

Algorithms Analysis and Design

Project

A Book on

**Graph Algorithms, Greedy
Algorithms and Hard problems**

Done by:

Venneti Sri Satya Vinay (2020101066)

Code for the algorithms in this book is available at

<https://github.com/srisatyavinay/AAD-project>

Table of Contents

1 Introduction	4
1.1 Some basic terms and definitions	4
1.2 Graph representation	5
1.2.1 Adjacency Matrix	5
1.2.2 Adjacency List	5
1.3 Some basic algorithms and methods	6
1.3.1 Basic graph operations	6
1.3.2 Depth First Search (DFS)	6
1.3.3 Breadth First Search (DFS)	7
2 Graph Algorithms	9
2.1 Dijkstra's algorithm	9
2.1.1 Problem	9
2.1.2 Description	9
2.1.3 Algorithm	9
2.1.4 Pseudo code	10
2.1.5 Code	10
2.1.6 Time Complexity	10
2.1.7 Applications	11
2.2 Bellman Ford algorithm	11
2.2.1 Problem	11
2.2.2 Description	11
2.2.3 Algorithm	11
2.2.4 Pseudo code	12
2.2.5 Code	13
2.2.6 Time Complexity	13
2.2.7 Applications	13
2.3 Prim's algorithm	13
2.3.1 Problem	13
2.3.2 Description	13
2.3.3 Algorithm	13
2.3.4 Pseudo code	14
2.3.5 Code	14
2.3.6 Time Complexity	15
2.3.7 Applications	15
2.4 Topological sort	15
2.4.1 Problem	15

2.4.2 Description	15
2.4.3 Algorithm	15
2.4.4 Pseudo code	16
2.4.5 Code	16
2.4.6 Time Complexity	16
2.4.7 Applications	16
2.5 Kruskal's algorithm	16
2.5.1 Problem	16
2.5.2 Description	16
2.5.3 Algorithm	17
2.5.4 Pseudo code	17
2.5.5 Time Complexity	17
2.5.6 Applications	17
2.6 Ford-Fulkerson algorithm	18
2.6.1 Problem	18
2.6.2 Description	18
2.6.3 Algorithm	18
2.6.4 Time Complexity	19
2.6.5 Applications	19
3 Greedy Algorithms	19
3.1 Introduction	19
3.2 Minimum product subset of an array	20
3.2.1 Problem	20
3.2.2 Algorithm	20
3.2.3 Code	20
3.2.4 Time Complexity	20
3.3 Maximum sum of increasing order elements from n arrays	20
3.3.1 Problem	20
3.3.2 Algorithm	20
3.3.3 Code	21
3.3.4 Time Complexity	21
4 Hard Problems	21
4.1 Travelling salesman problem	21
4.1.1 Problem	21
4.1.2 Approximate algorithm	21
4.2 Vertex Cover Problem	22
4.2.1 Problem	22
4.2.2 Approximate algorithm	22

1 Introduction

1.1 Some basic terms and definitions

Some of the basic terms used in this document are

Graph:

A Graph consists of a finite set of vertices (or nodes) and set of edges which connect a pair of nodes. Graph is a non-linear data structure.

- The vertex set of G is denoted $V(G)$, or just V if there is no ambiguity.
- An edge between vertices u and v is written as $\{u, v\}$. The edge set of G is denoted $E(G)$, or just E if there is no ambiguity.
- Vertex set $V = \{0, 1, 2, 3, 4, 5\}$ and
- Edge set $E = \{(0, 1), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (3, 5)\}$ for the following graph.

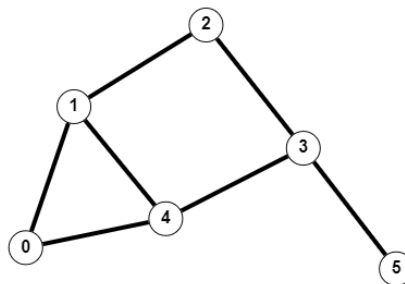


Fig 1: Graph

Degree:

The degree of a vertex v is the number of edges that connect to v .

