Design and Analysis of Algorithms

José Dorronsoro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contents

Before We Start

On reading and studying these notes:

From Brad DeLong's, UC Berkeley, A note on reading big, difficult books:

- It is certainly true that there are many who can parrot verbal formulas yet lack knowledge of facts, terms, and concepts.
- It is certainly true that there are many who have knowledge of facts, terms, and concepts and yet lack deep understanding.
- But **I am not aware of anyone who has deep understanding of a discipline and yet lacks knowledge of facts, terms, and concepts.**
- And **those who know the facts, terms, and concepts cold are the absolute best at parroting verbal formulas.**

1 Elementary Graph Algorithms

1.1 Basic Concepts on Graphs

Definitions

- Graph: Pair $G = (V, E)$ of a set V of vertices (nodes) and a set E of edges (u, v) with $u, v \in V$
- Edges imply direction: in (u, v) we go from u to v
- In general, graphs are **directed**
- **Undirected** graphs: $(u, v) \in E$ iff $(v, u) \in E$
- **Unweighted** graphs: we only consider edge structure
- **Weighted** graphs: edges (u, v) have weights w_{uv}
- **Multigraphs**: there might several edges between two vertices and also between a vertex and itself

Storing an Unweighted Graph

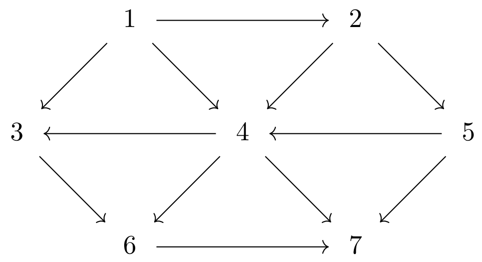
- **Adjacency matrix:** Assume $V = \{1, \ldots, N\}$. Then if $(i, j) \in E$, $m_{ij} = 1$; else, $m_{ij} = 0$
  - Not for multigraphs
  - By convention $m_{ii} = 1$ (although sometimes we may consider $m_{ii} = 0$)
  - Cost: $\Theta(|V|^2) = \Theta(N^2)$
- **Adjacency list:** We can consider a pointer table $T[\ ]$ where $T[i]$ points to a linked list
  - If $(i, j) \in E$, then j is in one of nodes pointed by $T[i]$
  - Cost: $\Theta(|V|) + \Theta(|E|)$
  - No problem for multigraphs

- In all cases the cost is $O(|V|^2)$ since for standard graphs

$$|E| \leq |V|(|V|-1) = O(|V|^2)$$

An Example

- A directed graph:



The Adjacency Matrix

- The first rows of the adjacency matrix are

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ & & & \dots & & & \end{pmatrix}$$

The Adjacency List

- Partial adjacency list: we use a lexicographic order

$$\begin{array}{ccccccc} 1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 \\ 2 & \rightarrow & 4 & \rightarrow & 5 & & \\ 3 & \rightarrow & 6 & & & & \\ 4 & \rightarrow & 3 & \rightarrow & 6 & \rightarrow & 7 \\ & & & \dots & & & \end{array}$$

The Size of a Graph

- While $|V|$ and $|E|$ are in general independent, we may expect $|V| = O(|E|)$ for interesting graphs
  - $|E|$ will usually be G's size
- G is **dense** if $|E| = \Theta(|V|^2)$
- G is **sparse** if $|E| \ll |V|^2$
- If G s dense, the adjacency matrix storage is more efficient; if G is sparse, adjacency lists are better
- We will usually work with adjacency lists, using adjacency matrices for special algorithms

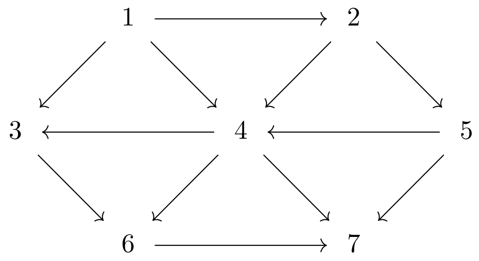
1.2 Minimum Distances on Graphs

Minimum Distance Problems

- **Path** from u to v: a subset $\pi = \{u = u_0, \ldots, u_K = v\}$ with $(u_i, u_{i+1}) \in E$
- **Length** of π: $|\pi| = K = \#$(number of) edges
- First problem: given u, find a **shortest path** (i.e., a path with the smallest number of edges) π from u to any other v
- First question: how to obtain such paths?
- First idea: get a tree like "descending representation" T of G starting from u and avoiding lower duplicate vertices

Minimum Distance Example

- Think of each vertex as a ball and of edges as equal lenght strings, and make G "hang" from u discarding "'repeated" edges
- On the previous graph,

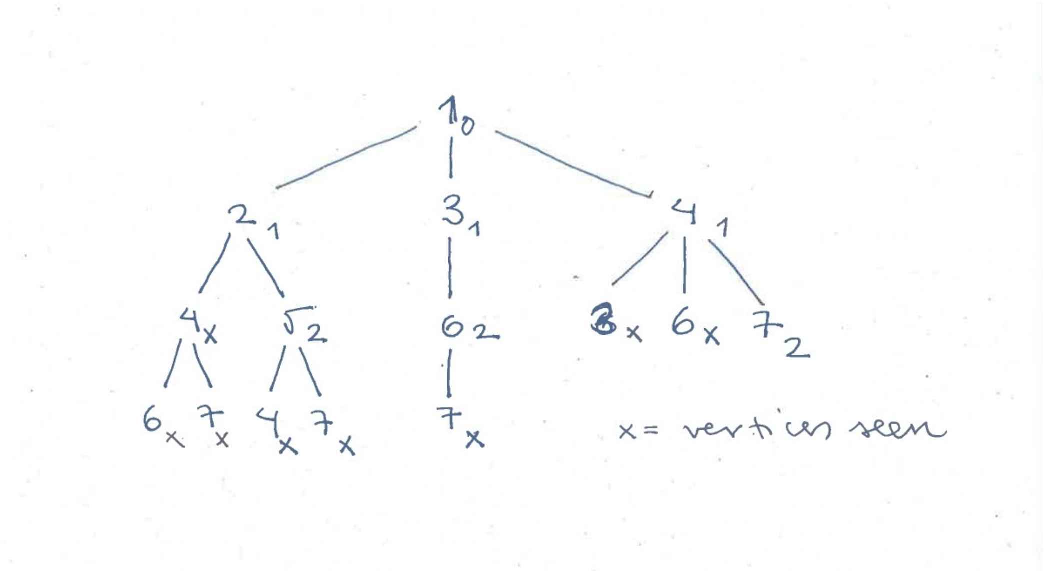


Breadth First Traversal

- We find the minimum distances by breadth first traversal (BFS) on this hanging representations

Some Observations on Minimum Distance Problems

- If $d[v]$ is the depth of v in T, it is reasonable to expect $d[v]$ to be the minimum distance from u to v
  - But we have to prove it
- If $p[v]$ is the father of v in T, we can obtain a minimum length path from u to v with edges $(w = p[v], v), (p[w], w), \ldots$, and so on
- Notice that we have found the minimum distances from u to **all** $v \in V$
  - They are unique, but the minimum paths are not
- Q: how can we derive an algorithm for this?
- We can use a standard FIFO queue Q to process the different vertices and the tables $p[v]$ and $d[v]$



- In fact, this fits in the general framework of **Breadth First Search**

First Algorithm for Minimum Distances

- We need tables $p[v]$ for the vertex "previous" to v, $d[v]$ for the minimum distance from u to v and $v[v]$ to mark v as seen
- First, queue–based, pseudocode:

```
def dist_min(u, G):
    s[ ] = F; p[ ] = None; d[ ] = inf
    Q = q()
    d[u] = 0; Q.put(u); s[u] = T
    while not Q.empty():
        v = Q.get()
        for all z adjacent to v:
            if not s[z]:  #first time z is seen
                d[z] = d[v] + c(v,z)
                p[z] = v; s[z] = T
                Q.put(z)
    return d, p
```

Some Observations on `dist_min`

- The table $s[\,]$ is redundant: $s[v] == T$ if and only if $d[v] < \infty$ (exercise: update the psc)
- We can use $p[\,]$ to reconstruct the minimum paths from u to all v (exercise)
- We can use $p[\,]$ to reconstruct the minimum distance table $d[\,]$ (exercise)
  - So $p[\,]$ would be the table to return in, say, a C function
- A vertex enters Q only once $\Rightarrow$ the linked lists are traversed only once $\Rightarrow$ the cost of `distMin` is $O(|E|)$, i.e., linear on G's size

- `dist_min` is a particular instance of the general **Breadth First Search** algorithm

Breadth First Search (BFS) v 1.0

- The pseudocode of the first, queue–based version of BFS is

```
def BFS(u, G):
    s[ ] = F; p[ ] = None
    Q = q()
    doSomething(u)
    s[u] = T; Q.put(u)
    while not Q.empty():
        v = Q.get()
        for all z adjacent to v:
            if not s[z]:
                doSomething(z)
                s[z] = T; p[z] = v
                Q.put(z)
    return p
```

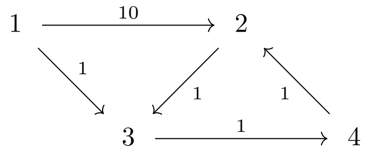
- Since we enter each list only once, if the cost of `doSomething` is $O(1)$, the cost of BFS is $O(|E|)$, i.e., linear,
- If needed, we add a driver to restart `BFS` at unseen nodes

Minimum Distances on Weighted Graphs

- $G = (V, E)$ is a **weighted** graph if there is a function $c : E \to R$
  - We think of $c(i, j)$ as the cost of going from i to j (although sometimes $c(i, j)$ can be negative)
- **Cost** of path π: $c(\pi) = c(\{u_0, \ldots, u_K\}) = \sum_1^K c(u_{j-1}, u_j)$
- Working with adjacency matrices we can store c as $m_{ij} = c_{ij}$ if $(i, j) \in E$ and $m_{ij} = \infty$ if not.
  - Now the convention is $m_{ii} = 0$
- Working with adjacency lists we can store c_{ij} in a second field of the same node of $T[i]$ that stores j

Problems . . .

- Applying our first algorithm to the graph



- Working here with the tree like representation of G is now trickier

Fixing The First Algorithm

	d	p	v	d	p	v	d	p	v
1	0	-	T	0	-	T	0	-	T
2	∞	-	F	10	1	T	10	1	T
3	∞	-	F	1	1	T	1	1	T
4	∞	-	F	∞	-	-	2	3	T
Q	1			$2, 3$			$3, 4$		

- The node 2 gets out of Q too soon $\Rightarrow$ we have to change the ordering in Q
- We need a **priority queue** Q that orders vertices using the current value of $d[v]$
- Now v is seen when it **leaves** