed for the algorithm. This includes the classes Vertex (Figure 2), Edge (Figure 3), Graph (Figure 4), and EdgeFunction (Figure 5). The initialize-single-source method is not needed in this variation of the algorithm as I initialize a vertex correctly when it is constructed (Figure 2, Lines 5-6). All that is necessary is to update the distance from the selected source vertex to be 0 (Figure 1, Line 26).

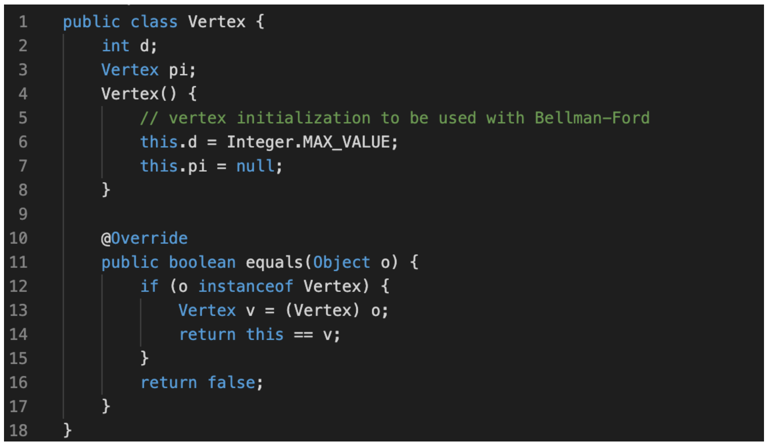
I time the execution of the Bellman-Ford algorithm when run on random graphs with n vertices and $2n$ edges. I start n at 10 and then increase it by 10 for each new instance. In total, I time the execution for up to $n = 1000$. The execution time is reported in microseconds. The plot below (Figure 6) shows the execution time as a function of the number of vertices, n. We expect the execution time to be quadratic in n ($V \cdot E = 2n^2$), and to be a tight bound. The graph confirms this claim. It can be closely modelled by the quadratic function of n shown.

The random graphs are created by creating a random path of length $n-1$ so that it is connected. The remaining edges are then added randomly. In addition, I generate a random weight function for each set of edges that assigns a nonzero weight in the range $(-100, 100)$ to the edge.



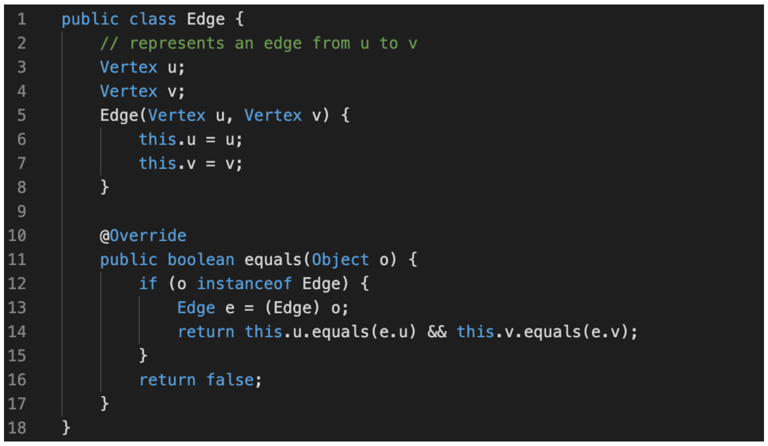
```java
public class ProofProject {

    // relax method:
    // updates the shortest path to a vertex
    // if taking edge e results in a better path
    public static void relax(Edge e, EdgeFunction w) {
        int wuv = w.apply(e);
        if (e.v.d > e.u.d + wuv) {
            e.v.d = e.u.d + wuv;
            e.v.pi = e.u;
        }
    }

    // Bellman-Ford algorithm:
    public static boolean bellmanFord(Graph g, EdgeFunction w, Vertex s) {
        // initialize single source, set distance from s to s as 0
        s.d = 0;

        // main for loop:
        // repeatedly relax edges, updating their shortest distance
        for(int i=1; i<g.vertexSet.size(); i++) {
            for (Edge e : g.edgeSet) {
                relax(e,w);
            }
        }
        for (Edge e : g.edgeSet) {
            // check if negative cycle found
            if (e.v.d > e.u.d + w.apply(e)) {
                return false;
            }
        }
        return true;
    }

```

Figure 1: The Bellman-Ford Algorithm from `ProofProject.java`.



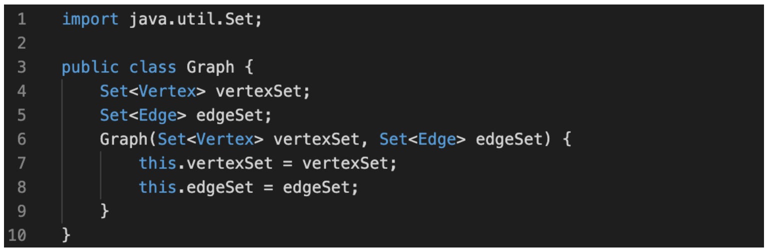
```java
public class Vertex {
    int d;
    Vertex pi;
    Vertex() {
        // vertex initialization to be used with Bellman-Ford
        this.d = Integer.MAX_VALUE;
        this.pi = null;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Vertex) {
            Vertex v = (Vertex) o;
            return this == v;
        }
        return false;
    }
}
```

Figure 2: Vertex class from `Vertex.java`.



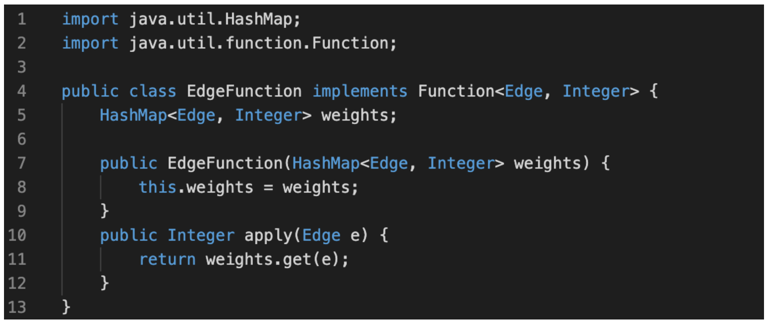
```java
public class Edge {
    // represents an edge from u to v
    Vertex u;
    Vertex v;
    Edge(Vertex u, Vertex v) {
        this.u = u;
        this.v = v;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Edge) {
            Edge e = (Edge) o;
            return this.u.equals(e.u) && this.v.equals(e.v);
        }
        return false;
    }
}
```

Figure 3: Edge class from `Edge.java`.



```java
import java.util.Set;

public class Graph {
    Set<Vertex> vertexSet;
    Set<Edge> edgeSet;
    Graph(Set<Vertex> vertexSet, Set<Edge> edgeSet) {
        this.vertexSet = vertexSet;
        this.edgeSet = edgeSet;
    }
}
```

Figure 4: Graph class from `Graph.java`.



```java
import java.util.HashMap;
import java.util.function.Function;

public class EdgeFunction implements Function<Edge, Integer> {
    HashMap<Edge, Integer> weights;

    public EdgeFunction(HashMap<Edge, Integer> weights) {
        this.weights = weights;
    }
    public Integer apply(Edge e) {
        return weights.get(e);
    }
}
```

Figure 5: EdgeFunction class from `EdgeFunction.java`.