Path:

A path in a graph $G = (V, E)$ is a sequence of vertices v_1, v_2, \dots, v_k , with the property that there are edges between v_i and v_{i+1} . We say that the path goes from v_1 to v_k . A path is simple if its vertices are all different.

Cycle:

A cycle is a path v_1, v_2, \dots, v_k for which $k > 2$, the first $k - 1$ vertices are all different, and $v_1 = v_k$. The sequence 4, 1, 2, 3, 4 is a cycle in the graph above.

Connected Graph:

A graph is connected if for every pair of vertices u and v , there is a path from u to v .

1.2 Graph representation

1.2.1 Adjacency Matrix

An adjacency matrix is a $|V| \times |V|$ matrix of integers, representing a graph $G = (V, E)$.

- The vertices are numbered from 1 to $|V|$.
- The number at position (i, j) indicates the number of edges from i to j .

Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done $O(1)$.

Consumes more space $O(V^2)$. Adding a vertex is $O(V^2)$ time.

For the graph in Fig 1 the adjacency matrix is as follows

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

1.2.2 Adjacency List

The adjacency list graph data structure consists of an array of size $|V|$, where position k in the array contains a linked list of all neighbours to k .

Saves space $O(|V| + |E|)$. In the worst case, there can be $C(V, 2)$ number of edges in a graph thus consuming $O(V^2)$ space. Adding a vertex is easier.

Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done $O(V)$.

Adjacency List			
0:	1	4	
1:	0	2	4
2:	1	3	
3:	2	4	5
4:	0	1	3
5:	3		

1.3 Some basic algorithms and methods

1.3.1 Basic graph operations

Basic graph operations include

- Creating a graph
- Creating a node
- Removing a node
- Creating an edge
- Deleting an edge

All these basic operations are implemented [here](#) or click on the link below.

https://github.com/srisatyavinay/AAD-project/blob/main/graph_algorithms/graph.c

1.3.2 Depth First Search (DFS)

1.3.2.1 Description

Depth first Search (DFS) or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Here **traversal** means visiting or touching all the nodes of the graph (in general for any data structure) and doing something with it. Depth first search (DFS) algorithm starts with the initial node of the graph G , and then goes to deeper and deeper until we find the goal node or the node which has no children. It then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

1.3.2.2 Algorithm

In this algorithm we first divide all the nodes into two categories – visited and not visited. We first make an array of the size equal to no.of vertices and initialise all the elements to zero indicating that all are not visited and then we change that element's value to 1 when we visit it. We use **stack** data structure in this algorithm

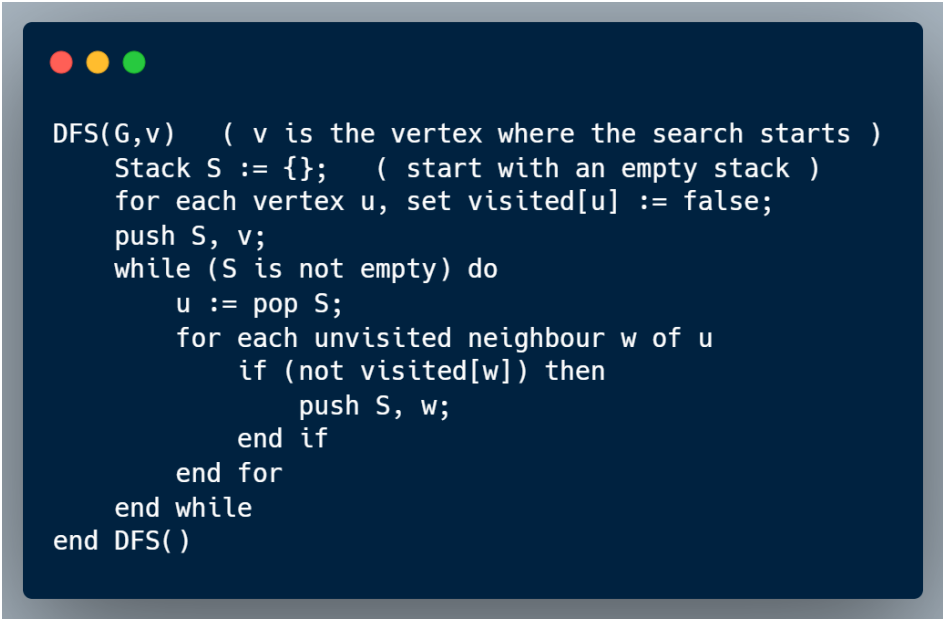
The algorithm is as follows

- 1) We first start by keeping the given source vertex at the top of stack.
- 2) We now pop the top element of stack and mark it as visited.
- 3) We then push all the adjacent vertices of this element into the stack.

4) Now repeat steps 2 and 3 till all the elements are visited.

The above algorithm is for connected graph now for disconnected graphs (and in general for both) we implement DFS algorithm on all vertices which are unvisited.

1.3.2.3 Pseudo-code



```
DFS(G,v)  ( v is the vertex where the search starts )
    Stack S := {};  ( start with an empty stack )
    for each vertex u, set visited[u] := false;
    push S, v;
    while (S is not empty) do
        u := pop S;
        for each unvisited neighbour w of u
            if (not visited[w]) then
                push S, w;
            end if
        end for
    end while
end DFS()
```

1.3.2.4 Code

The complete implementation Depth first search (DFS) algorithm can be found [here](https://github.com/srisatyavinay/AAD-project/blob/main/graph_algorithms/dfs.c) or click on the link below.

https://github.com/srisatyavinay/AAD-project/blob/main/graph_algorithms/dfs.c

1.3.3 Breadth First Search (BFS)

1.3.3.1 Description

Breadth first Search (BFS) or Depth first traversal is also a recursive algorithm for searching all the vertices of a graph or tree data structure like DFS. Here **traversal** means visiting or touching all the nodes of the graph (in general for any data structure) and doing something with it. The algorithm starts by traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

1.3.3.2 Algorithm

In this algorithm we first divide all the nodes into two categories – visited and not visited. We first make an array of the size equal to no.of vertices and initialise all the elements to zero indicating that all are not visited and then we change that element's value to 1 when we visit it. We use **queue** data structure in this algorithm

The algorithm is as follows

- 1) We first start by enqueueing the given source vertex from the rear end of the queue.
- 2) We now dequeue the topmost element of queue (from front end) and mark it as visited.
- 3) We then enqueue all the adjacent vertices of this element into the queue from the rear end.
- 4) Now repeat steps 2 and 3 till all the elements are visited.

The above algorithm is for connected graph now for disconnected graphs (and in general for both) we implement **BFS** algorithm on all vertices which are unvisited. Here the following pseudo-code is for BFS, but the code implemented is for modified BFS.

1.3.3.3 Pseudo-code

```
BFS(G,s)
  Queue Q := {};
  for each vertex u, set visited[s] := false;
  end for
  enqueue Q, s;
  while (Q is not empty) do
    u := dequeue Q;
    for each v adjacent to u do
      if (not visited[v]) then
        enqueue Q, v;
      end if
    end for
  end while
end BFS(G, s)
```

1.3.3.4 Code

The complete implementation Depth first search (DFS) algorithm can be found [here](#) or click on the link below.

2 Graph Algorithms

2.1 Dijkstra's algorithm

2.1.1 Problem

Given a graph (each edge has some weights) with a source and destination vertex we need to find the path having minimum costs i.e., we need to find the shortest path.

2.1.2 Description

We can use Dijkstra's technique to find the shortest path between any two network vertices. Because the shortest distance between two vertices may not include all of the graph's vertices, it varies from the minimum spanning tree. Dijkstra's algorithm is a greedy algorithm. Dijkstra's Algorithm doesn't work when there are negative edges in the graph.

2.1.3 Algorithm

Two sets of vertices are kept in Dijkstra's algorithm: one set comprises a list of vertices that have previously been included in SPT (Shortest Path Tree), while the other set contains vertices that have not yet been included. BFS can traverse all vertices of a graph in $O(V+E)$ time using adjacency list representation. The aim is to utilise BFS to explore all vertices of a graph and a Min Heap to store the vertices that aren't yet in SPT (or the vertices for which shortest distance is not finalised yet). Min Heap is a priority queue for obtaining the shortest distance vertex from a set of vertices that have not yet been included. For Min Heap, the time complexity of operations like extract-min and decrease-key value is $O(\log V)$.

