# Advanced Algorithms Notes

Riccardo Cappi

February 2024

## 0.1 Disclaimer

These are just my notes that I used to prepare for the exam. So, probably, there will be both spelling and conceptual errors. Feel free to contact me at riccardo.cappi@studenti.unipd.it if you find any errors. This is the github repo where you can find the latex files of the notes: `https://github.com/riccardocappi/Computer-Science-notes`

# Contents

0.1 Disclaimer . . . 2

**1 Lec 01 - Graph Algorithms I 7**
1.1 Graphs: The basis . . . 7
1.2 Terminology . . . 7
1.3 Concepts . . . 7
1.4 Basic problems . . . 8
1.5 Notations and Properties . . . 8

**2 Lec 02 - Graphs Representation and DFS 9**
2.1 Graphs Representation . . . 9
2.2 Graph search and its application . . . 10
  2.2.1 DFS Algorithm . . . 10
  2.2.2 Correctness of the algorithm . . . 11
  2.2.3 Complexity of the algorithm . . . 11
  2.2.4 DFS Extension . . . 12
  2.2.5 Problem 1 . . . 12
  2.2.6 Problem 2 . . . 12

**3 Lec 03 - From DFS to BFS 15**
3.1 More applications of DFS . . . 15
  3.1.1 Connected Components . . . 15
  3.1.2 Connectivity . . . 15
3.2 Summary . . . 15
3.3 Breadth-First search (BFS) . . . 16
  3.3.1 Correctness . . . 17
  3.3.2 Complexity . . . 17
  3.3.3 Applications . . . 17

**4 Lec 04 - Minimum Spanning Tree 19**
4.1 Minimum Spanning Tree (MST) . . . 19
4.2 Generic greedy algorithm . . . 19
4.3 Prim's alorithm . . . 20

**5 Lec 05 - Prim's & Kruskal's algorithms 23**
5.1 Efficient Prim's algorithm . . . 23
5.2 Kruskal's algorithm . . . 24

**6 Lec 06 - Efficient Kruskal & Union-Find 25**
6.1 Efficient Kruskal . . . 25
6.2 Union-Find implementation . . . 26

7 Lec 07 - Shortest Path 29
7.1 Definitions and Terminology 29
7.2 Problem: Shortest Path 29
7.3 Single-Source Shortest Paths (SSSP) 29
7.3.1 Special case: non-negative edge weights 30

8 Lec 08 - General SSSP Problem 33
8.1 General SSSP Problem 33
8.1.1 Bellman-Ford algorithm 34

9 Lec 09 - All-Pairs Shortest Paths (APSP) 37
9.1 All-Pairs Shortest Paths (APSP) 37
9.1.1 Floyd-Warshall algorithm 37
9.2 Bellman-Ford algorithm via dynamic programming 39

10 Lec 10 - Maximum Flows 41
10.1 Definitions 41
10.2 Maximum flow problem 41
10.3 Ford-Fulkerson method 42

11 Lec 11 - NP-Hardness 45
11.1 Tractable vs Intractable problems 45
11.2 NP-Hard 45
11.3 Cook-Levin Theorem 46
11.4 Reductions 47

12 Lec 12 - Reduction 49
12.1 NP-Hardness (formal definition) 49
12.2 NP-Hardness proof 49
12.3 Maximum Independent Set 50

13 Lec 13 - Approximation Algorithms 53
13.1 Approximation Algorithms 53
13.1.1 Approximation 53
13.2 Approximation Algorithm for Vertex Cover 54
13.2.1 Analysis 55

14 Lec 14 -TSP & Metric TSP 57
14.1 Travelling Salesperson Problem 57
14.2 Metric TSP 58

15 Lec 15 - 2-Approximation Algorithm for Metric TSP 59
15.1 Metric TSP: A 2-Approximation Algorithm 59
15.1.1 Analysis of the cost of $H$ 60

16 Lec 16 - A 1.5-Approximation Algorithm for Metric TSP 63
16.1 Christofides' algorithm 63
16.1.1 Analysis 65

17 Lec 17 - Set Cover 67
17.1 Set cover 67
17.2 Approximation algorithm: greedy approach 68
17.3 Analysis 68

**18 Lec 18 - Randomized Algorithms** **71**
- 18.1 Randomized algorithms . . . 71

**19 Lec 19-20 - Minimum cut** **73**
- 19.1 Classification of randomized algorithms . . . 73
- 19.2 Terminology . . . 73
- 19.3 Karger's algorithm for Minimum cut . . . 74
- 19.4 Analysis of Karger's algorithm . . . 76
  - 19.4.1 Conditional probability recall . . . 76
  - 19.4.2 Analysis of *FULL_CONTRACTION* . . . 77
  - 19.4.3 Complexity . . . 78

**20 Lec 21 - Chernoff bounds** **79**
- 20.1 Chernoff bounds . . . 79
  - 20.1.1 A more powerful tool . . . 79
  - 20.1.2 Variants of Chernoff bound: . . . 80

**21 Lec 22 - Analysis of Randomized Quicksort** **81**
- 21.1 Randomized Quicksort . . . 81
- 21.2 Analysis . . . 82

# Chapter 1

# Lec 01 - Graph Algorithms I

## 1.1 Graphs: The basis

A graph is a representation of the relationships between **pairs** of objects. We denote a graph as follows:

$$G = (V, E)$$

where $V$ is a **set** of vertices (nodes) and $E$ is a collection of edges. An edge is a pair of vertices $(u, v)$ which indicates the connection between the two nodes.

- if $(u, v) = (v, u)$ the graph is **undirected**
- if $(u, v) \neq (v, u)$ the graph is **directed**

In directed graphs an edge is usually called *"arc"*.

Examples of real-world graphs are:

- Road networks: $(Cities, Roads)$
- Computer networks (e.g. internet): $(Computers, connections)$
- World Wide Web: $(Webpages, hyperlinks)$
- Social networks: $(People, friendship\ connections)$

## 1.2 Terminology

- Given an edge $e = (u, v)$, $e$ is **incident** on $u$ and $v$, while $u, v$ are **adjacent**.
- The **neighbors** of a vertex $u$ are all the vertices $v$ such that $(u, v) \in E$.
- The **degree** of a vertex $d(v)$ is the number of edges incident on $v$.

## 1.3 Concepts

- **Simple path**: A sequence of nodes $v_1, v_2, ..., v_k$ all distinct such that $(v_i, v_{i+1}) \in E \; \forall 1 \leq i < k$. Having all distinct nodes makes the path a simple path.
- **Cycle**: is a path such that $v_1 = v_k$

- **Sub-graph:** $G' = (V', E')$ such that $V' \subseteq V,\ E' \subseteq E$ and the edges of $E'$ are incident only on $V'$.
- **Spanning sub-graph:** a sub-graph with $V' = V$
- **Connected graph:** a graph is connected if $\forall u, v \in V$ it exists a path from $u$ to $v$.
- **Connected components:** a partition of $G$ in sub-graphs $G_i = (V_i, E_i)$ such that $\forall 1 \leq i < k$
  - $G_i$ is connected.
  - $V = V_1 \cup V_2 \cup ... \cup V_k$
  - $E = E_1 \cup E_2 \cup ... \cup E_k$
  - $\forall i \neq j$ there is no edge between $V_i$ and $V_j$

  If $G$ is connected, then $k = 1$.
- **Tree:** connected graph without cycles.
- **Forest:** set of trees (disjoint)
- **Spanning tree:** a spanning connected sub-graph without cycles (it exists only if $G$ is connected).
- **Spanning forest:** a spanning sub-graph without cycles.

## 1.4 Basic problems

- Traversal
- Connectivity (tell if the graph is connected or not)
- Compute connected components
- Minimum-weight spanning trees
- Shortest paths
- ...

## 1.5 Notations and Properties

- $n = |V|$
- $m = |E|$
- The **size** of the graph is given by $n + m$
- $\sum_{v \in V} d(v) = 2m$ because in the summation every edge counts twice.
- $m \leq \binom{n}{2}$
- if $G$ is a tree $m = n - 1$ because, fixed a root, $E$ represents father-child similarities, which are $n - 1$.
- if $G$ is connected $m \geq n - 1$ because $G$ is a *"tree"* that may have cycles, thus it can only have more edges.
- if $G$ is acyclic $m \leq n - 1$ because $G$ is a tree that may not be connected, thus it can only have less edges.

# Chapter 2

# Lec 02 - Graphs Representation and DFS

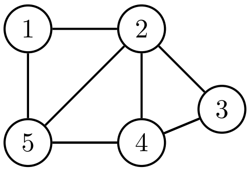
## 2.1 Graphs Representation

How can we encode a graph for using it in an algorithm ? A first simple solution can consist of using a list of vertices $L_V$ and a list of edges $L_E$. However, this representation does not allow for fast algorithms.

In order to allow for **direct access to edges**, the following data structures are used in addition to $L_V$ and $L_E$.

- **Adjacency list:** An array of $n$ list, one for each vertex $v \in V$, each containing all the vertices adjacent to $v$.
  - Pro: Space required → $\Theta(n + m)$, that is **linear**.
  - Cons: No quick way to determine if an edge is present in the graph.
- **Adjacency matrix:** A $n \times n$ matrix $A$ such that $A[i, j] = 1$ if $edge(i, j) \in E$, 0 otherwise.

  **Example:**

  The Adjacency matrix of the graph above is the following:

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

  In undirected graphs this matrix is **symmetric**, while in directed graphs it is **asymmetric**. In case of a weighted graph, each cell of the matrix has either the value of the edge weight $w$ or $-$.

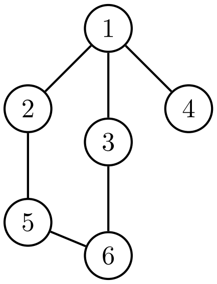
  - Pro: Quick to determine if a given edge is present.
  - Cons: Space required → $\Theta(n^2)$.

## 2.2 Graph search and its application

Graph search algorithms provide a systematic way to **explore** a graph starting from a vertex $s \in V$ visiting all the vertices.

The most famous algorithms are:

- **Depth-first search** (DFS).
- **Breadth-first search** (BFS).



**DFS:** $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 4$
**BFS:** $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$

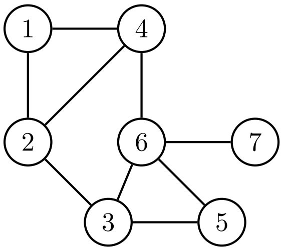
### 2.2.1 DFS Algorithm

The DFS algorithm is a recursive algorithm that, starting from a source $s \in V$, visits all the vertices of the **connected component** $C_s \subseteq G$ containing $s$.

- We use the adjacency list as graph representation.
- Every vertex $v$ has a field $L_v[v].ID$ which can either be 1 if the vertex has been visited, 0 otherwise.
- Every edge $e$ has a field $L_E[e].Label$ which can either be null or *Discovery edge / Back edge* [1].

Initially, every vertex ID is 0 and each edge Label is null.

**Example:**



**DFS:** $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 4 \rightarrow 7$

---

[1] During the search, we label the edges. This operation is useful for solving some problems.

**Algorithm 1** DFS

```
procedure DFS (G, v)
    visit $v$
    $L_V[v].ID \leftarrow 1$
    for $e \in G$.incidentEdges($v$) : do
        if $L_E[e].label = null$ then
            $w \leftarrow G$.opposite($v, e$)
            if $L_V[w].ID = 0$ then
                $L_E[e].label \leftarrow Discovery\ Edge$
                DFS(G, w)
            else
                $L_E[e].label \leftarrow Back\ Edge$
```

## 2.2.2 Correctness of the algorithm

At the end of the algorithm:

1. All the vertices of $C_s$ have been visited, and all the edges in $C_s$ are labelled either *Discovery Edge* or *Back Edge*
2. The set of *Discovery Edges* is a **spanning tree** $T$ of $C_s$.

**Proof:**

1. A vertex $u$ is visited when DFS$(G, u)$ is called. Assume by contradiction that $\exists u \in C_s$ not visited. Since $C_s$ is connected, it exists a path from $s$ to $u$ $s = u_0 \rightarrow u_1 \rightarrow u_2, ..., u_k = u$. Let $u_i$ be the first unvisited vertex in the path. When DFS is called on $u_{i-1}$ it finds $u_i$ not visited and therefore DFS$(G, u_i)$ is called. Then $u_i$ is visited. This is in contradiction with the initial hypothesis.

   DFS is called $\forall v \in C_s$. Therefore, all incident edges on $v$ are labelled, by construction.
2. DFS is called for every vertex $v \in C_s$ once, and $\forall v \neq s$ it exists a vertex $u$ such that the edge $(u, v)$ exists and is labelled *Discovery Edge*. This implies that $\forall v \in C_s$ $v \neq s$ it exists a *father* and going back father to father, eventually $s$ is reached. Then, the set of *Discovery Edges* is a rooted tree that *touches* all the vertices of $C_s$, that is, a spanning tree of $C_s$.

## 2.2.3 Complexity of the algorithm

Given:

- $n_s$: number of vertices of $C_s$.
- $m_s$: number of edges of $C_s$.

The complexity of DFS is:

$$\Theta\left(\sum_{v \in C_s} d(v)\right) = \Theta(m_s)$$

Note that $C_s$ is connected, so $m_s \geq n_s - 1 \rightarrow m_s = \Omega(n_s)$.

### 2.2.4 DFS Extension

DFS$(G, s)$ visits only the connected component of $s$. If the graph is not connected and we want to visit all the graph, the DFS algorithm can be extended as follows:

**Algorithm 2** DFS

```
procedure DFS $(G, v)$
    for $v \leftarrow 1$ to $n$ do
        if $L_V[v].ID = 0$ then
            DFS$(G, v)$
```

Its complexity is $\Theta(n + m)$ because it scans over all the vertices.

### 2.2.5 Problem 1

Given a graph $G$ and two vertices $s, t$, determines, if exists, a path from $s$ to $v$.

Modified version of DFS:

**Algorithm 3** DFS

```
procedure DFS (G, V)
    visit $v$
    $L_V[v].ID \leftarrow 1$
    for $e \in G$.incidentEdges$(v)$ : do
        if $L_E[e].label = null$ then
            $w \leftarrow G$.opposite$(v, e)$
            if $L_V[w].ID = 0$ then
                $L_E[e].label \leftarrow Discovery\ Edge$
                $L_V[w].parent \leftarrow v$
                DFS(G, w)
            else
                $L_E[e].label \leftarrow Back\ Edge$
```

- $\forall v \in V$ add a field $L_V[v].parent$
- Modify the DFS algorithm such that when a *Discovery Edge* $(v, w)$ is labelled, we set $L_V[w].parent = v$.
- Run DFS$(G, s)$, starting from the source $s$, and check if $t$ jas been visited. If $t$ has not been visited, there is no path from $s$ to $t$, otherwise build the path starting from $t$ and going backward parent to parent until $s$ is reached.

The complexity of the algorithm is the same as DFS since we only added constant operations.