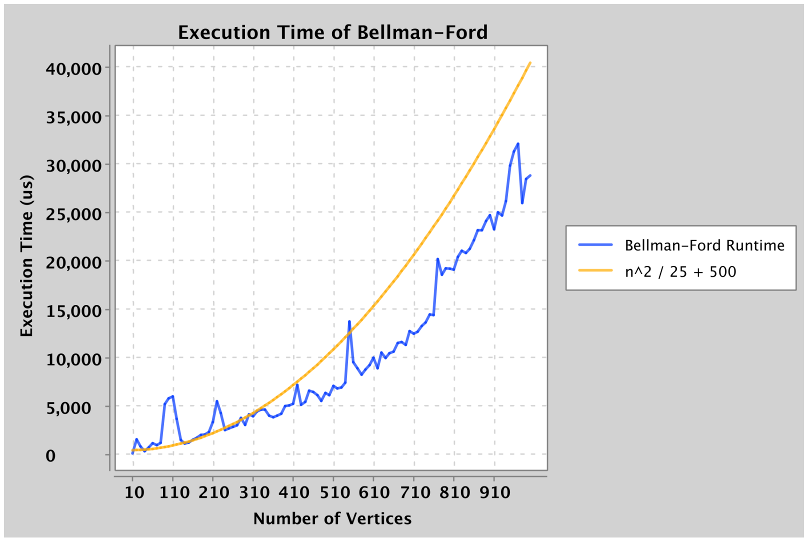


Figure 6: Execution time in microseconds of Bellman-Ford as a function of n. The code to generate this graph is in `ProofProject.java`.

Real World Application

The Bellman-Ford algorithm finds the shortest path in a directed graph that may contain negative weight cycles. This can be applied to any formulation of the shortest path problem. A useful application is data routing on a network, including many internet protocols. The goal is to send information through a network of routers as quickly as possible. We represent the network as a graph where vertices correspond to distributors in the network and the edges are links between distributors. The cost of an edge is the expected delay of sending information through that route. The Bellman-Ford algorithm can find the optimal route to send a packet of data from one vertex to the other as to minimize the delay. This example is from *Brilliant Math & Science Wiki*[1].

Claim 2

All maximal independent subsets in a matroid have the same size.

Proof

This proof is a paraphrased version of the proof written in [3], page 439. Let $M = (S, I)$ be a matroid and that $A, B \in I$ were maximally independent sets. Now suppose, to reach a contradiction, that $|A| < |B|$. Then by the exchange property of M there must exist an element $x \in B - A$ such that $A \cup \{x\} \in I$. This contradicts A being a maximally independent set in I, and therefore $|A| = |B|$.

Proof summary

The real meat of the proof is the exchange property of matroids, which says that if there are two independent subsets with one larger than the other, then we can add an element from the larger one which is not in the smaller one to the smaller one to get a new (larger) independent set. This gives us a way to build larger and larger independent sets until we reach some maximum size, which must be shared across all maximal independent sets. This property may reflect the original desired application of matrices, modelling linear independence of matrix rows, where vector space bases all have the same size (cardinality).

Algorithm

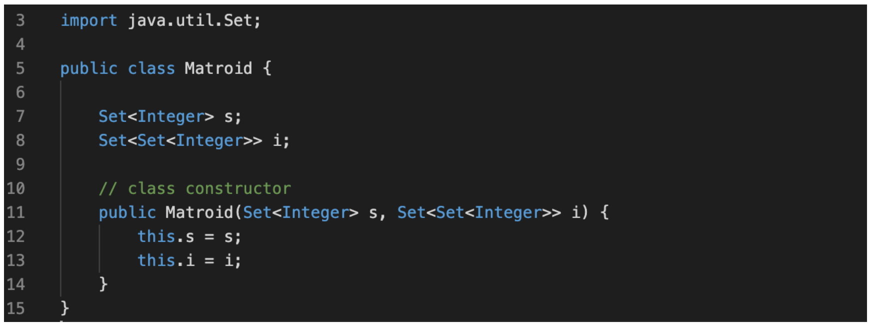
Below is the Java code for finding a maximal-weight independent subset of a matroid (Figure 8). The implementation required a basic matroid class (Figure 7) and a file which contains the algorithm, main function and several helper functions for the test cases.

The main function executes the algorithm on three cases. The construction of the test cases are shown in Figure 9. The output of the algorithm on the three test cases is shown in Figure 10.

First is a general case where the matroid has the underlying set $\{1, 2, 3\}$ and independent sets $\{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}\}$, and the weight vector is $\{1, 2, 3\}$ (the identity function). This is a general case because it is clear that the answer should be $\{2, 3\}$ (with a weight of 5, compared to $\{1, 2\}$ of weight 3 and $\{1, 3\}$ of weight 4).

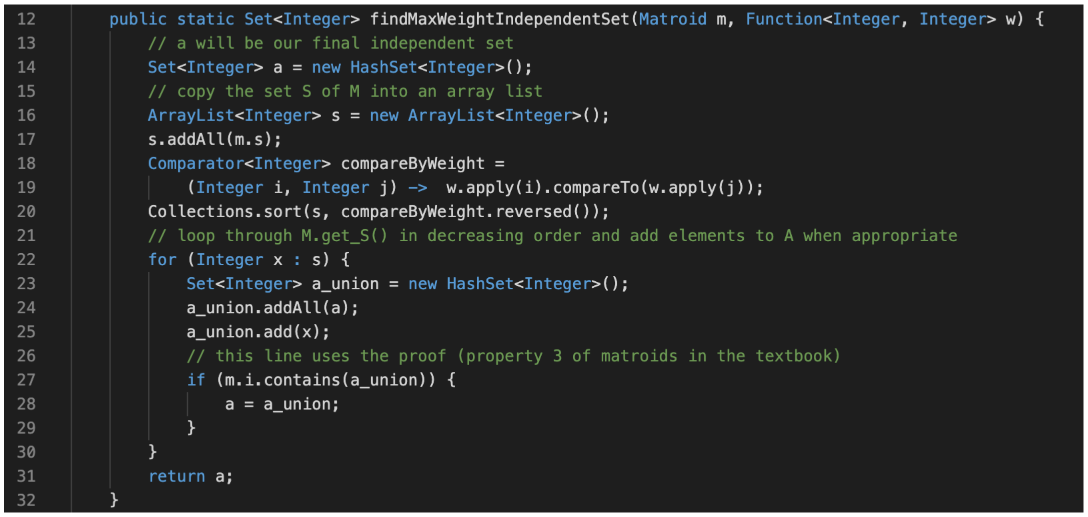
The first edge case has the same matroid and weight vector, except the only independent set is the empty set, $\phi = \{\}$. The expected and observed output in this case is the empty set.

Finally, another edge case is when there are multiple equally-weighted maximal independent sets (i.e. ensuring that one of the correct options is returned). For this test case we use the same matroid as in the general case, but set all weights to 1 (i.e. the weight vector was $\{1, 1, 1\}$). Any of the three maximal independent sets constitute a correct solution, including $\{1, 2\}$ which is what the algorithm returns.



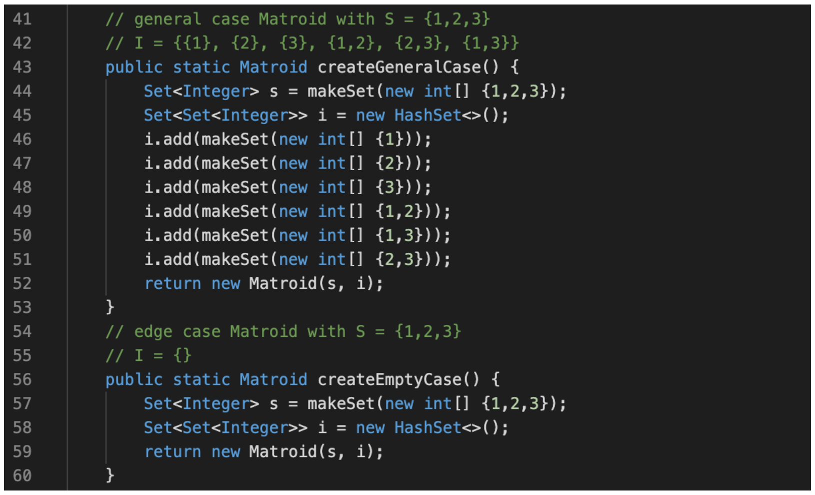
```java
import java.util.Set;

public class Matroid {

    Set<Integer> s;
    Set<Set<Integer>> i;

    // class constructor
    public Matroid(Set<Integer> s, Set<Set<Integer>> i) {
        this.s = s;
        this.i = i;
    }
}
```

Figure 7: Matroid class from `Matroid.java`.



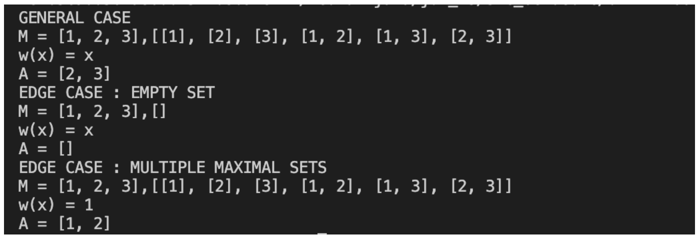
```java
    public static Set<Integer> findMaxWeightIndependentSet(Matroid m, Function<Integer, Integer> w) {
        // a will be our final independent set
        Set<Integer> a = new HashSet<Integer>();
        // copy the set S of M into an array list
        ArrayList<Integer> s = new ArrayList<Integer>();
        s.addAll(m.s);
        Comparator<Integer> compareByWeight =
            (Integer i, Integer j) ->  w.apply(i).compareTo(w.apply(j));
        Collections.sort(s, compareByWeight.reversed());
        // loop through M.get_S() in decreasing order and add elements to A when appropriate
        for (Integer x : s) {
            Set<Integer> a_union = new HashSet<Integer>();
            a_union.addAll(a);
            a_union.add(x);
            // this line uses the proof (property 3 of matroids in the textbook)
            if (m.i.contains(a_union)) {
                a = a_union;
            }
        }
        return a;
    }
```

Figure 8: Algorithm for finding the maximumal-weighted independent subset in a matroid from `Proof.java`.



```java
    // general case Matroid with S = {1,2,3}
    // I = {{1}, {2}, {3}, {1,2}, {2,3}, {1,3}}
    public static Matroid createGeneralCase() {
        Set<Integer> s = makeSet(new int[] {1,2,3});
        Set<Set<Integer>> i = new HashSet<>();
        i.add(makeSet(new int[] {1}));
        i.add(makeSet(new int[] {2}));
        i.add(makeSet(new int[] {3}));
        i.add(makeSet(new int[] {1,2}));
        i.add(makeSet(new int[] {1,3}));
        i.add(makeSet(new int[] {2,3}));
        return new Matroid(s, i);
    }
    // edge case Matroid with S = {1,2,3}
    // I = {}
    public static Matroid createEmptyCase() {
        Set<Integer> s = makeSet(new int[] {1,2,3});
        Set<Set<Integer>> i = new HashSet<>();
        return new Matroid(s, i);
    }
```

Figure 9: Functions to create a general test case matroid and edge case matroid from `Proof.java`.



```
GENERAL CASE
M = [1, 2, 3],[[1], [2], [3], [1, 2], [1, 3], [2, 3]]
w(x) = x
A = [2, 3]
EDGE CASE : EMPTY SET
M = [1, 2, 3],[]
w(x) = x
A = []
EDGE CASE : MULTIPLE MAXIMAL SETS
M = [1, 2, 3],[[1], [2], [3], [1, 2], [1, 3], [2, 3]]
w(x) = 1
A = [1, 2]
```

Figure 10: Console output after running the findMaxWeightIndependentSet method on the three test cases.

Real World Application

This algorithm can be used to find a maximally-weighted spanning tree of an undirected graph G by constructing a matroid M_G whose set S is the edges in G, I is the set of all acyclic subsets of S, and w is the weights of the edges ([3] pages 437-8). A real world example of such a usage would be minimal spanning trees (a simple adjustment to the algorithm) in telecommunication network coverage [2] - for instance a minimal cost and non-redundant way to ensure that cellphone towers can all communicate with each other. It turns out that many methods that were used for these types of problems reduce to minimum spanning tree problems.

References

[1] Bellman-ford algorithm. https://brilliant.org/wiki/bellman-ford-algorithm/. Brilliant Math Science Wiki.

[2] W. Chou and A. Kershenbaum. A unified algorithm for designing multidrop teleprocessing networks. In *Proceedings of the Third ACM Symposium on Data Communications and Data Networks: Analysis and Design*, DATACOMM '73, pages 148–156, New York, NY, USA, 1973. Association for Computing Machinery.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.