The algorithm is as follows

- 1) Make a set called visited to keep track of the vertices in the graph, i.e., those which are visited. This collection is initially completely of 0's.
- 2) Assign a distance value to each of the input graph's vertices. All distance values should be set to INFINITE. Assign the source vertex a distance value of 0 to ensure that it is chosen first.

3) Because visited does not contain all vertices

a) Choose a vertex u that is not visited and has a minimal distance value.

b) Mark u as visited.

c) Update the distance between all of u 's neighbouring vertices. Iterate over all nearby vertices to update the distance values. If the total of the distance value of u (from source) and the weight of edge $u-v$ is less than the distance value of v for each neighbouring vertex v , then update the distance value of v .

2.1.4 Pseudo code

```
dijkstra(G, S)
  for each vertex V in G
    distance[V] <- infinite;
    If V != S, add V to Priority Queue Q;
  end for
  distance[S] <- 0;

  while Q IS NOT EMPTY do
    U <- Extract MIN from Q;
    for each unvisited neighbour V of U
      tempDistance <- distance[U] + edge_weight(U, V);
      if tempDistance < distance[V]
        distance[V] <- tempDistance;
      end if
    end for
  end while
  return distance[];
end dijkstra
```

2.1.5 Code

The complete implementation of Dijkstra's algorithm can be found by clicking on the link below.

https://github.com/srisatyavinay/AAD-project/blob/main/graph_algorithms/dijkstra.c

2.1.6 Time Complexity

The time complexity for the way implemented in the above link is $O(E \log V)$ but it can be reduced to $O(E + V \log V)$ if we use fibonacci heaps for priority queues.

2.1.7 Applications

- 1) **Google Maps Digital Mapping Services:** We've all attempted to discover the distance between cities in G-Maps, or from your position to the next desired site. Because there are several routes/paths linking them, the Shortest Path Algorithm is used to discover the shortest distance between two points along the way. Dijkstra's Algorithm is used to determine the shortest distance between two points along the path.
- 2) **Social Networking Apps:** You may have noticed that in many apps, the programme proposes a list of friends that a certain user may know. What do you believe the most efficient way for numerous social media businesses to integrate this functionality, especially when the system has over a billion users? The conventional Dijkstra method may be used to find the shortest path between users, which can be determined through handshakes or connections.

2.2 Bellman Ford algorithm

2.2.1 Problem

Given a graph (each edge has some weights) with a source and destination vertex we need to find the path having minimum costs i.e., we need to find the shortest path.

2.2.2 Description

The Bellman Ford algorithm is used to discover the shortest path between any two vertices in a weighted network. Unlike Dijkstra's algorithm we can also work with negative weighted edges but there should not be any negative cycles present.

2.2.3 Algorithm

The Bellman Ford method operates by overestimating the length of the path between the beginning and ending vertices. It then relaxes those estimations repeatedly by discovering new pathways that are shorter than the previously overestimated paths. We may ensure that the outcome is optimal by repeating this process for all vertices.

The algorithm is as follows

1) The distances from the source to all vertices are set to infinite, and the distance to the source is set to 0. Make a `dist[]` array of size $|V|$ with all values set to infinite except `dist[src]`, where `src` is the source vertex.

2) The shortest distances are calculated in this stage. Follow the steps below $|V| - 1$ times, where $|V|$ is the number of vertices in the supplied graph.

a) For each `u-v` edge, do the following

If `dist[v] > dist[u] + edge uv weight`, then `dist[v]` should be updated.

`dist[v] = dist[u] + edge weight uv`

3) If there is a negative weight cycle on the graph, this step will indicate it. For each `u-v` edge, do the following

"Graph has negative weight cycle" if `dist[v] > dist[u] + edge weight uv`.

Step 2 assures the shortest distances if the graph does not contain a negative weight cycle, which is the premise behind step 3. There is a negative weight cycle if we iterate through all edges one more time and obtain a shorter route for any vertex.

2.2.4 Pseudo code

```
bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite;
  end for
  distance[S] <- 0;

  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V);
      if tempDistance < distance[V]
        distance[V] <- tempDistance;
      end if
    end for
  end for

  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V]
      Error: Negative Cycle Exists;
    end if
  end for

  return distance[];
end bellmanford
```

2.2.5 Code

The complete implementation of Bellman ford's algorithm can be found by clicking on the link below.

https://github.com/srisatyavinay/AAD-project/blob/main/graph_algorithms/bellman_ford.c

2.2.6 Time Complexity

Time complexity of Bellman-Ford is $O(VE)$, which is more than Dijkstra.

2.2.7 Applications

It has the same applications as Dijkstra's algorithm but it can also be used in situations where negative weights are used like heat released in chemical reactions etc.

2.3 Prim's algorithm

2.3.1 Problem

Given a graph (each edge has some weights) spanning tree having minimum costs i.e., we need to find the minimum spanning tree (MST).

2.3.2 Description

We can use Prim's algorithm to find the minimum spanning tree of a graph. Prim's algorithm is a greedy algorithm. This algorithm also works with negative edge weights.

2.3.3 Algorithm

The process begins with an empty spanning tree. The goal is to keep two groups of vertices apart. The first set comprises vertices that have already been included in the MST, whereas the second set contains vertices that have not yet been included. It evaluates all the edges that connect the two sets at each step and selects the least weighted connection from among them. It transfers the other endpoint of the edge to the set containing MST after choosing the edge. The algorithm is very much similar to Dijkstra's algorithm.

The algorithm is as follows

- 1) Make a set called `mstSet` to keep track of the vertices that are already in MST.
- 2) Assign a key value to each of the input graph's vertices. All key values should be set to **INFINITE**. Assign the first vertex a key value of 0 to ensure that it is chosen first.
- 3) `mstSet` does not contain all vertices.
 - a) Choose a vertex `u` that is not in `mstSet` and has the smallest key value.
 - b) Add `u` to `mstSet`.
 - c) Change the key value of all `u`'s neighbouring vertices. Iterate over all nearby vertices to update the key values. If the weight of edge `u-v` is smaller than the previous key value of `v` for each neighbouring vertex `v`, update the key value to weight of `u-v`.

2.3.4 Pseudo code

```
prims(G)
  for each vertex V in G
    key[V] <- infinite;
    parent[v] <- -1;
    If V != 0, add V to Priority Queue Q;
  end for
  distance[0] <- 0;

  while Q IS NOT EMPTY do
    U <- Extract MIN from Q;
    for each unvisited neighbour V of U
      if edge_weight(U, V) < key[V];
        key[V] <- edge_weight(U, V);
        parent[V] <- U;
      end if
    end for
  end while
  return distance[];
end prims
```

2.3.5 Code

The complete implementation of Prim's algorithm can be found by clicking on the link below.

https://github.com/srisatyavinay/AAD-project/blob/main/graph_algorithms/prims.c

2.3.6 Time Complexity

The time complexity for the way implemented in the above link is $O(E \log V)$.

2.3.7 Applications

- 1) Minimum spanning trees are used to investigate how to put out electrical networks in such a way that the overall cost of the wiring is kept to a minimum. In a minimal spanning tree, all nodes (houses) are linked to power through cables in a cost-effective and redundant manner.
- 2) Assume you own a company with many locations. To link all of these offices, you'll need to rent phone wires. So you can use these algorithms to figure out how much it will cost to link all of your offices using the least amount of phone wires.

2.4 Topological sort

2.4.1 Problem

Given a Directed acyclic graph we need to find the topological sort order of the vertices of the given graph.

2.4.2 Description

For a Directed Acyclic Graph (DAG), topological sorting is a linear ordering of vertices in which vertex u occurs before v in the ordering for any directed edge $u \rightarrow v$. If the graph is not a DAG, topological sorting is not feasible.

2.4.3 Algorithm

DFS may be tweaked to discover a graph's Topological Sorting. In DFS, we start with a vertex, print it, and then call DFS for its nearby vertices recursively. A temporary stack is used in topological sorting. We don't display the vertex right away; instead, we call topological sorting for all of its nearby vertices in a recursive manner, then push it to a stack. Finally we print the stack's contents.

The algorithm is as follows

- 1) We first start by keeping the given source vertex at the top of stack.
- 2) We now pop the top element of stack and mark it as visited.

- 3) We then push all the adjacent vertices of this element into the stack.
- 4) Now repeat steps 2 and 3 till all the elements are visited.
- 5) We also maintain a second stack and push each vertex into it when all it's adjacent vertices are marked visited.
- 6) We finally pop all the vertices from the second stack and print them.

2.4.4 Pseudo code

For this the pseudo code is not necessary as it is just tweaking the DFS code as explained above. You can refer to DFS code and it's pseudo code above.

2.4.5 Code

The complete implementation of Topological sort can be found by clicking on the link below.

https://github.com/srisatyavinay/AAD-project/blob/main/graph_algorithms/topological_sort.c

2.4.6 Time Complexity

The time complexity for the way implemented in the above link is $O(V + E)$.

2.4.7 Applications

Topological Sorting is mostly used to schedule jobs based on work dependencies.

2.5 Kruskal's algorithm

2.5.1 Problem

Given a graph (each edge has some weights) spanning tree having minimum costs i.e., we need to find the minimum spanning tree (MST).

2.5.2 Description

We can use Kruskal's algorithm to find the minimum spanning tree of a graph. Kruskal's algorithm is a greedy algorithm. This algorithm also works with negative edge weights.

2.5.3 Algorithm

As we already know it belongs to a category of algorithms known as greedy algorithms, which seek for the local optimum in the hopes of discovering a global optimum. We begin with the lightest edges and gradually increase the weight of the edges until we accomplish our aim.

The algorithm is as follows

- 1) Sort all of the edges from light to heavy.
- 2) Add the spanning tree to the edge with the heaviest weight. If adding the edge resulted in a cycle, discard it.
- 3) Continue to add edges until we've reached all of the vertices.

2.5.4 Pseudo code

```
Kruskal(G):  
  A = ∅;  
  for each vertex v ∈ G.V:  
    MAKE-SET(v);  
  end for  
  for each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):  
    if FIND-SET(u) ≠ FIND-SET(v):  
      A = A ∪ {(u, v)};  
      UNION(u, v);  
    end if  
  end for  
  return A  
end Kruskal
```

2.5.5 Time Complexity

The overall time complexity of this algorithm is $O(E \log E)$ or $O(E \log V)$

2.5.6 Applications

The applications of Kruskal's algorithm are same as that of prim's algorithm. To see the applications please go to that section.

2.6 Ford-Fulkerson algorithm

2.6.1 Problem

Given a graph that represents a flow network with capacity at each edge. Find the largest possible flow from s to t with the following constraints: Given two vertices in the graph, source ' s ' and sink ' t ', find the maximum possible flow from s to t with the following constraints:

- a) The flow on an edge does not exceed the edge's capacity.
- b) For all vertex types except s and t , incoming flow equals outgoing flow.

2.6.2 Description

With this algorithm we can solve the maximum flow problem given a directed graph with capacities for every edge. This algorithm is also a greedy algorithm. We need to first learn some of the basic definitions before learning the algorithm.

Residual Graph: It's a graph which indicates additional possible flow. If there is such path from source to sink then there is a possibility to add flow.

Residual Capacity: It's the original capacity of the edge minus the flow through it.

Minimal Cut: It is also known as bottle-neck capacity which determines the maximum possible flow between source and sink through an augmented path.

Augmenting path: Augmenting path can be done in two ways

- Non-full forwarding edges
- Non-empty backward edges

2.6.3 Algorithm

As we already know it belongs to a category of algorithms known as greedy algorithms, which seek for the local optimum in the hopes of discovering a global optimum. We begin with the lightest edges and gradually increase the weight of the edges until we accomplish our aim.

The algorithm is as follows

- 1) Start with initial flow as 0.
- 2) While there is a augmenting path from source to sink. Add this path-flow to flow.
- 3) Return flow.

We can write the pseudo code based on the above algorithm so it is not given here.

2.6.4 Time Complexity

Time complexity of the above algorithm is $O(\text{max_flow} * E)$.

2.6.5 Applications

It's major application in real world is Maximum flow in Water Distribution Pipeline Network.

3 Greedy Algorithms

3.1 Introduction

A greedy algorithm is a method of solving a problem that chooses the best solution available at the time. It is unconcerned with whether the present best outcome will lead to the ultimate best result. Even if the option was incorrect, the algorithm never reverses the previous judgement. It operates in a top-down manner. For some issues, this approach may not provide the optimum solution. It's because it always opts for the best local option to get the greatest global outcome.

However, if the problem meets the following qualities, we can decide if the approach can be utilised with any problem:

1. Greedy Choice Property

A greedy approach can be used to solve a problem if an optimal solution can be discovered by selecting the best option at each stage and without reviewing the previous steps after they have been selected. This property is called greedy choice property.

2. Optimal Substructure Property

The problem can be addressed using a greedy approach if the optimal overall solution to the problem matches to the optimal solution to its subproblems. This is referred to as optimum substructure.

Now we will look at two simple examples of the greedy algorithms.

3.2 Minimum product subset of an array

3.2.1 Problem

Given an array a , we must find the smallest product feasible using the array's subset of items. A single ingredient might also be the simplest product.

3.2.2 Algorithm

- 1) If there are an even number of negative numbers and no zeros, the outcome is the product of all negative numbers except the largest.
- 2) If there are an odd number of negative numbers but no zeros, the outcome is just the sum of all the negative numbers.
- 3) The outcome is 0 if there are zeros and positives but no negatives. When there is no negative number and all other factors are positive, we should use the first minimum positive number as our result.

3.2.3 Code

The complete implementation of this problem can be found on clicking the link below.

https://github.com/srisatyavinay/AAD-project/blob/main/greedy_algorithms/minimum_product_subset.c

3.2.4 Time Complexity

The time complexity for the way implemented in the above link is $O(n)$.

3.3 Maximum sum of increasing order elements from n arrays

3.3.1 Problem

There are n arrays, each of size m . Find the largest sum by picking a number from each array so that the elements from the i -th array are greater than the items from the $(i-1)$ -th array. If the largest sum cannot be found, return 0.

3.3.2 Algorithm

- 1) The objective is to start with the most recent array.
- 2) We select the largest entry from the previous array, then continue on to the second-to-last array.
- 3) We locate the greatest element in the second final array that is smaller than the maximum element in the previous array.
- 4) This method is repeated until we reach the first array.

3.3.3 Code

The complete implementation of this problem can be found on clicking the link below.

https://github.com/srisatyavinay/AAD-project/blob/main/greedy_algorithms/min_sum_increasing_n_arrays.c

3.3.4 Time Complexity

The time complexity for the way implemented in the above link in the worst case is $O(mn \log m)$.

4 Hard Problems

4.1 Travelling salesman problem

4.1.1 Problem

The challenge is to discover the shortest feasible route that visits each city precisely once and returns to the beginning point given a collection of cities and distances between each pair of cities.

4.1.2 Approximate algorithm

This is a well-known NP-hard problem. This problem does not have a polynomial time solution. This algorithm below is one of the approximate naive algorithms.

- 1) Take city 1 as your beginning and finishing place. Because the path is cyclic, any point can be used as a beginning point.
- 2) All $(n-1)!$ permutations of cities should be generated.
- 3) Calculate the cost of each permutation and keep track of the permutation with the lowest cost.
- 4) Return the permutation with the lowest possible cost.

4.2 Vertex Cover Problem

4.1.1 Problem

Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.

A vertex cover of an undirected graph is a subset of its vertices in which either 'u' or 'v' is in the vertex cover for every graph edge (u, v).

4.1.2 Approximate algorithm

This is also a well-known NP-hard problem. This problem does not have a polynomial time solution. This algorithm below is one of the approximate naive algorithms.

- 1) Set the result's initial value to {}
- 2) Consider the collection of all the edges in a graph. Let's call the set E.
- 3) Carry out the steps below while E is not empty.
 - a. From set E, choose an arbitrary edge (u, v) and add 'u' and 'v' to the result.
 - b. Remove any edges of E that are incident on either u or v.
- 4) Return the outcome.