### 2.2.6 Problem 2

Given a graph determine a cycle (if any):

**Algorithm 4** DFS

```
procedure DFS (G, V)
    visit $v$
    $L_V[v].ID \leftarrow 1$
    for $e \in G$.incidentEdges($v$) : do
        if $L_E[e].label = null$ then
            $w \leftarrow G$.opposite($v, e$)
            if $L_V[w].ID = 0$ then
                $L_E[e].label \leftarrow$ Discovery Edge
                $L_V[w].parent \leftarrow v$
                DFS(G, w)
            else
                $L_E[e].label \leftarrow$ Back Edge
                $L_E[e].ancestor \leftarrow w$
```

- $\forall v \in V$ add a field $L_V[v].parent$
- $\forall e \in E$ add a field $L_E[e].ancestor$
- Modify the DFS algorithm such that if a *Discovery Edge* $(v, w)$ is labelled we set $L_V[w].parent = v$, and if $(v, w)$ is labelled as *Back Edge*, we set $L_E[e].ancestor = w$.
- Run DFS on each connected component.
- Check all the edges. When a *Back Edge* $e = (v, w)$ is found and $L_E[e].ancestor = w$, return a cycle adding to $e$ all the edges found in the path from $v$ to $w$.

Since we run DFS on each connected component, the complexity is $\Theta(n + m)$ (we scan all the nodes).

# Chapter 3

# Lec 03 - From DFS to BFS

## 3.1 More applications of DFS

DFS can also be used to solve the following 2 problems:

- Connected Components: Labelling all the vertices of $G$ such that 2 vertices have the same label iff they are in the same connected components.
- Connectivity: Return whether the graph is connected or not.

### 3.1.1 Connected Components

**Algorithm 5** ConnectedComponents

```
procedure CONNECTEDCOMP ($G$)
    for $v \leftarrow 1$ to $n$ do
        $L_V[v].ID = 0$
    $k \leftarrow 0$
    for $v \leftarrow 1$ to $n$ do
        if $L_V[v].ID = 0$ then
            $k \leftarrow k + 1$
            DFS($G, v, k$)
```

We have to modify the DFS algorithm in such a way that $L_V[v] \leftarrow k$. The $ID$ field of each node is no more either 0 or 1, but it corresponds to the ID of the $k$-th connected component.

### 3.1.2 Connectivity

The algorithm is similar to the one used for computing the connected components, but in this case we return if the graph is connected or not ($k = 1$).

The complexity of both algorithms is $\Theta(n + m)$.

## 3.2 Summary

Given a graph $G = (V, E)$, the following problems can be solved using DFS:

**Algorithm 6** Connectivity

```
procedure CONNECTIVITY ($G$)
    for $v \leftarrow 1$ to $n$ do
        $L_V[v].ID = 0$
    $k \leftarrow 0$
    for $v \leftarrow 1$ to $n$ do
        if $L_V[v].ID = 0$ then
            $k \leftarrow k + 1$
            DFS($G, v, k$)
    if $k = 1$ return $Yes$ else $False$
```

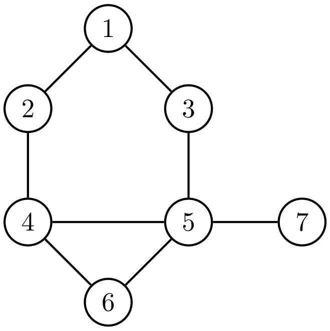
- Test if $G$ is connected.
- Find the connected components of $G$.
- Find a spanning tree of $G$ (if $G$ is connected).
- Find a path between 2 vertices (if any).
- Find a cycle (if any).

## 3.3 Breadth-First search (BFS)

BFS is an **iterative** algorithm that, starting from a source vertex $s$, visits all the vertices in the same connected components of $s$, and it partitions all the vertices in levels $L_i$ depending on their distance $i$ from $s$.

As we did for DFS, every vertex $v$ has a field $L_v[v].ID$ which can either be 1 if the vertex has been visited, 0 otherwise. Furhermore, every edge $e$ has a field $L_E[e].Label$ which can either be null or *Discovery edge / Cross edge*.

**Example:**



**DFS:** $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$

- $L_0 = \{1\}$
- $L_1 = \{2, 3\}$
- $L_2 = \{4, 5\}$
- $L_3 = \{6, 7\}$

**Algorithm 7** BFS

```
procedure BFS $(G, s)$
    visit(s)
    $L_V[s].ID = 1$
    Create a set $L_0$ containing $s$
    $i \leftarrow 0$
    while ! $L_i.isEmpty$ do
        Create a set of vertices $L_{i+1}$
        for $v \in L_i$ do
            for $e \in G.incidentEdges(v)$ do
                if $L_E[e].label = null$ then
                    $w \leftarrow G.opposite(v, e)$
                    if $L_V[w].ID = 0$ then
                        $L_E[e].label \leftarrow$ Discovery Edge
                        visit($w$)
                        $L_V[w].ID \leftarrow 1$
                        add $w$ in $L_{i+1}$
                    else
                        $L_E[e].label \leftarrow$ Cross Edge
        $i \leftarrow i + 1$
```

### 3.3.1 Correctness

At the end of BFS$(G, s)$ we have:

1. All the vertices in $C_s$ are visited and all the edges are labelled *Discovery Edges* or *Cross Edges*
2. The set of *Discovery Edges* are a spanning tree $T$ of $C_s$ which is called BFS Tree.
3. $\forall v \in L_i$ the path in $T$ from $s$ to $v$ has $i$ edges and every other path from $s$ to $v$ has at least $i$ edges (i.e. $\geq i$ edges)

The proof of the first two properties is the same as for the DFS.

**Proof of 3:**
Let $P : s = u_0 \to u_1 \to \ldots \to u_i = v$ a path from $s$ to $v$, where $u_j \in L_j$ is *discovered* from the vertex $u_{j-1} \forall j$. Then, the edge $(u_{j-1}, u_j)$ is a *Discovery Edge*. By contradiction, assume assume $\exists$ a path $P' : s = z_0 \to z_1 \to \ldots \to z_t = v$ with $t < i$. This implies that $s = z_0 \in L_0$, $z_1 \in L_1$, $z_2 \in L_1$ *or* $L_2$, ..., $z_t \in L_1$ *or* $L_2$ *or* ... $L_t$. This means that, since $z_t = v$, $v \notin L_i$, but this is a contradiction.

### 3.3.2 Complexity

$\forall v \in C_s$ there is one iteration of the first **for** loop and $d(v)$ iterations of the second **for** loop. Then, the complexity of BFS is:

$$\Theta\left(\sum_{v \in C_s} d(v)\right) = \Theta(m_s)$$

### 3.3.3 Applications

- Same as for DFS in $\Theta(n + m)$ time

- Given a graph $G = (V, E)$ and $s, t \in V$ return the **shortest** path from $s$ to $v$ (if any) [1]. In order to solve this problem, we can build the path following the same approach as for DFS.

---

[1] Shortest path in terms of number of edges between the two nodes

# Chapter 4

# Lec 04 - Minimum Spanning Tree

## 4.1 Minimum Spanning Tree (MST)

**Definition:**

- **Input:** A graph $G = (V, E)$ undirected, connected, weighted. A weight $w(u, v)$ defines the cost of an edge ($w : E \to \mathbb{R}$)
- **Output:** A spanning tree $T \subseteq E$ such that $W(T) = \sum_{(u,v) \in T} w(u, v)$ is minimized.

Applications of MST are:

- Networks (computers, sensors, electrical)
- Machine Learning (clustering algorithms)
- Computer vision (object detection)
- Data mining
- Subroutine in other algorithms (approximation algorithms)

The simplest MST algorithm is to compute all the spanning trees and select the one with minimum weight. This solution is not possible because the so called **complete graph**, which is a graph that contains all the $\binom{n}{2}$ possible edges, has $n^{n-2}$ different spanning trees. Then, the algorithm would be exponential in the worst case.

However, MST can be solved in near-linear time using **greedy algorithms**. We'll see two algorithms that deal with this problem:

- Prim's algorithm
- Kruskal algorithm

They both apply, in different ways, a **generic greedy algorithm**.

## 4.2 Generic greedy algorithm

The idea of a generic-MST algorithm is to maintain the following invariant: At each iteration, $A$ is always a subset of edges of some MST. Basically, at each iteration, the algorithm adds an edge that does not violate the invariant (i.e. *safe edge* for $A$).

**Terminology:**

- A **cut** of a graph $G = (V, E)$ is a partition of the set of vertices. It is usually denoted as $(S, V \setminus S)$.
- An edge $(u, v) \in E$ **crosses** a cut $(S, V \setminus S)$ if $u \in S$ and $v \in V \setminus S$.
- A cut **respects** a set of edges $A$ if no edge of $A$ crosses the cut.
- Given a cut, an edge that crosses the cut and is of minimum weight is called **light-edge**.

**Algorithm 8** Generic MST

```
procedure GENERIC-MST ($G$)
    $A = \emptyset$
    while $A$ does not form a spanning tree do
        find an edge $(u, v)$ that is safe for $A$
    return $A$
```

How can we find a *safe edge* ? Luckily, MSTs enjoy the following structural property:

**Theorem:** Let $G = (V, E)$ be an undirected, connected and weighted graph. Let $A$ be a subset of $E$ included in some MST of $G$. Let $(S, V \setminus S)$ a cut that respects $A$. Let $(u, v)$ be a light-edge for $(S, V \setminus S)$. Then, $u, v$ is safe for $A$.

**Termination**: All edges added to $A$ are in a minimum spanning tree, and so the set $A$ must be a minimum spanning tree.

Basically we can iteratively select a cut that respects $A$ and add to $A$ a light-edge according to the cut. Prim and Kruskal algorithms define how to choose a cut.

**Proof of the theorem:**
Let $T$ be an MST that includes $A$. Assume that $(u, v) \notin T$. We'll build a new MST $T'$ that includes $A \cup \{(u, v)\}$. This would imply that the edge $(u, v)$ is safe for $A$, and the theorem would be demonstrated. By hypothesis, $(u, v)$ crosses the cut $(S, V \setminus S)$. This implies that there exists another edge that crosses $(S, V \setminus S)$. This is because $(u, v) \notin T$, so if no other edge crosses the cut it would mean that $G$ is not connected, which is false by hypothesis. Let $(x, y)$ be such edge. By hypothesis $(S, V \setminus S)$ respects $A$, therefore $(x, y) \notin A$. This implies that by removing $(x, y)$ from $T$ and adding $(u, v)$ we obtain a new spanning tree $T' = T \setminus \{(x, y)\} \cup \{(u, v)\}$ that includes $A \cup \{(u, v)\}$.

Now we need to show that $T'$ not only is a spanning tree, but also a MST. Both $(x, y)$ and $(u, v)$ cross $(S, V \setminus S)$, but by hypothesis $(u, v)$ is a light-edge. This implies that $w(u, v) \leq w(x, y) \rightarrow w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$. Since $T$ is a MST by hypothesis, it follows that $w(T') = w(T)$.

## 4.3 Prim's alorithm

Prim's algorithm applies the Generic-MST in the following way:

**Algorithm 9** Prim

```
procedure PRIM $(G, s)$
    $X = \{s\}$
    $A = \emptyset$
    while There is an edge $(u, v)$ with $u \in X$ and $v \notin X$ do
        $(u^*, v^*)$ = a minimum weight such edge
        add $v^*$ to $X$
        add $(u^*, v^*)$ to $A$
    return $A$
```

This algorithm *grows* a spanning tree from a source vertex $s$ by adding an edge at a time.

- **Correctness:** it follows from the theorem
- **Complexity:** assume the $G$ is represented with an adjacency list, the complexity is $O(m \cdot n)$, which is polynomial time. Therefore, Prim's algorithm is an efficient algorithm.

# Chapter 5

# Lec 05 - Prim's & Kruskal's algorithms

## 5.1 Efficient Prim's algorithm

The *standard* implementation of Prim's algorithm has a complexity of $O(m \cdot n)$. For small graph it is an efficient algorithm, but it is not so efficient in very large graphs (e.g. Facebook graph).

In the basic implementation the calculation of a light-edge (i.e. the **minimum** cross edge with respect to the cut $(S, V \setminus S)$) is done repeatedly. If we are able to speed-up this computation, then the algorithm will be faster. This kind of optimization can be done using a **heap**. A heap is a data structure which supports the following operations:

- **insert:** add an object to the heap
- **extractMin:** remove the object with the smallest key (highest priority)
- **delete:** given a pointer to an object, remove it.

In a heap with $n$ objects, the complexity of these operations is $O(log(n))$

We can redefine the Prim's algorithm exploiting this efficient data structure:

**Algorithm 10** Prim

```
procedure PRIM $(G, s)$
    for $v \in V$ do
        Key$(v) = +\infty$
        $\Pi(v) = null$
    Key$(s) = 0$
    $H = V$
    while $H \neq \emptyset$ do
        $v^* =$ extractMin$(H)$
        for $v$ adjacent to $v^*$ do
            if $v \in H$ and $w(v^*, v) <$ Key$(v)$ then
                $\Pi(v) = v^*$
                Key$(v) = w(v^*, v)$    //Updating the heap
```

$H$ is the heap which contains all the vertices not in the tree. $\Pi(v)$ denotes the parent of $v$ in the tree.

**Complexity:** The algorithm performs an initialization, a while loop and a for loop. Let's see the complexity of each of these operations:

- **init:** $O(n)$
- **while:** $n$ iterations
- **extractMin:** $O(log(n))$

Total cost of extractMin → $O(n\,log(n))$

- **for loop:** executed $O(m)$ times in total:
  - $v \in H \rightarrow O(1)$
  - $\text{Key}(v) = w(v^*, v) \rightarrow O(log(n))$

Total cost of for loop → $O(m\,log(n))$.

Then, the total complexity of the algorithm is $O(n\,log(n) + m\,log(n))$. Since the graph is connected, it becomes $O(m\,log(n))$.

## 5.2 Kruskal's algorithm

Kruskal's algorithm is another algorithm for the MST problem which implements the Generic-MST algorithm. In particular:

- $A$ is a forest (set of trees)
- *safe edge* is a light-edge connecting two distinct components

**Algorithm 11** Kruskal

```
procedure KRUSKAL ($G$)
    $A = \emptyset$
    Sort edges of $G$ by weight
    for edge $e$ in non-decreasing order do
        if $A \cup \{e\}$ is acyclic then
            $A = A \cup \{e\}$
    return $A$
```

Note that:

- A source vertex is not needed
- We can perform a simple optimization, that is, stop the for loop when $A$ has $n-1$ edges.

**Correctness:** Follows from correctness of Generic-MST.

**Complexity:**

- **sorting:** $O(m\,log(n))$
- **for loop:** check whether $e = (u, v)$ closes a cycle, which has a complexity of $O(n)$ (DFS implementation).

Then, the total complexity is $O(m \cdot n)$.

# Chapter 6

# Lec 06 - Efficient Kruskal & Union-Find

## 6.1 Efficient Kruskal

In the Kruskal's algorithm the operation repeated many times is the cycle-checking when an edge is added to $A$. This operation can be speeded up by using a data structure called **Union-Find**. Union-Find is a data structure to merge disjoint set of objects. It implements the following operations:

- **Init:** Given an array $X$ of objects, it creates a Union-Find data structure with each object $x \in X$ in its own set.
- **Find:** Given an object $x$, it returns the name of the set that contains $x$.
- **Union:** Given two objects $x, y$, it merges the sets that contain $x$ and $y$ into a single set (if $x$ and $y$ are already in the same set, it does nothing).

These operations can be implemented with the following complexities:

- Init: $O(n)$
- Find: $O(log\, n)$
- Union: $O(log\, n)$

The idea of the efficient implementation of Kruskal's algorithm is to use Union-Find to keep track of the connected components of the solution. $A \cup \{(v, w)\}$ creates a cycle iff $v, w$ are already in the same connected components. Let's analyze the complexity of the algorithm:

**Algorithm 12** Kruskal

```
procedure Kruskal ($G$)
    $A = \emptyset$
    $U = \text{init}(V)$
    Sort edges of $G$ by weight
    for edge $e = (v, w)$ in non-decreasing order do
        if Find$(v) \neq$ Find$(w)$ then
            $A = A \cup \{(v, w)\}$
            Union$(v, w)$
    return $A$
```

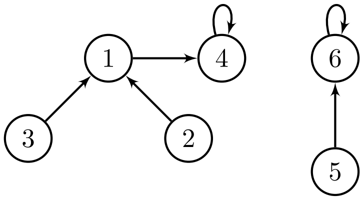
- Init: $O(n)$
- Sorting: $O(m\, log\, n)$
- Loop:
  - $2m$ Find: $O(m\, log\, n)$
  - $n-1$ Union: $O(n\, log\, n)$
  - Updating $A$: $O(n)$

The total complexity is $O(m\, log\, n)$.

## 6.2 Union-Find implementation

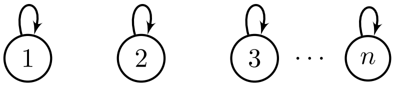
We'll use an array, which can be visualized as a set of **directed trees**. Each element of the array has a field $parent(x)$ that contains the index in the array of some object $y$.

| index of $x$ | $parent(x)$ |
|---|---|
| 1 | 4 |
| 2 | 1 |
| 3 | 1 |
| 4 | 4 |
| 5 | 6 |
| 6 | 6 |



This is called *parent graph.* The name of the parent graph has the name of the root.

- **Init**:



- **Find**: It follows parent by parent until the root is reached $parent \to parent \to ... \to root$.

  $find(x):$

  1. Starting from the position of $x$ in the array, traverse parent edges until reaching a position $j$ such that $parent(j) = j$.
  2. Return $j$

  **Definition:** The **depth** of an object $x$ is the number of edges traversed by $find(x)$.
  **Examples:**

  - $depth(4) = 0$

  - $depth(1) = 1$
  - $depth(2) = 2$

  The complexity of $find(x)$ is the largest depth of $x$, it depends on the Union implementation.

- **Union:** Given two objects $x, y$, the two trees of the parent graph containing $x$ and $y$ must be merged in a single tree. The simplest way is to point one of the two roots to another node of the other tree. However, this implementation is not smart since it produces **unbalanced** trees.

  In order to implement in an efficient way the Union operation, we need to decide two things:

  1. Which of the two roots remains a root.
  2. To which node should a root point.

  In order to have the minimum increasing in depth, a root must always point to the other root. For what concerning the point 1), there are two alternatives:

  - *Union-by-size*: It minimizes the number of objects whose depth increases.
  - *Union-by-rank*: The root of the less tall tree points to the tallest tree.

  We'll use the *Union-by-size* implementation.

**Algorithm 13** Union

```
procedure UNION $(x, y)$
    $i = \text{Find}(x)$
    $j = \text{Find}(y)$
    if $i = j$ then return
    if size$(i) \geq$ size$(j)$ then
        $parent(j) = i$
        size$(i)$ = size$(i)$ + size$(j)$
    else
        $parent(i) = j$
        size$(j)$ = size$(i)$ + size$(j)$
```

The complexity of Find and Union is $O(log\, n)$.

**Proof:**
Initially, $depth(x) = 0\ \forall x$. $depth(x)$ can only increase because of a union in which the root of the tree of $x$ points to another root. This happens only when the tree of $x$ gets merged to a tree of size not smaller. Therefore, when the depth of $x$ increases, the size of the tree of $x$ at least doubles. This can happen at most $log_2 n$ times.

# Chapter 7

# Lec 07 - Shortest Path

## 7.1 Definitions and Terminology

- Given a weighted graph, the **length** of a path $p = v_1, v_2, ..., v_k$ is defined as $len(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$
- A **shortest path** from a vertex $u$ to vertex $v$ is a path with minimum length among all $u - v$ paths.
- The **distance** between 2 vertices $s$ and $t$, denoted as $dist(s,t)$ is the length of a shortest path from $s$ to $t$. If there is no path at all from $s$ to $t$, then $dist(s,t) = \infty$

## 7.2 Problem: Shortest Path

Given a directed, weighted graph, a source vertex $s \in V$ and a destination $t \in V$, compute the shortest path from $s$ to $t$[1].

**Applications:**

- Road networks (Google maps)
- Routing in computer networks (e.g. internet)
- ...

## 7.3 Single-Source Shortest Paths (SSSP)

Description of the problem:

- **input:** A directed, weighted graph $G$ with edge weights $w : E \to \mathbb{R}$ and a source vertex $s \in V$
- **output:** $dist(s,v)\ \forall v \in V$

Observations:

- No algorithms are known that run asymptotically faster than the best SSSP algorithm in the worst case.
- We'll work with **directed** graphs, but all the algorithms that we'll see can be adapted easily for undirected graphs

---

[1] In directed graph, in general, $dist(u,v) \neq dist(v,u)$

### 7.3.1 Special case: non-negative edge weights

We've already solved a special case of this special case, that is, when all the weights are $w = 1$ it can be solved in linear time using BFS.

Can we modify BFS in order to deal also with weights greater that 1 ? A first simple solution can be to replace and edge with weight $w = n$ with $n$ edges of weight $w = 1$. However, with this implementation, the size of the transformed graph can be much bigger than the size of the starting graph and the complexity can grow exponentially.

**Intuition of a new algorithm:**

The arc $(s, v)$, at the first step, must be the shortest path from $s$ to $v$ since the first segment of **any other** path is already longer. A similar reasoning works in the next steps.

**Dijkstra algorithm:**

- **input:** directed $G$, $s \in V$, $w : E \to R_{>0}$
- **output:** $len(v) = dist(s, v)\ \forall v \in V$

**Algorithm 14** Dijkstra

```
procedure DIJKSTRA $(G, s)$
    X = $\{s\}$
    $len(s) = 0$
    $len(v) = \infty$
    while There is an edge $(v, w)$ with $v \in X$ and $w \notin X$ do
        $(v^*, w^*)$ = such an edge minimizing $len(v) + w(v, w)$
        add $w^*$ to X
        $len(w^*) = len(v^*) + w(v^*, w^*)$
```

At each iteration, $X$ contains the node whose shortest paths are known.

Dijkstra does **not** work on graphs with negative edge weights.

**Correctness:** The proof is performed by induction on $|X|$.

- **Invariant:** $\forall x \in X$, $len(x) = dist(s, x)$
- **Base case:** $|X| = 1$ it's trivial
- **Inductive hypothesis:** Invariant is true $\forall |X| = k \geq 1$

Let $v$ be the **next** vertex added to $X$ and $(u, v)$ the arc. The shortest path from $s$ to $u$ $+(u, v)$ is a path from $s$ to $v$ of length $\pi(v) = min_{(u,v):u \in X,\, v \notin X} len(u) + w(u, v)$. Let's consider any path $P$ from $s$ to $v$, we'll show that $P$ is not shorter than $\pi(v)$. Let $(x, y)$ be the first arc in $P$ that traverses $X$ and let $P^{'}$ be the sub-path from $s$ to $x$. Since all the weights are non-negative, it follows that:

$$len(P) \geq len(P^{'}) + w(x, y)$$

By Inductive hypothesis we know that $len(x) :\ x \in X = dist(s, x)$:

$$len(P^{'}) + w(x, y) \geq len(x) + w(x, y)$$

By definition of $\pi$ we obtain:

$$len(x) + w(x, y) \geq \pi(y)$$

But since the algorithm always selects the minimum edge:

$$\pi(y) \geq \pi(v)$$

**Complexity:** $O(m \cdot n)$. This is an inefficient implementation since it keeps searching the minimum edge checking all the edges $n$ times. It can be speeded up using heaps:

**Algorithm 15** Dijkstra-Heaps

```
procedure DIJKSTRA $(G, s)$
    $X = \emptyset$
    $H =$ empty heap
    $key(s) = 0$
    for $v \neq s$ do
        $key(v) = \infty$
    for $v \in V$ do
        insert $v$ into $H$
    while $H$ is not empty do
        $w^* =$ extractMin$(H)$
        add $w^*$ to $X$
        $len(w^*) = key(w^*)$
        for each edge $(w^*, y)$ such that $y \notin X$ do
            $key(y) = min(key(y), len(w^*) + w(w^*, y))$    //Update the heap
```

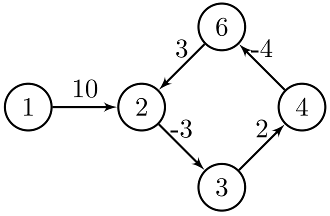
The complexity of this new implementation is $O((n + m) log\, n)$. There are $O(n + m)$ operations on the heap.

# Chapter 8

# Lec 08 - General SSSP Problem

## 8.1 General SSSP Problem

In the general formulation of the SSSP problem, the graphs can have edges with negative weights. With negative weights we must be careful about what we mean by shortest path. Let's consider the following graph:



This graph contains a **negative weights cycle**. Let's say that we want to compute the shortest path from 1 to 2. If we run Dijkstra algorithm, it will get stucked in the cycle forever. Basically $dist(1, 2) = -\infty$.

Therefore, we need to compute the shortest cycle-free/simple paths. Now the problem is well defined, but it is NP-Hard (problem for which we don't have any polynomial time algorithm).

So let's redefine the SSSP problem in a revised version:

- **input:** A directed weighted graph $G = (V, E)$ and a source vertex $s \in V$.
- **output:** One of the following:
  1. $dist(s, v)\ \forall v \in V$
  2. A declaration that $G$ contains a negative cycle.

Note that a shortest path **cannot** contain a cycle, except for 0-weight cycles, but in that case we can just remove all of them.

What needs to be changed in Dijkstra's algorithm in order to make it able to deal with negative-weights edges?

**Intuition:**

The problem in Dijkstra's algorithm is that it never updates its decisions. $len(v)$ should be an **estimated distance** and it needs to be updated for every vertex ($n - 1$ times because this is the maximum number of edges in a simple path between any two vertices).

### 8.1.1 Bellman-Ford algorithm

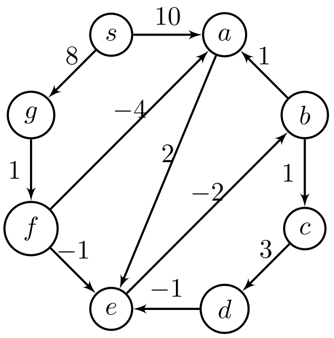
- **input:** A directed graph $G$ with edge weights $w : E \to \mathbb{R}$, and a source vertex $s \in V$.
- **output:** Either $dist(s, v)$ or a declaration that $G$ contains a negative cycle.

**Algorithm 16** Bellman-Ford

```
procedure BELLMAN-FORD ($G, s$)
    $len(s) = 0$
    $len(v) = \infty \ \forall v \neq s$
    for $n - 1$ iterations do
        for edge $(u, v) \in E$ do
            $len(v) = min(len(v), len(u) + w(u, v))$
    for edge $(u, v) \in E$ do
        if $len(v) > len(u) + w(u, v)$ then return $G$ contains a negative cycle
```

**Complexity:** $O(m \cdot n)$.

**Example:**



Iterations

Vertices

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | last |
|---|---|---|---|---|---|---|---|---|---|
| $s$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $a$ | $\infty$ | 10 | 10 | 5 | 5 | 5 | 5 | 5 | 5 |
| $b$ | $\infty$ | $\infty$ | $\infty$ | 10 | 6 | 5 | 5 | 5 | 5 |
| $c$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 11 | 7 | 6 | 6 | 6 |
| $d$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 14 | 10 | 9 | 9 |
| $e$ | $\infty$ | $\infty$ | 12 | 8 | 7 | 7 | 7 | 7 | 7 |
| $f$ | $\infty$ | $\infty$ | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| $g$ | $\infty$ | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |

**Correctness of the algorithm:**

Let $len(i, v)$ be the length of a shortest path from $s$ to $v$ that contains at most $i$ edges. Since the shortest path from $s$ to $v$ contains at most $n-1$ edges, it's sufficient to prove that after $i$ iterations $len(v) \leq len(i, v)$.

We'll prove it by induction on $i$:

- **Base case:** $i = 0$

$$len(s) = 0 \leq len(0, s) = 0$$

$$len(v \neq s) = +\infty = len(0, v \neq s)$$

- **Inductive hypothesis:** $len(v) \leq len(k,v)\ \forall\, 1 \leq k \leq i$

Take $i \geq 1$ and take a shortest path from $s$ to $v$ with at most $i$ edges. Let $(u,v)$ be the last edge of this path. Then:

$$len(i,v) = len(i-1,u) + w(u,v)$$

By inductive hypothesis, $len(u) \leq len(i-1,u)$. In the $i$-th iteration we update $len(v)$ as follows:

$$len(v) = min(len(v), len(u) + w(u,v))$$

$$len(u) + w(u,v) \leq len(i-1,u) + w(u,v) = len(i,v)$$

Therefore, $len(v) \leq len(i,v)$.

# Chapter 9

# Lec 09 - All-Pairs Shortest Paths (APSP)

## 9.1 All-Pairs Shortest Paths (APSP)

- **Input:** A directed, weighted graph $G = (V, E)$
- **Output:** One of the following:
  1. $dist(u, v)$ for every ordered vertex pair.
  2. A declaration that $G$ contains a negative cycle.

A first simple solution can be to invoke Bellman-Ford algorithm for every vertex. However, this solution is very inefficient since it has a complexity of $O(m \cdot n^2)$.

We can do better by exploiting dynamic programming.

### 9.1.1 Floyd-Warshall algorithm

Floyd-Warshall algorithm is an algorithm for APSP problem defined using dynamic programming. Let's define the sub-problems:

- Call the vertices $1, 2, ..., n$
- Compute $dist(u, v, k)$, that is, the length of a shortest path from $u$ to $v$ that uses only the vertices $\{1, 2, .., k\}$ as internal vertices [1] (excluded $u$ and $v$), and that does not contain a directed cycle. If no such path exists, define $dist(u, v, k) = +\infty$

The parameter $k$ defines the sub-problem size. In total, there are $O(n^3)$ sub-problems ($n$ choices for $u$, $n$ choices for $v$, $n$ choices for $k$).

The idea of the algorithm is to expand the set of allowed internal vertices, one vertex at time, until this set is $V$.

---

[1] it passes through only those vertices

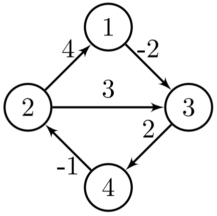
Note that there are only two candidates for the optimal solution to a sub-problem, depending on whether it uses vertex $k$ or not. If the sum of the distance between $u$ and the new node in question $k$ added to the distance between $k$ and $v$ is less than the distance directly between $u$ and $v$, then the latter is replaced with the previous sum (to go from node $u$ to node $v$ I should go through node $k$).

**Algorithm 17** Floyd-Warshall

```
procedure FLOYD-WARSHALL ($G$)
    Label the vertices $V = \{1, 2, 3, ..., n\}$ arbitrarily
    $A = n \times n \times (n + 1)$ array    //in which we store the solutions
    // Base case ($k = 0$)
    for $u = 1$ to $n$ do
        for $v = 1$ to $n$ do
            if $u = v$ then
                $A[u, v, 0] = 0$
            else if $(u, v) \in E$ then
                $A[u, v, 0] = w(u, v)$
            else
                $A[u, v, 0] = +\infty$
    // Solve sub-problems
    for $k = 1$ to $n$ do
        for $u = 1$ to $n$ do
            for $v = 1$ to $n$ do
                $A[u, v, k] = min(A[u, v, k - 1], A[u, k, k - 1] + A[k, v, k - 1])$
    for $u = 1$ to $n$ do
        if $A[u, u, n] < 0$ then return $G$ contains a negative cycle
```

Since we have constant work ($O(1)$) per sub-problem, the complexity of the algorithm is $O(n^3)$.

**Example of execution:**



| $k = 0$ | | $v$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $u$ | **1** | 0 | $\infty$ | -2 | $\infty$ |
| | **2** | 4 | 0 | 3 | $\infty$ |
| | **3** | $\infty$ | $\infty$ | 0 | 2 |
| | **4** | $\infty$ | -1 | $\infty$ | 0 |

| $k = 1$ | | $v$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $u$ | **1** | 0 | $\infty$ | -2 | $\infty$ |
| | **2** | 4 | 0 | **2** | $\infty$ |
| | **3** | $\infty$ | $\infty$ | 0 | 2 |
| | **4** | $\infty$ | -1 | $\infty$ | 0 |

| $k = 2$ | | $v$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $u$ | **1** | 0 | $\infty$ | -2 | $\infty$ |
| | **2** | 4 | 0 | 2 | $\infty$ |
| | **3** | $\infty$ | $\infty$ | 0 | 2 |
| | **4** | **3** | -1 | **1** | 0 |

| $k = 3$ | | $v$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $u$ | **1** | 0 | $\infty$ | -2 | **0** |
| | **2** | 4 | 0 | 2 | **4** |
| | **3** | $\infty$ | $\infty$ | 0 | 2 |
| | **4** | 3 | -1 | 1 | 0 |

| $k = 4$ | | $v$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $u$ | **1** | 0 | **-1** | -2 | 0 |
| | **2** | 4 | 0 | 2 | 4 |
| | **3** | **5** | **1** | 0 | 2 |
| | **4** | 3 | -1 | 1 | 0 |

## 9.2 Bellman-Ford algorithm via dynamic programming

The idea is to introduce a parameter $i$ that restricts the **maximum number of edges** allowed in a path, with smaller sub-problems having smaller edge budgets.

**Sub-problems:**
Compute $len(i, v)$, that is, the length of a shortest path from $s$ to $v$ that contains at most $i$ edges. If no such path exists, define $len(i, v) = +\infty$. There are $O(n^2)$ sub-problems in total. Note that every sub-program works with the full input, the idea is to control the allowable size of the output.

**Bellman-Ford recurrence:**

$$len(i,v) = \left\{ \begin{array}{ll} 0 & i = 0 \text{ and } v = s \\ +\infty, & i = 0 \text{ and } v \neq s \\ min \left\{ \begin{array}{l} len(i-1,v) \\ min_{(u,v)\in E}(len(i-1,u) + w(u,v)) \end{array} \right\} & \text{otherwise} \end{array} \right\}$$

This formulation can be adopted to APSP and obtain an algorithm with complexity $O(n^3 log\, n)$

# Chapter 10

# Lec 10 - Maximum Flows

## 10.1 Definitions

A **flow network** is a directed graph $G = (V, E)$ where each edge has a **capacity** $c(e) \in \mathbb{R}^+$, along with a designated **source** $s \in V$ and **sink** $t \in V$. For convenience, write $c(e) = 0$ if $e \notin E$. We assume that there are no edges entering $s$ and no edges leaving $t$.

A **flow** is a function $f : E \to \mathbb{R}^+$ satisfying the following constraints:

- capacity: $\forall e \in E\ f(e) \leq c(e)$
- conservation: $\forall u \in V \setminus \{s, t\}$ we have $\sum_{v \in V\ s.t.\ (v,u) \in E} f(v, u) = \sum_{v \in V\ s.t.\ (u,v) \in E} f(u, v)$. Basically, the flow that *enters* in a node must be equal to the flow that *goes out* from that node.

The **value** of a flow is $|f| = \sum_{v \in V\ s.t.\ (s,v) \in E} f(s, v)$ that is, the total flow out of the source.

## 10.2 Maximum flow problem

Given a flow network, find a flow of **maximum value**.

**Applications:**

- Rail networks
- Road networks
- Electrical networks
- ...

Max flow reduces to linear programming (like many other problems), but there are more efficient algorithms. We'll see one of them (Ford-Fulkerson), but there are plenty of more efficient algorithms.

A first simple algorithm can be the following:

1. Find a path from $s$ to $t$ (in linear time using BFS)
2. Find the *bottleneck* edge (i.e. the edge with smallest capacity in the path) and send as much flow along the path as possible.
3. Update capacities

4. Remove edges that have 0 remaining capacity
5. Repeat until the graph has no more $s - t$ paths

Actually, this algorithm will **not** always work. We can improve it by exploiting the following idea: revise/undo some of the flow later in the algorithm by *pushing back* some flow through new edges in the reverse direction.

## 10.3 Ford-Fulkerson method

**Definition:**
Given a flow network $G$ and a flow $f$, the **residual network** of $G$ with respect to $f$, $G_f$, is a network with vertex set $V$ and edge set $E_r$ as follows:
For every edge $e = (u, v)$ in $G$:

1. if $f(e) < c(e)$, add $e$ to $G_f$ with capacity $c_f(e) = c(e) - f(e)$
2. if $f(e) > 0$, add another edge $(v, u)$ to $G_f$ with capacity $c_f(e) = f(e)$

Intuitively, the residual network $G_f$ consists of edges with capacities that represent how we can change the flow on edges of $G$.

The residual network $G_f$ may also contain edges that are not in $G$. As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge. In order to represent a possible decrease of a positive flow $f(u, v)$ on an edge in $G$, we place an edge $v, u$ into $G_f$ with residual capacity $c_f(v, u) = f(u, v)$. Sending flow back along an edge is equivalent to *decreasing* the flow on the edge.

The Ford-Fulkerson method repeatedly finds an $s - t$ path $P$ in $G_f$ (e.g. using BFS) and uses $P$ to **increase** the current flow. $P$ is called **augmenting path**.

**Algorithm 18** Ford-Fulkerson

```
procedure FORD-FULKERSON($G, s, t$)
    initialize the $f(e) = 0\ \forall e \in G.E$
    $G_f = G$
    while There exists an augmenting path $P$ in $G_f$ do
        $\Delta p = min_{e \in P}(c_f(e))$   //$\Delta p$ is the bottleneck capacity in $P$
        for each edge $e = (u, v) \in P$ do
            if $(u, v) \in G.E$ then
                $f(u, v) = f(u, v) + \Delta p$
            else
                $f(v, u) = f(v, u) - \Delta p$   //if the edge $(u, v) \notin G.E$, the opposite edge $(v, u)$ must be in $E$
        Update the residual graph $G_f$
```

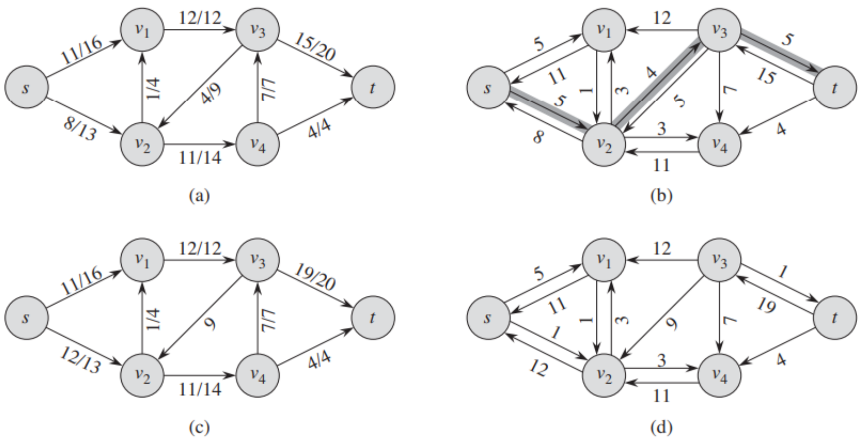


**Figure 26.4** **(a)** The flow network $G$ and flow $f$ of Figure 26.1(b). **(b)** The residual network $G_f$ with augmenting path $p$ shaded; its residual capacity is $c_f(p) = c_f(v_2, v_3) = 4$. Edges with residual capacity equal to 0, such as $(v_1, v_3)$, are not shown, a convention we follow in the remainder of this section. **(c)** The flow in $G$ that results from augmenting along path $p$ by its residual capacity 4. Edges carrying no flow, such as $(v_3, v_2)$, are labeled only by their capacity, another convention we follow throughout. **(d)** The residual network induced by the flow in (c).

Basically, when we find an augmenting path $P$:

- if an edge $(u, v) \in E$, we are increasing its flow by the bottleneck capacity
- if an edge $(u, v) \notin E$, it's like we are decreasing the flow on the edge $(v, u)$ and adding that *amount* of flow to other edges

**Complexity:**

- The flow value increases by $\geq 1$ at each iteration.
- The complexity of each iteration is $O(m)$.

The total complexity is $O(m \cdot |f^*|)$ where $|f^*|$ is a max flow.

# Chapter 11

# Lec 11 - NP-Hardness

## 11.1 Tractable vs Intractable problems

Problems for which a polynomial algorithm exists are called **tractable**. If no such algorithm exists, the problem is called **intractable**.

**Examples:**

1. **Eulerian circuit problem:** Given an undirected graph, an Eulerian circuit is a cycle that traverses all the edges only once. This problem can be solved in linear time. Note that in an Euler circuit you might pass through a vertex more than once.
2. **Hamiltonian circuit problem:** Given an undirected graph, an Hamiltonian circuit is a cycle that traverses all the vertices only once. To date, no one knows a polynomial algorithm to solve it. Note that in a Hamiltonian circuit you may not pass through all edges.
3. **Minimum spanning tree:**
4. **Traveling Salesperson Problem (TSP):** Given a complete, undirected graph and a function $w : E \to \mathbb{R}$, output a **tour** (i.e. a cycle that visits every vertex exactly once) of minimum cost

   $$T \subseteq E \quad \text{minimizing} \sum_{e \in T} w(e)$$

   To date, no one knows a polynomial algorithm to solve it.

A much easier task can be the following: Given a graph and a list of vertices $C$, **check** if $C$ is an Hamiltonian circuit.

Problems that are *easy* to solve belong to the **complexity class P** (*polynomial time*), 1), 3) $\in$ **P**. Problems that are *easy* to verify belong to the complexity class **NP** (*Non-deterministic polynomial*), 1), 2) ,3) ,4) $\in$ **NP**.

## 11.2 NP-Hard

**Decision Problems:**
To simplify the study of the complexity of problems, we limit our attention to problems with a boolean answer (YES, NO), that is, decision problems.

Let's define **P** and **NP** classes in a more formal way:

- **P** is the set of decision problems that can be solved in polynomial time.
- **NP** is the set of decision problems with the following property: if the answer is YES, then there is a proof of this fact that can be checked in polynomial time.
- **NP-Hard:** A computational problem is **NP-Hard** if a polynomial-time algorithm that solves it would imply a polynomial-time algorithm that solves every problem in **NP**.

  A problem is **NP-Complete** if it is both in **NP** and **NP-Hard**. Basically, these problems are the hardest in **NP**.

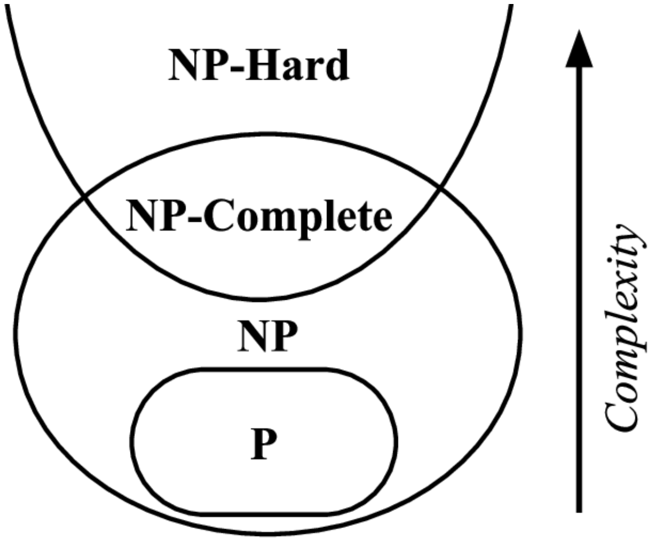


Figure 11.1: What we think the computation classes look like

One of the main open questions in computer science is whether **P=NP**.

Studying NP-Hardness is important because if a problem is **NP-Hard** is a strong evidence that it is intractable. It suggests you to use a different approach, such as identify tractable special-cases, or use **approximation algorithms**.

## 11.3 Cook-Levin Theorem

*In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is* ***NP-complete****.*[1]

SAT: formula satisfiability:

- input: A boolean formula like the following: $(b \wedge \neg c) \vee (\neg a \wedge b)$
- output: It is possible to assign boolean values to the variables $a, b, c, ...$ such that the entire formula evaluates to TRUE?

---

[1] Wikipedia definition

determining the satisfiability of a formula in conjunctive normal form (CNF) where each clause is limited to at most three literals is **NP-complete** also; this problem is called **3-SAT**.

**Example of 3-CNF formula:**

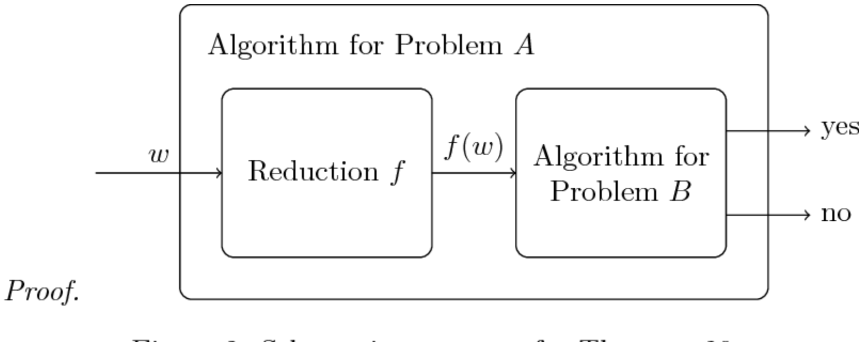
$$(a \lor b \lor c) \land (b \lor \neg c \lor \neg d) \land (\neg a \lor c \lor d)$$

How can we show that a problem is **NP-Hard**?

## 11.4 Reductions

To prove that a problem $P$ is **NP-Hard** we use **reduction**. We want to prove that every problem in **NP** reduces to problem $P$.

Reducing problem $A$ to problem $B$ means describing an algorithm to solve $A$ under the assumption that an algorithm for $B$ exists.


Algorithm for Problem $A$
$w$
Reduction $f$
$f(w)$
Algorithm for Problem $B$
yes
no
*Proof.*

$A < B$ means $A$ *reduces to* $B$ where $B$ is the hardest problem ($A$ is as hard as $B$).

# Chapter 12

# Lec 12 - Reduction

## 12.1 NP-Hardness (formal definition)

**Definition:**
A problem $A$ **reduces** in polynomial time to problem $B$ ($A <_p B$) if exists a polynomial time algorithm that transforms an arbitrary input instance $a$ of $A$ into an input instance $b$ of $B$ such that:

1. if $a$ is a YES instance of $A$, then $b$ is a YES instance of $B$
2. if $b$ is a YES instance of $B$, then $a$ is a YES instance of $A$

**Property:**
$A <_p B$ and $B <_p C \rightarrow A <_p C$

Now we can give a more formal definition of **NP-Hardness**:
A problem is **NP-Hard** if every problem in **NP** reduces in polynomial time to it.

Then, to prove that a problem $X$ is **NP-Hard**, we can reduce a known **NP-Hard** problem $Y$ (e.g. 3SAT) to $X$. Note that NP-Hardness **does not** mean that the problem is not in $P$, it provides a strong evidence for that.

## 12.2 NP-Hardness proof

**Theorem:** TSP is **NP-Hard**

**Proof:**
Reduction from Hamiltonian circuit to TSP ($Ham. <_p TSP$). Since TSP is not a decision problem, we have to redefine it.

**TSP:**

- **input:** $G = (V, E)$ complete, undirected, weighted graph, $k \in \mathbb{R}$
- **output:** $\exists$ in $G$ a Hamiltonian circuit of cost at most $k$ ($\leq k$).

Pick an arbitrary input instance for Ham. and create the following input for TSP: $G'(V, E')$ complete, undirected, weighted graph with:

$$w(e \in E') = \left\{ \begin{array}{cc} 1 & \text{if } e \in E \\ \infty & \text{otherwise} \end{array} \right\}$$

If we set $k = n$ the following holds:

1. if $G$ has an Hamiltonian circuit, then the TSP algorithm run on $G'$ returns an Hamiltonian circuit of cost $n$

2. if $G$ doesn't have an Hamiltonian circuit, then any Hamiltonian circuit in $G'$ must have at least one edge not in $G$, hence of weight $\infty$. In this case a TSP algorithm run on $G'$ returns an Hamiltonian circuit of cost $> n$.

If we had a fast algorithm for TSP we would solve also the Hamiltonian circuit problem.

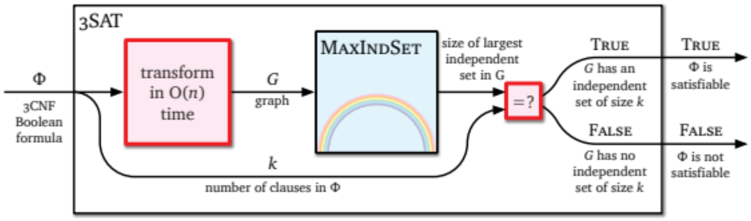
## 12.3 Maximum Independent Set

**Independent set:** Given a graph $G = (V, E)$, an independent set in $G$ is a set $I \subseteq V$ with no edges between them.

**Maximum independent set problem:** compute an independent set of maximum size.

**Theorem:** Maximum Independent set is **NP-hard**

**Proof:**
Reduction from 3SAT to maximum independent set.


3SAT
Φ
3CNF Boolean formula
transform in O(n) time
G
graph
MaxIndSet
size of largest independent set in G
=?
k
number of clauses in Φ
True
G has an independent set of size k
False
G has no independent set of size k
True
Φ is satisfiable
False
Φ is not satisfiable

Pick an arbitrary 3CNF boolean formula $f$ with $k$ clauses:

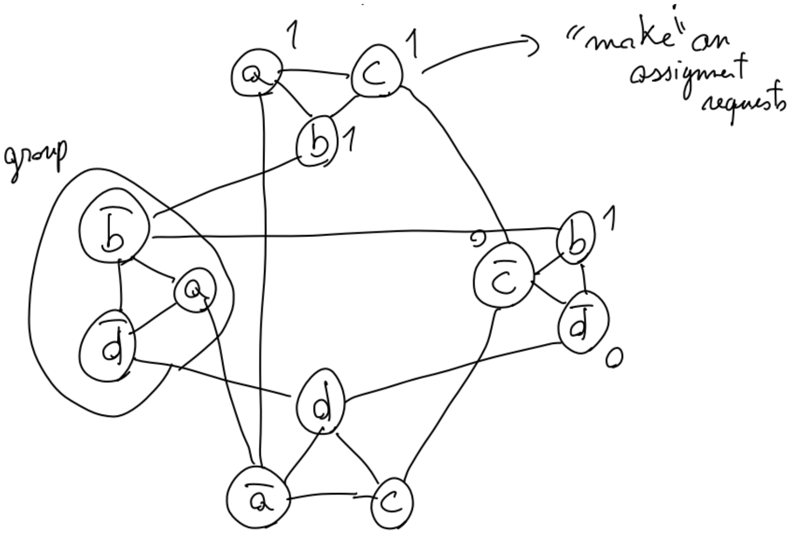
$$(a \vee b \vee c) \wedge (b \vee \neg c \vee \neg d) \wedge (\neg a \vee c \vee d) \wedge (a \vee \neg b \vee \neg d)$$

We define a graph starting from $f$:

- Vertices: Each vertex represents one literal in $f$. A *group* of 3 vertices represents a clause.

- Edges: Edges is $E$ are of two types:

  1. We add an edge between a literal and its inverse, for all the literals.
  2. We add an edge between every pair of vertices that are in the same group.



Suppose we assign the value true to each literal, then, an edge connects two vertices that are either in the same group or that are *asking* for inconsistent assignment for the same variable. Therefore, an independent set is forced to select at most one vertex per group (the largest independent set in G has size at most $k$). If $G$ contains an independent set of size exactly $k$ (number of clauses), the formula $f$ is satisfiable.

**Formal proof:**
**Claim** : $G$ contains an independent set of size exactly $k \iff$ the formula $f$ is satisfiable.

**Proof:**

- $\Rightarrow$: Suppose $f$ is satisfiable. Fix an arbitrary satisfying assignment. By definition, each clause contains at least one true literal.Choose a subset $S$ of $k$ vertices in $G$ that contains exactly one vertex per group, such that the corresponding $k$ literals are all true. Then, $S$ is an independent set of size $k$ because it does not contain both endpoints of any edge of a group, nor of any edge that connects inconsistent literals.
- $\Leftarrow$: Suppose $G$ contains an independent set $S$ of size $k$. Each vertex in $S$ must lie in a different group. Suppose we assign the value true to each literal in $S$. Since inconsistent literals are connected by an edge , this assignment is consistent. Since $S$ contains one vertex per group, each clause in $f$ contains (at least) one true literal. This implies that $f$ is satisfiable.

# Chapter 13

# Lec 13 - Approximation Algorithms

## 13.1 Approximation Algorithms

- **Optimization problem:**
$$\Pi : I \times S$$
where $I$ is the set of input and $S$ is the set of the solutions.
- **Cost function:**
$$c : S \to \mathbb{R}^+$$
the cost function $c$ maps each solution to a real number.
- **Set of feasible solutions:**
$$\forall i \in I \ S(i) = \{s \in S : i\Pi_s\}$$
The best solution $s^*$ for a minimization (maximization) problem is defined as follows:
$$s^* \in S(i) \text{ and } c(s^*) = min\{c(S(i))\}$$

### 13.1.1 Approximation

Given a feasible solution $s \in S(i)$, with $s \neq s^*$, we want:

1. guarantee on the *quality* of $s$
2. guarantee on the complexity: Polynomial-time algorithm

**Definition:** Let $\Pi$ be an optimization problem, and let $A_\Pi$ an algorithm for $\Pi$ that returns , $\forall i \in I$, $A_\Pi(i) \in S(i)$. We say that $A_\Pi$ has an **approximation factor** of $\rho(n)$ if $\forall i \in I$ such that $|i| = n$, we have

- minimization problem:
$$\frac{c(A_\Pi(i))}{c(s^*(i))} \leq \rho(n)$$
- maximization problem:
$$\frac{c(s^*(i))}{c(A_\Pi(i))} \leq \rho(n)$$

we assume that $c$ maps each feasible solution to a real number $\geq 1$.

The goal is to have an algorithm such that $\rho(n) = 1 + \epsilon$ with $\epsilon$ as small as possible. We'll get:

- $\epsilon = 1$ for the vertex cover problem.
- $\epsilon = log_2 n$ for the set cover problem.

when we have an approximation algorithm with an approximation factor of 2 it is called **2-approximation algorithm**

There exists problems for which we can prove that $\rho(n) = \Omega(n^\epsilon)$ (not smaller than $n^\epsilon$) $\forall \epsilon < 1$ (e.g. clique).

**Definition:** An **approximation scheme** for $\Pi$ is an algorithm with 2 inputs $A_\Pi(i, \epsilon)$, that for all $\epsilon$ is a $(1 + \epsilon)$-approximation algorithm. In this case we just have to choose *how much approximation* we want by tuning the value of $\epsilon$.

**Definition:** An approximation scheme is **polynomial** (PTAS) if $A_\Pi(i, \epsilon)$ is polynomial in $|i|$ $\forall \epsilon$ fixed. The smaller the value of $\epsilon$, the longer will take the computation (but still polynomial).

## 13.2 Approximation Algorithm for Vertex Cover

**Vertex Cover:** A vertex cover is a set of vertices that includes at least one endpoint of every edge of the graph.

**Minimum vertex cover problem:** Compute the smallest vertex cover in a given graph.

The first simple idea for a greedy approximation algorithm can be the following:

1. Select the vertex with highest degree
2. Remove the *touched* edges
3. Repeat

Unfortunately, it can be proved that for this algorithm $\rho(n) = \Omega(log\, n)$.

Let's try a new greedy approach:

1. Choose any edge
2. Add its endpoints to the solution
3. Remove all the touched edges
4. Repeat

We'll show that this is a 2-approximation algorithm.

**Algorithm 19** Approx_Vertex_Cover

```
procedure APPROX_VERTEX_COVER($G$)
    $V' = \emptyset$
    $E' = E$
    while $E' \neq \emptyset$ do
        Let $(u, v)$ be an arbitrary edge of $E'$
        $V' = V' \cup \{u, v\}$
        $E' = E' \setminus \{(u, z), (v, w)\}$    // remove the edges that have $u$ and $v$ as endpoints
    return $V'$
```

**Complexity:** The complexity of the algorithm is $O(n + m)$

### 13.2.1 Analysis

We'll prove that *Approx_Vertex_Cover* is a 2-approximation algorithm. In order to do that, we show that:

$$\frac{|V'|}{|V^*|} \leq 2$$

**Proof:**
Let $A$ be the set of selected edges [1]. $A$ is a matching: $\forall e, e' \in A \Rightarrow e \cap e' = \emptyset$, that is, there are no vertices in common among the edges in $A$.

*Approx_Vertex_Cover* selects a **maximal** matching, that is, $\forall$ edge $y$, $A \cup y$ is not a matching [2].

We can observe that:

1. $|V^*| \geq |A|$. Because $A$ is a matching, and in $V^*$ there must be at least one vertex for each edge in $A$.
2. $|V'| = 2|A|$ by construction.

Then, $|V'| \leq 2|A| \leq 2|V^*|$. This implies that:

$$\frac{|V'|}{|V^*|} \leq 2$$

---

[1] first instruction in the while loop

[2] Note that maximal $\neq$ maximum

# Chapter 14

# Lec 14 -TSP & Metric TSP

## 14.1 Travelling Salesperson Problem

Given a complete, undirected graph and a function $w : E \to \mathbb{R}^+$, output a **tour** (i.e. a cycle that visits every vertex exactly once)

$$T \subseteq E \quad \text{minimizing} \sum_{e \in T} w(e)$$

To date, no one knows a polynomial algorithm to solve it.

We'll work only with positive edges, that is, $w : E \to \mathbb{R}^+$. We can do this without loss of generality because every TSP tour has the same number of edges. Then, we can add a large weight to each edge, such that edges have non-negative weights.

**Theorem:** For any function $\rho(n)$ that can be computed in polynomial time in $n$, there is no polynomial time $\rho(n)$-approximation algorithm for TSP (unless $\mathbf{P} = \mathbf{NP}$).

**Proof:** The proof is by contradiction. Suppose to the contrary that for some number $\rho \geq 1$, there is a polynomial-time approximation algorithm $A$ with approximation ratio $\rho$. We shall then show how to use $A$ to solve instances of the Hamiltonian cycle problem in polynomial time. Since Hamiltonian circuit problem is **NP-Hard**, if we can solve it in polynomial time, then $\mathbf{P} = \mathbf{NP}$.

Given $n = |V|$, build a reduction from Hamiltonian Circuit:

- $G \to G' = (V, E')$ complete
- $w(e \in E') = \begin{cases} 1 & e \in E \\ \rho n + 1 & \text{otherwise} \end{cases}$

1. if $G$ has an Hamiltonian circuit, the cost of an optimal TSP solution in $G'$ is $n$.
2. if $G$ has not an Hamiltonian circuit, then any tour of $G'$ must use some edge not in $E$. In this case, the cost of an optimal TSP solution in $G'$ is at least:

$$(\rho n + 1) + (n - 1) = \rho n + n > \rho n$$

Note that there is a gap of at least $\rho n$ between the cost of a tour that is a Hamiltonian cycle in G (cost $n$) and the cost of any other tour (cost at least $\rho n + n$).

Now, suppose that we apply the approximation algorithm $A$ on $G'$. Because $A$ is guaranteed to return a tour of cost no more than $\rho$ times the cost of an optimal tour, if $G$ contains a Hamiltonian cycle, then $A$ must return it. If $G$ has no Hamiltonian cycle, then $A$ returns a tour of cost more than $\rho n$. Therefore, we can use $A$ to solve the Hamiltonian-cycle problem in polynomial time.

## 14.2 Metric TSP

**Metric TSP** is a special case of TSP where the weight function $w$ satisfies the triangle inequality:

$$\forall\, u, v, z \in V, \text{ it holds that } w(u,v) \leq w(u,z) + w(z,v)$$

The path from $u$ to $v$ is shorter than any other path that passes through another vertex.

**Theorem:** Metric TSP is **NP-Hard**.

**Proof:** TSP $<_p$ Metric TSP.
Given an instance of the TSP problem $< G = (V,E), w, k >$, we build an instance of Metric TSP $< G' = (V,E), w', k' >$ such that the triangle inequality is satisfied in $G'$. In order to to this, we can define the weight function $w'$ as follows:

$$w'(u,v) = w(u,v) + W$$

where $W = max_{u,v \in V}\{w(u,v)\}$.

We need to prove 2 thing:

1. the new definition of $w'$ satisfies triangle inequality:

   Let $u, v, z$ be vertices. Then we have $w'(u,v) = w(u,v) + W \leq 2W \leq w(u,z) + W + w(z,v) + W$. Note that it's important that the weights of edges are non-negative.

2. $\exists$ a Hamiltonian circuit of cost $k$ in $G \iff \exists$ a Hamiltonian circuit of cost $k'$ in $G'$:

   - $\Rightarrow$ Assume that exists a Hamiltonian circuit of cost $k$ in $G$. Note that an optimal solution contains exactly $n$ edges. The same circuit in $G'$ introduces a $+W\ \forall\, edge$. Thus, the cost of the tour in $G'$ is $k + nW$.
   - $\Leftarrow$ Just remove the $+W\ \forall\, edge$ to obtain a Hamiltonian circuit of cost $k$ in $G$.

# Chapter 15

# Lec 15 - 2-Approximation Algorithm for Metric TSP

## 15.1 Metric TSP: A 2-Approximation Algorithm

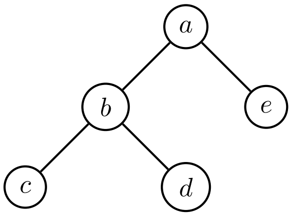
For the Vertex Cover problem we used the concept of maximal matching to find a 2-approximation algorithm, for Metric TSP we'll use **Minimum Spanning Tree**. The total MST weight gives a lower bound on the length of an optimal traveling-salesman tour. We shall then use the minimum spanning tree to create a tour whose cost is no more than twice of the minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality.

**Intuition:** First of all, we need to transform the tree in a cycle. In order to do this, we can use the **Preorder** visit of the MST.

**Algorithm 20** PREORDER

```
procedure PREORDER($T, v$)
    visit($v$)
    if $v$ is internal then
        for each $u \in$ children($v$) do
            PREORDER($T, u$)
    return
```

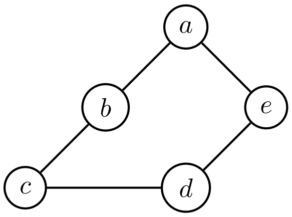
**Example:**



**Preorder** list: $a, b, c, d, e$.

Note that the result is **not** a cycle since $e$ and $a$ are not connected. We can simply solve this problem by adding the edge $(e, a)$ to the Preorder list and make it a Hamiltonian circuit. We are free to add every edge we want because the graph is complete by definition.

The resulting cycle of the MST above is the following:



Now we can define an algorithm that exploits this idea:

**Algorithm 21** Approx_Metric_TSP

```
procedure APPROX_METRIC_TSP($G$)
    $V = \{v_1, v_2, ..., v_n\}$
    $r = v_1$    // root from which Prim is run
    $T^* = Prim(G, r)$
    $< v_{i_1}, v_{i_2}, ..., v_{i_n} >= H' = PREORDER(T^*, r)$
    return $< H', v_{i_1} >= H$    // close the cycle
```

This algorithm uses Prim as a subroutine to compute the MST.

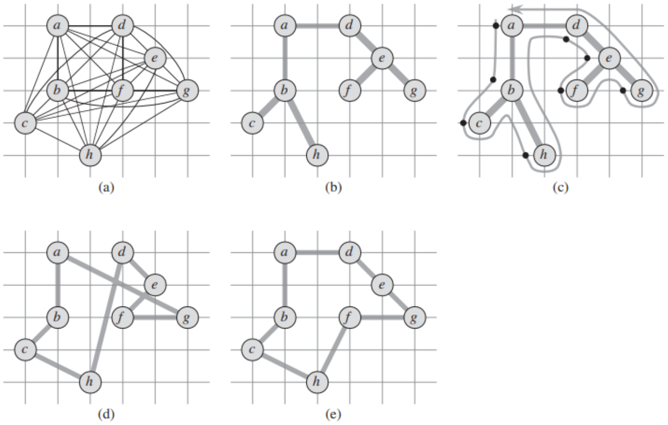


**Figure 35.2** The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, $f$ is one unit to the right and two units up from $h$. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning tree $T$ of the complete graph, as computed by MST-PRIM. Vertex $a$ is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of $T$, starting at $a$. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of $T$ lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering $a, b, c, h, d, e, f, g$. **(d)** A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour $H$ returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour $H^*$ for the original complete graph. Its total cost is approximately 14.715.

## 15.1.1 Analysis of the cost of $H$

Let $H^*$ denote an optimal tour for the given set of vertices.

1. The cost of $T^*$ is a lower bound to the cost of $H^*$.

   $$w(H^*) \geq w(T^*)$$

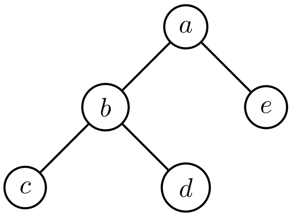
   This is because we obtain a spanning tree by deleting any edge from a tour, and each edge cost is non-negative. Therefore, the weight of the minimum spanning tree $T^*$ provides a lower bound to the cost of an optimal tour.

2. Then we have to find an upper bound to the cost of $H$ (the returned solution). We want to prove that:

   $$w(H) \leq 2w(T^*) \leq 2w(H^*)$$

   **Definition:** given a tree, a **full preorder chain** is a list with repetitions of the vertices of the tree which identifies the vertices reached from the recursive calls of $PREORDER(T, u)$.

   **Example**:

   **full preorder chain:** $a, b, c, b, d, b, a, e, a$

   Since the full preorder chain traverses every edge of $T^*$ exactly twice, we have:

   $$w(f.p.c) = 2w(T^*) \leq 2w(H^*)$$

   Unfortunately, the f.p.c is generally not a tour, since it visits some vertices more than once. By **the triangle inequality**, however, we can delete a visit to any vertex from f.p.c and the cost does not increase. By repeatedly applying this operation, we can remove from f.p.c. all but the first visit to each vertex (except for the last occurrence of the root). This is like adding a *shortcut* between vertices that does not increase the cost. In our example:

   $$2w(T^*) = w(< a, b, c, b, d, b, a, e, a >)$$
   $$\geq w(< a, b, c, d, b, a, e, a >)$$
   $$...$$
   $$\geq w(< a, b, c, d, e, a >)$$

   Then, we have that:

   $$2w(T^*) \geq w(H)$$

**Putting all together:**

1. $w(H^*) \geq w(T^*)$
2. $2w(T^*) \geq w(H)$

This implies that:

$$2w(H^*) \geq 2w(T^*) \geq w(H)$$
$$\frac{w(H)}{w(H^*)} \leq 2$$

# Chapter 16

# Lec 16 - A 1.5-Approximation Algorithm for Metric TSP

## 16.1 Christofides' algorithm

The *APPROX_METRIC_TSP* algorithm has a 2-approximation factor because the full Preorder chain of $T^*$ uses every edge of $T^*$ twice. We'll try to improve on this by constructing a tour that traverses MST edges **only once**.

**Definition:** A path (or cycle) is **Eulerian** if it visits every edge of the graph exactly once.

**Definition**: A connected graph is Eulerian if it exists an Eulerian cycle.

If the MST was Eulerian (cannot be), we would have a 1-approximation algorithm. Our idea is to build a *cheap* Eulerian cycle in the MST.

**Theorem:** A connected graph is Eulerian $\iff$ every vertex has even degree [1].

So, let's handle the **odd degree** vertices of the MST explicitly.

**Property:** In any (finite) graph the number of vertices of odd degree is even.

**Proof:** The proof comes from the following equality:

$$\sum_{v \in V} deg(v) = 2m$$

Note that we can split the summation in two parts:

$$\sum_{u \in even} deg(u) + \sum_{w \in odd} deg(w) = 2m$$

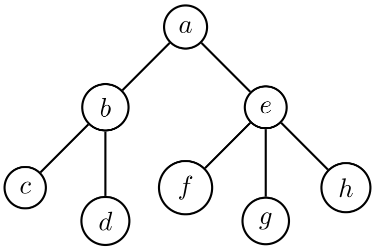
Since the result must be even, $\sum_{w \in odd} deg(w)$ must be even. But this happens only if the number of odd degree vertices is even.

**Idea:** Augment the initial MST $T^*$ with a minimum-weight **perfect matching** between the vertices that have odd degree in the MST (perfect means that it includes all such vertices).

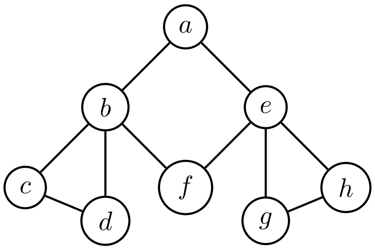
---

[1] if i want to traverse every edge exactly once, i must have a *way out* from every vertex

For example, let's consider the following MST $T^*$:



In this example the odd-degree vertices are $b, c, d, f, g, h$. If we augment $T^*$ with a minimum-weight perfect matching, it becomes as follows [2]:



The resulting graph has only even-degree vertices, that is, Eulerian graph. Now that we have an Eulerian graph, we can approximate a Metric TSP tour by finding an Eulerian cycle in this graph adding shortcuts when needed (as we did with *APPROX_METRIC_TSP*).

**Algorithm 22** Christofides' algorithm

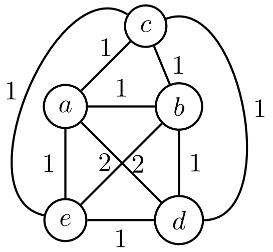
```
procedure CHRISTOFIDES($G$)
    $T^* = Prim(G, r)$    $//T^* = (V, E^*)$
    $D$ = set of vertices of $T^*$ with odd degree
    $M^*$ = a min-weight perfect matching on the graph iduced by $D$    // can be done in poly-time
    $G^* = (V, E^*) \cup M^*$    // this is an Eulerian graph
    $E$ = an Eulerian cycle on $G^*$
    return the cycle that visits all the vertices of $G$ in the order of their first appearance in $E$
```

Just to summarize: The idea of *APPROX_METRIC_TSP* is to approximate an optimal Metric TSP tour by using the Preorder traversal of the MST $T^*$. As we have seen, this kind of traversal is like passing through every edge twice, but thanks to triangle inequality we are able to add shortcuts that do not increase the cost. Christofides' algorithm, instead, exploits the concept of Eulerian cycle. It creates an Eulerian graph starting from the MST $T^*$, and then it approximates an optimal Metric TSP tour by finding an Eulerian cycle in this new graph. Finally, it adds shortcuts in order to make the Eulerian cycle (that can pass through the same vertex more the once) an Hamiltonian circuit.

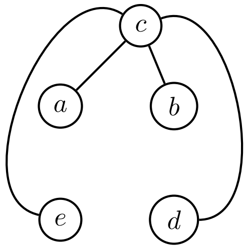
**Example:**

---

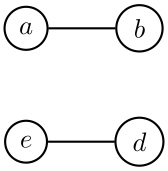
[2] Note that it's important that the odd-degree vertices are even, because in this case a perfect matching always exists.



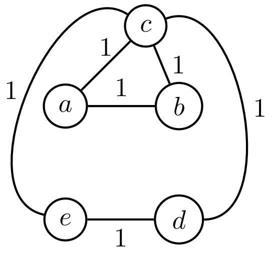
We can build the following MST $T^*$:



The odd-degree vertices are $a, b, e, d$. Then, a possible min-weight perfect matching $M^*$ is the following:



Finally, the graph $G^* = (V, E^*) \cup M^*$ becomes the following:



An Eulerian cycle computed on this grapg can be: $c, d, e, c, b, a, c$. We can find shortcuts as we did for *APPROX_METRIC_TSP* and the final Tour $H$ becomes: $c, d, e, b, a, c$

### 16.1.1 Analysis

1. $w(H) \leq w(T^*) + w(M^*)$ due to triangle inequality
2. $w(T^*) \leq w(H^*)$

We want to prove that $w(H) \leq \frac{3}{2}w(H^*)$

It can be proved that $w(M^*) \leq \frac{1}{2}w(H^*)$ [3]. Then, putting all together:

$$w(H) \leq w(H^*) + \frac{w(H^*)}{2} = \frac{3}{2}w(H^*)$$

---

[3] see lecture notes

# Chapter 17

# Lec 17 - Set Cover

## 17.1 Set cover

The set-covering problem is an optimization problem that models many problems that require resources to be allocated. An instance $I = (X, F)$ of the set-covering problem consists of a finite set $X$ and a family $F$ of subsets of $X$, such that every element of $X$ belongs to at least one subset in $F$ (i.e. $F$ covers $X$):

$$X = \bigcup_{S \in F} S$$

- $X$ set of objects/elements usually called *universe.*
- $F \subseteq \{S : S \subseteq X\}$

The problem is to find a **minimum-size** subset $F' \subseteq F$ whose members cover all of $X$.

**Example:**

- $X = \{1, 2, 3, 4, 5\}$
- $F = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$
- $F' = \{\{1, 2, 3\}, \{4, 5\}\}$

The set-covering problem abstracts many commonly arising combinatorial problems. As a simple example, suppose that $X$ represents a set of skills that are needed to solve a problem and that we have a given set of people available to work on the problem. We wish to form a committee, containing as few people as possible, such that for every requisite skill in $X$, at least one member of the committee has that skill. In the decision version of the set-covering problem, we ask whether a covering exists with size at most $k$, where $k$ is an additional parameter specified in the problem instance.

Set cover in its decision version is **NP-Hard**:

**Proof:** Reduction from Vertex cover. Given an instance of Vertex cover Problem $< G = (V, E), k >$, we create an instance of Set cover problem $< (X, F), k >$ where

- $X = E$.
- $F = \{S_1, S_2, ..., S_n\}$ one $\forall v \in V$ from 1 to $n$.
- $S_i = \{e = (u, v)$ such that $u = i$ or $v = i\}$

Basically, there are $|V| = n$ subsets $S_i$, and each subset is the set of edges incident to vertex $i$.

Claim: Finding a set cover of size $k \iff$ finding a vertex cover of size $k$.

- $\Rightarrow$ Suppose $\{S_1, S_2, ..., S_k\}$ is a set cover for $X$. Then, every edge in $E$ must be incident to at least one vertex $u1, ..., u_k$. Therefore, it forms a vertex cover of size $k$ in $G$.
- $\Leftarrow$ Suppose $u_1, ..., u_k$ is a vertex cover in $G$. Then, $S_i$ covers all the edges incident to vertex $u_i$. Therefore, $\{S_1, ..., S_k\}$ is a set cover of size $k$ for $X$.

## 17.2 Approximation algorithm: greedy approach

The greedy method works by picking, at each stage, the set $S$ that covers the greatest number of remaining elements that are uncovered:

- Choose the subset that contains the largest number of uncovered elements.
- Remove from $X$ those elements.
- Repeat.

**Algorithm 23** APPROX_SET_COVER

```
procedure APPROX_SET_COVER($X, F$)
    $U = X$
    $F' = \emptyset$
    while $U \neq \emptyset$ do
        let $S \in F = max_{S' \in F}\{|S' \cap U|\}$
        $U = U \setminus S$
        $F = F \setminus \{S\}$
        $F' = F' \cup \{S\}$
    return $F'$
```

**Correctness:** At every iteration $|U|$ decreases by at least one

**Complexity:** Since the number of iterations of the loop is bounded from above by $min\{|X|, |F|\}$, and we can implement the loop body to run in time $O(|X|\,|F|)$, a simple implementation runs in time $O(|X|\,|F|\,min\{|X|, |F|\})$. It can be at most cubic in the input size (with the right data structure can be implemented in $O(|X| + |F|)$, i.e. in linear time).

## 17.3 Analysis

Let $|F^*|$ be the size of the optimal solution. We'll show that:

$$\frac{|F'|}{|F^*|} \leq (log_2 n) + 1$$

where $n = |X|$.

**Property:** if $(X, F)$ admits a cover with $|F| \leq k$, then $\forall X' \subseteq X$ $(X', F)$ admits a cover with $|F| \leq k$.

**Idea:** Try to bound the number of iterations such that the set of remaining elements gets empty.

- $U_0 = X$.
- $U_i$ = residual universe at the end of the i-th iteration.
- $|F^*| = k$ unknown.

**Lemma:** After the first $k$ iterations the residual universe is at least halved, that is $|U_k| \leq \frac{n}{2}$.

**Proof:** $U_k \subseteq X \Rightarrow U_k$ admits a cover size $\leq k$, all in $F$, i.e. not selected by the algorithm (by the property above). Let $T_1, T_2, ..., T_k \in F$ be those sets, where $\bigcup T_i$ covers $U_k$.

We apply the **pigeonhole** principle: given a set of elements where there is an order relation, there is always at least one element whose value is greater than the mean value[1]:

$$\exists \bar{T} : |U_k \cap \bar{T}| \geq \frac{|U_k|}{k}$$

We'll now show that in the first $k$ iterations, for each iteration $\frac{|U_k|}{k}$ elements get covered:

$\forall 1 \leq i \leq k$, let $S_i \in F$ the selected subset by the algorithm. This subset has the following property:

$$|S_i \cap U_i| \geq |T_j \cap U_i| \forall 1 \leq j \leq k$$

This is true because at each interaction I select the set with largest cardinality. This property is valid also for $\bar{T}$. Then:

$$|S_i \cap U_i| \geq |\bar{T} \cap U_i| \geq |\bar{T} \cap U_k| \geq \frac{|U_k|}{k}$$

Thus, after the first $k$ iterations the algorithm covered at least $\frac{|U_k|}{k} k = |U_k|$ elements. Since $U_k$ is the set of elements not selected by the algorithm after $k$ iterations, it follows that:

$$\begin{aligned} |U_k| &\leq n - |U_k| \\ |U_k| &\leq \frac{n}{2} \end{aligned} \tag{17.1}$$

We can use the lemma to prove the approximation factor of the algorithm. Since it is a greedy algorithm, we can see it as a recursive algorithm that chooses a subset and then iterates recursively on the residual universe:

- After the first $k$ iterations, the residual universe is at least halved:

$$|U_k| \leq \frac{n}{2}$$

- After $k \cdot i$ iterations:

$$|U_{k \cdot i}| \leq \frac{n}{2^i}$$

Then, the number of necessary iterations is $(log_2 n)k + 1$ (the $+1$ is used to cover any last element that remains to be covered).

$\Rightarrow |F'| \leq (log_2 n)|F^*| + 1$ (because at every iteration $|F'|$ is increased by one).

---

[1] Given a shelter for pigeons with $n$ nests, if there are $n+1$ pigeons it means that at least one nest is occupied by two pigeons.

# Chapter 18

# Lec 18 - Randomized Algorithms

## 18.1 Randomized algorithms

Randomized algorithms are algorithms that may do **random choices**. Surprisingly, adding random choices into algorithms can make them simpler and faster.

**Example: Randomized quicksort**

In the worst case[1], quicksort has a complexity of $O(n^2) \rightarrow T_{qs} = O(n^2)$.

Randomized quicksort (RQS) chooses the *pivot* at random. The expected complexity of RQS is:

$$E[T_{RQS}] = O(nlog\, n)$$

**Example 2: Verify polynomial identities** Check whether two polynomials are equivalent:

$$\begin{aligned} &H(x) = (x+1)(x-2)(x+3)(x-4)(x+5)(x-6) \\ &G(x) = x^6 - 7x^3 + 25 \\ &H(x) \overset{?}{\equiv} G(x) \end{aligned} \tag{18.1}$$

The obvious algorithm is to transform $H(x)$ in canonical form:

$$\sum_{i=0}^{d=6} c_i x^i$$

where $c_i$ is the $i$-th coefficient and $d$ is the maximum degree of $H(x)$. Then, we can verify whether all the coefficients $c_i$ of all monomial are equal.

The **complexity** of this simple algorithm is $O(d^2)$.

**A faster algorithm:**

1. choose a random integer $r$
2. compute $H(r)$
3. compute $G(r)$

---

[1] when the *pivot* is the first elements and the elements are already sorted

4. if $H(r) = G(r)$ then return *YES*, else return *NO*

Note that this is a linear algorithm $O(d)$, but does it work?

Let's consider the following example:

1. $r = 2$
2. $H(2) = 0$
3. $G(2) = 33$
4. $G \neq H$

In this case the algorithm always outputs the correct answer, but what if $H(r) = G(r)$ ?

Let's consider the following two polynomials:

$$x^2 + 7x + 1 \overset{?}{\equiv} (x+2)^2$$

1. $r = 1$
2. $1 + 7 + 1 = 3^2 = 9$

The algorithm return *YES* even if the two polynomial are **not** equivalent.

The algorithm fails only if $r$ is a root (solution) of the polynomial $F(x) = G(x) - H(x) = 0$. Basically, if the algorithm outputs *NO* is always correct, but may fail when it outputs *YES*.

Therefore, the algorithm fails only if it is *unlucky* in choosing the value of $r$. But how likely is to choose such a value of $r$?

If $r \in \{1, 2, ..., 100d\}$ where $d$ is the max degree in $F(x)$, then the probability that the algorithm fails is the followinng:

$$P_r(\text{algorithm fails}) \leq \frac{d}{100d} = \frac{1}{100} = 1\%$$

It's unlikely that the algorithm fails, but still not satisfactory. We can reduce the probability of error by running the algorithm multiple times:

1. run the algorithm 10 times.
2. if it returns *YES* is **all** the runs then return *YES*, else return *NO*.

Now the probability of error is much lower:

$$P_r(\text{algorithm fails}) \leq \left(\frac{1}{100}\right)^{10} = 10^{-20} < 2^{-64}$$

It's easier that your computer returns a wrong answer because it gets hit by a cosmic radiation that makes some bits flip.

# Chapter 19

# Lec 19-20 - Minimum cut

## 19.1 Classification of randomized algorithms

1. Randomized algorithms that never fail. They are called *LAS VEGAS* algorithms (e.g. randomized quicksort).

   $$\forall i \in I,\ A_R(i) = s \text{ s.t. } (i,s) \in \Pi$$

   where $\Pi \subseteq I \times S$ is a computational problem.

   Randomness comes into play in the analysis of the complexity. $\forall n$, $T(n)$[1] is a **random variable** of which we usually study its expectation $E[T(n)]$.

   If $P(T(n) > c\dot{f}(n)) \leq \frac{1}{n^k}$ for some constants $c$ and $k$, then we say that $T(n) = O(f(n))$ **with high probability**.

2. Randomized algorithms that may fail. They are called *MONTE CARLO* algorithms (e.g. verifying polynomial identity).

   $$\forall i \in I,\ \text{it's possible that } A_R(i) = s \text{ s.t. } (i,s) \notin \Pi$$

   We study the probability $P((i,s) \notin \Pi)$ as a function of the input size $|i| = n$.

   For decision problems, these algorithms can be divided in two classes:

   - **One-sided:** They may fail only on one answer.
   - **Two-sided:** They may fail in both answer.

## 19.2 Terminology

**Definition:** Given a computational problem $\Pi \subseteq I \times S$, an algorithm $A_\Pi$ has complexity $T(n) = O(f(n))$ **with high probability** (w.h.p.) if $\exists$ constants $c, d > 0$ such that $\forall i \in I$ of size $n$ the following probability holds:

$$P(A_\Pi(i) \text{ terminates in } > c \cdot f(n) \text{ steps}) \leq \frac{1}{n^d}$$

Then the probability of having a complexity of $O(f(n))$ is $> 1 - \frac{1}{n^d}$, that is, tends to 1 as $n$ tends to $\infty$.

---

[1] $T(n)$ is called complexity function

**Definition:** Given a computational problem $\Pi \subseteq I \times S$, an algorithm $A_\Pi$ is correct w.h.p. if $\exists$ constant $d > 0$ such that $\forall i \in I,\ |i| = n$:

$$P((i, A_\Pi(i)) \notin \Pi) \leq \frac{1}{n^d}$$

**Markov's lemma:** Let $T$ be a non-negative, bounded ($\exists b \in \mathbb{N}$ s.t. $P(T > b) = 0$), integer random variable. Then, $\forall t$ such that $0 \leq t \leq b$:

$$t \cdot P(T \geq t) \leq E[T] \leq t + (b - t)P(T \geq t)$$

**Application of Markov's lemma:** Given a *LAS VEGAS* algorithm $A_\Pi$, assume that:

1. $T_{A_\Pi}(n) = O(f(n))$ with high probability, in particular, $P(T_{A_\Pi}(n) > c \cdot f(n)) \leq \frac{1}{n^d}$.
2. $A_\Pi$ has a worst-case deterministic complexity $O(n^a),\ a \leq d\ \forall n$

We can use Markov's lemma to prove the following property:

$$E[T_{A_\Pi}(n)] = O(f(n))$$

**Proof:** Just apply Markov's lemma:

$$E[T_{A_\Pi}(n)] \leq c \cdot f(n) + (n^a - c \cdot f(n)) \cdot \frac{1}{n^d} \leq c \cdot f(n) + \frac{n^a}{n^d} \leq c \cdot f(n) + 1$$

where:

- $c \cdot f(n)$ is $t$
- $(n^a - c \cdot f(n))$ is $b - t$
- $\frac{1}{n^d}$ is $P(T \geq t)$

## 19.3 Karger's algorithm for Minimum cut

The minimum cut problem is finding a cut of minimum size, that is, the minimum number of edges whose removal disconnects the graph.

Karger's algorithm actually solves a more general problem: minimum cut on **multigraphs** (i.e. multiple edges between two vertices are allowed).

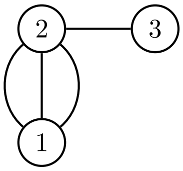
**Definition:** A **multiset** is a collection of objects with repetitions allowed.

$$S = \{\{\text{objects}\}\}$$

where $\forall$ object $o \in S$, $m(o) \in \mathbb{N} \setminus \{0\}$ is called multiplicity, that is, how many copies of $o$ are in $S$.

**Definition:** We denote a multigraph as follows: $G = (V, E)$ where $V \subseteq \mathbb{N}$ and $E$ is a multiset of elements $(u, v)$ $u \neq v$.

**Example:**



A simple graph is also a multigraph.

**Definition:**Given a connected multigraph $G = (V, E)$. A cut $C \subseteq E$ is a multiset of edges such that $G' = (V, E \setminus C)$ is not connected.

**Karger's algorithm (simplified version):**

1. Choose an edge at random;
2. *Contract* the two vertices of that edge;
3. Repeat until only two vertices remain;
4. Return the edges between them;

**Definition:** Given a multigraph $G = (V, E)$ and an edge $e = (u, v)$, the **contraction** of $G$ with respect to $e$ is: $G_{/e} = (V', E')$ where:

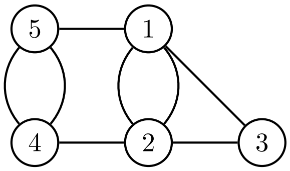
- $V' = V \setminus \{u, v\} \cup \{z_{u,v}\}$ where $z_{u,v} \notin V$.
- $E' = E \setminus \{\{(x, y) : (x = u) \text{ or } (x = v)\}\} \cup \{\{(z_{u,v}, y) : (u, y) \in E \text{ or } (v, y) \in E,\ y \neq u \text{ and } y \neq v\}\}$

It follows that:

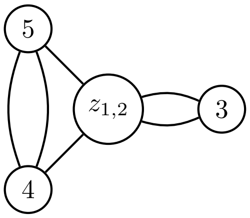
- $|V'| = |V| - 1$
- $|E'| = |E| - m(e) \leq |E| - 1$

Basically, it makes the two vertices to collapse in just one vertex connected with all the previous adjacent vertices. If as a result there are several edges between some pairs of (newly formed) vertices, retain them all.

**Example:**



The contraction of $G$ with respect to the edge $(1, 2)$ is the following:



Actually, the probability that at the first run Karger's algorithm returns a minimum cut is **not** high. The idea is to repeat the procedure $k$ times to reduce the probability of error. The value of $k$ will be determined by the analysis of the algorithm.

**Algorithm 24** Karger's algorithm

```
procedure KARGER($G, k$)
    $min = \infty$
    for $i = 1$ to $k$ do
        $t$ = FULL_CONTRACTION(G)
        if $t < min$ then
            $min = t$
    return $min$
```

**Algorithm 25** FULL_CONTRACTION

```
procedure FULL_CONTRACTION($G = (V, E)$)
    for $i = 1$ to $n - 2$ do
        $e$ = random($E$)
        $G' = (V', E') = G_{/e}$    //Contraction
        $V = V'$
        $E = E'$
    return $|E|$
```

## 19.4 Analysis of Karger's algorithm

We'll show for which value of $k$ the algorithm returns a minimum cut with high probability.

**Property:** $\forall$ cut $C'$ in $G_{/e}$ $\exists$ a cut $C$ in $G$ of the same cardinality.

This implies that $|$minimum cut in $G_{/e}| \geq |$minimum cut in $G|$.

**Proof:**Given a cut $C'$ in $G_{/e}$, the idea is to determine the corresponding cut $C$ in $G$ by substituting each edge $(z_{u,v}, y)$ in $C'$ with $(u, y)$ or $(v, y)$. Then, it remains to show that $C$ is a cut in $G$.

Let $C'$ be a cut in $G_{/e} = (V', E')$. Then, $C'$ separates $V'$ in 2 connected components. Let $V_1 \subset V'$ the connected component containing $z_{u,v}$, and let $x \notin V_1$. Then, in $G_{/e}$ every path from $z_{u,v}$ to $x$ must use an edge in $C'$.

Now we'll show that $C$ in $G$ disconnects $u$ and $v$ from $x$. Assume by contradiction that $C$ is **not** a cut in $G$. This implies that it exists a path between $u$ and $x$ after the removal of $C$ from $E$. Then, the path between $z_{u,v}$ and $x$ *survives* the removal of $C'$ in $G_{/e}$. This implies that $C'$ is not a cut in $G_{/e}$ → contradiction!

Note that the cuts that disappear in the contracted graph are the ones hit by the random choice of the edge $e$, all the others are preserved. This means that the only time the algorithm fails is when an edge belonging to a minimum cut in $G$ is hit by the random choice. Basically, by proving the property above we have shown that, if the algorithm never hits an edge belonging to a minimum cut, then it returns a correct solution (because the minimum cut is preserved in $G_{/e}$).

Therefore, we want the probability of **not** hitting edges of the minimum cut to be sufficiently high.

### 19.4.1 Conditional probability recall

**Definition:** The events $E_1, E_2$ are **independent** if:

$$P(E_1 \cap E_2) = P(E_1) \cdot P(E_2)$$

**Definition:** Given $P(E_1) > 0$, then:

$$P(E_2|E_1) = \frac{P(E_1 \cap E_2)}{P(E_1)}$$

extension to $k$ events:

$$P(E_1 \cap E_2 \cap E_3 \cap ... \cap E_k) = P(E_1)P(E_2|E_1)P(E_3|E_1 \cap E_2)...P(E_k|E_1 \cap ... \cap E_{k-1})$$

### 19.4.2 Analysis of *FULL_CONTRACTION*

**Intuition:** Since |min. cut| is a small portion of $|E|$, it is *unlikely* to hit an edge of a minimum cut. Let's calculate what is this probability.

**Property:** Let $G = (V, E),\ |V| = n$. If $G$ has a minimum cut of size $t$, then $|E| \geq t \cdot \frac{n}{2}$.

**Proof:** $d(v) \geq t \ \forall v \in V$. This is because If we take a node and remove all the edges incident to that node, that's a cut.

$$\sum_{v \in V} d(v) = 2m \geq t \cdot n$$

$$m = |E| \geq \frac{t \cdot n}{2}$$

Let $t = |\text{minimum cut}|$. Given the event $E_i$ = in the i-th contruction i did **not** hit an edge of the minimum cut, it follows that:

$$P(\overline{E}_1) = \frac{t}{|E|} \leq \frac{t}{t \cdot \frac{n}{2}} = \frac{2}{n}$$

$$P(\overline{E}_1) \leq \frac{2}{n}$$

$$P(E_1) \geq 1 - \frac{2}{n}$$

Since after every contruction $|V'| = |V| - 1$, the following holds:

$$P(E_2|E_1) \geq 1 - \frac{t}{t \cdot \frac{(n-1)}{2}} = 1 - \frac{2}{n-1}$$

$$...$$

$$P(E_i|E_1 \cap ... \cap E_{i-1}) \geq 1 - \frac{t}{\frac{t(n-i+1)}{2}} = 1 - \frac{2}{n-i+1}$$

Therefore, the probability that *FULL_CONTRUCTION* succeeds becomes:

$$P(\textit{FULL\_CONTRUCTION succeeds}) \geq P(\bigcap_{i=1}^{n-2} E_i) = \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1}$$

$$\geq \frac{\cancel{n-2}}{n} \cdot \frac{\cancel{n-3}}{n-1} \cdot \frac{\cancel{n-4}}{\cancel{n-2}} \cdots \frac{\cancel{3}}{\cancel{5}} \cdot \frac{2}{\cancel{4}} \cdot \frac{1}{\cancel{3}} = \frac{2}{n(n-1)}$$

Basically, the probability that *FULL_CONTRUCTION* does not hit an edge of a minimum cut is at least $\frac{2}{n^2}$. Not so good as a bound. Our algorithm may err in declaring the cut it outputs to be a min-cut. Karger's amplifies this probability by repeating *FULL_CONTRUCTION* multiple times:

$$P(\textit{the k runs of FULL\_CONTRUCTION do not return the minimum cut}) \leq \left(1 - \frac{2}{n^2}\right)^k$$

We want to find a value for $k$ such that $\left(1-\frac{2}{n^2}\right)^k \leq \frac{1}{n^d}$. In order to do this, we'll use the following two rules:

- $(1+\frac{x}{y})^y \leq e^x \;\; y \geq 1, y \geq x$
- $e^{-ln\, n} = \frac{1}{n}$

By choosing $k = d\frac{n^2}{2} ln\, n$, it follows that:

$$= \left(\left(1-\frac{2}{n^2}\right)^{n^2}\right)^{\frac{d}{2} ln\, n}$$

$$\leq (e^{-2})^{\frac{d}{2} ln\, n} = e^{-ln\, n^d} = \frac{1}{n^d}$$

Then, by choosing that value for $k$ the Karger's algorithm succeeds with high probability:

$$P(Karger\text{'s } succeeds) \geq 1 - \frac{1}{n^d}$$

### 19.4.3 Complexity

*FULL_CONTRUCTION* can be implemented in $O(n^2)$. Then, the complexity of Karger's algorithm is $O(n^4 log\, n)$. It is not very fast, but it can be improved to obtain a complexity of $O(n^2 log^3\, n)$.

The fastest algorithm for this problem has a complexity of $O(mlog\, n)$.

# Chapter 20

# Lec 21 - Chernoff bounds

## 20.1 Chernoff bounds

Chernoff bounds are tools from modern probability theory that can be used for the analysis of randomized algorithms. They're a more powerful version of the Markov's lemma.

In many cases, the study of $P(T(n) > c \cdot f(n))$ can be rephrased as the study of the distribution of some sum of **indicator** random variables (0-1 variables). In general:

$$X = \sum_{i=1}^{n} X_i \quad X_i \text{ indicator random variable}$$

The variable $X$ counts the number of successes obtained by the $n$ indicator variables (e.g. counting the number of times we get *head* by tossing a coin 10 times). We'll usually have that $X_i$'s are **independent**.

Given $P(X_i = 1) = p_i$, the expected value (mean value) of $X$ is given by:

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} p_i = \mu$$

We want to analyze the probability that $X$ deviates from its expected value. Let's try to use Markov's lemma:

$$P(X > (1+d)\mu) \leq \frac{E[X]}{(1+d)\mu} = \frac{\mu}{(1+d)\mu} = \frac{1}{1+d}$$

If $d = 1$, the probability that $X$ deviates from $\mu$ is $\leq \frac{1}{2}$. This bound is not so informative!

### 20.1.1 A more powerful tool

**Chernoff bound:** Let $X_1, X_2, ..., X_n$ be independent indicator random variables where $E[X_i] = p_i$, $0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then, $\forall\, d > 0$:

$$P(X > (1+d)\mu) < \left(\frac{e^d}{(1+d)^{(1+d)}}\right)^{\mu}$$

**Example: tossing a coin**

$n$ coin flips → $X_1, X_2, ..., X_n$.

- Let $X_i = 1$ get *heads* and $X_i = 0$ get *tails*.
- Since we assume the coin is fair, $P(X_i = 1) = \frac{1}{2}\ \forall\, i$.
- $X = \sum_{i=1}^{n} X_i$ counts the number of *heads* obtained by flipping $n$ times the coin.
- $E[X] = \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}$.

Question: What's the probability of getting more than $\frac{3}{4}n$ *heads*? Let's apply both Markov's lemma and Chernoff bound:

1. Markov's lemma:
$$P(X > (1+\frac{1}{2})\mu) \leq \frac{1}{1+\frac{1}{2}} = \frac{2}{3}$$
2. Chernoff bound:
$$P(X > (1+\frac{1}{2})\mu) < \left(\frac{e^{\frac{1}{2}}}{(\frac{3}{2})^{\frac{3}{2}}}\right)^{\frac{n}{2}} < (0.95)^n$$

The Chernoff bound tells us that the probability goes to 0 exponentially fast as $n$ increases.

### 20.1.2 Variants of Chernoff bound:

- $P(X < (1-d)\mu) < e^{-\mu\frac{d^2}{2}} \quad 0 < d \leq 1$
- $P(X > (1+d)\mu) < e^{-\mu\frac{d^2}{2}} \quad 0 < d \leq 2e-1$

# Chapter 21

# Lec 22 - Analysis of Randomized Quicksort

## 21.1 Randomized Quicksort

**Algorithm 26** Randomized Quicksort

```
procedure RandQuicksort($S$)
    if $|S| \le 1$ then return $S$
    $p$ = random($S$)    // pick a pivot element uniformly at random from $S$
    $S_1 = \{x \in S$ s.t. $x < p\}$
    $S_2 = \{x \in S$ s.t. $x > p\}$
    $z_1 = RandQuicksort(S_1)$
    $z_2 = RandQuicksort(S_2)$
    return $z_1, z_2$
```

It is a *LAS VEGAS* algorithm (we choose the pivot at random).

Suppose that $p$ is always the median of $S$, then:

$$T_{RQS}(n) = \begin{cases} 2T_{RQS}(\frac{n}{2}) + O(n) & n > 1 \\ 0 & n \le 1 \end{cases}$$

where $n = |S|$. Then $T_{RQS}(n) = O(nlog\, n)$.

However, $p$ is the median with probability $\frac{1}{n}$, very low. Actually, there exists a deterministic linear algorithm to compute the median. This implies that this version of deterministic Quicksort has complexity $O(nlog\, n)$. The problem is that the hidden constant is very high, thus it is inefficient in practice.

Fortunately, we don't really need that the random choice hits always the median. For example, let's see what happens if the pivot is always chosen between the $(\frac{n}{4}+1)$-th order statistics and the $(\frac{3}{4}n)$-th order statistics. By looking at the recursion tree, if such a pivot is always chosen, all the root-leaf paths are no longer than $log_{\frac{4}{3}} n$. This is because $|S|$ after $i$ *successes* is $\le (\frac{3}{4})^i n$. This implies that $T_{RQS} = O(nlog\, n)$. It is not necessary that $S_1$ and $S_2$ are perfectly balanced.

If we prove that the depth of the recursion tree is $O(log\, n)$ w.h.p. then $T_{RQS} = O(nlog\, n)$ w.h.p.

## 21.2 Analysis

Let's call the event $E =$ *lucky choice of the pivot*, that is, pivot chosen between the $(\frac{n}{4}+1)$-th order statistics and the $(\frac{3}{4}n)$-th order statistics. Then:

$$P(E) = \left(\frac{\frac{3}{4} - (\frac{n}{4}+1)+1}{n}\right) = \frac{1}{2}$$

Fix **one** root-leaf path $P_1$:

**Lemma:** $P(|P_1| > a \cdot log_{\frac{4}{3}} n) < \frac{1}{n^3}$.

If this lemma is true, it follows that:

1. Given the event $E_i =$ *the path* $p_i$ *has length* $> a \cdot log_{\frac{4}{3}} n$:

$$P(\exists \textit{ a path with length } > a \cdot log_{\frac{4}{3}} n) = P\left(\bigcup_{i=1}^{n} E_i\right) \leq \sum_{i=1}^{n} P(E_i) < \frac{1}{n^2}$$

2. Therefore, the probability that all the root-leaf paths have length $\leq a \cdot log_{\frac{4}{3}} n$ is the following:

$$P(\textit{all the root-leaf paths have length} \leq a \cdot log_{\frac{4}{3}} n) \geq 1 - P(\exists \textit{ a path with length } > a \cdot log_{\frac{4}{3}} n) \geq 1 - \frac{1}{n^2}$$

This would imply that $T_{RQS}(n) = O(nlog\, n)$ w.h.p.

It remains to prove the Lemma above.

**Proof:** Given a path $\Pi$, let $l = a \cdot log_{\frac{4}{3}} n$. We want to study the event $E =$ *in the first* $l$ *nodes of* $\Pi$ *there have been* $< log_{\frac{4}{3}} n$ *lucky choices.*

- $X_i \quad 1 \leq i \leq l$
- $X_i = 1$ if at the $i$-th node of $\Pi$ there is a *lucky choice* of the pivot.
- $P(X_i = 1) = \frac{1}{2} \quad \forall i.$
- $X_i$ are independent.

We want the probability $P(\sum_{i=1}^{l} X_i) < log_{\frac{4}{3}} n$ to be bound.

Given $X = \sum_{i=1}^{l} X_i$, its expected value is defined as follows:

$$\mu = E[X] = E\left[\sum_{i=1}^{l} X_i\right] = \sum_{i=1}^{l} E[X_i] = \sum_{i=1}^{l} \frac{1}{2} = \frac{a}{2} log_{\frac{4}{3}} n$$

Now, let's apply the following Chernoff bound:

$$P(X < (1-d)\mu) < e^{-\mu\frac{d^2}{2}} \quad 0 < d \leq 1$$

We want $(1-d)\mu = log_{\frac{4}{3}} n$:

$$(1-d)\frac{a}{2} log_{\frac{4}{3}} n = log_{\frac{4}{3}} n$$

One possible solution is:

- $a = 8$
- $d = \frac{3}{4}$

$$\begin{aligned}
P(X < log_{\frac{4}{3}} n) &< e^{-\frac{8}{4} log_{\frac{4}{3}} n \cdot \frac{9}{16}} \\
&= e^{-log_{\frac{4}{3}} n \cdot \frac{9}{8}} \\
&< e^{-log_{\frac{4}{3}} n} \\
&= e^{-\frac{ln\, n}{ln\, \frac{4}{3}}} \\
&= \left(e^{-ln\, n}\right)^{\frac{1}{ln\, \frac{4}{3}}} \\
&= \left(\frac{1}{n}\right)^{1/ln\, \frac{4}{3}} \\
&< \frac{1}{n^3}
\end{aligned}$$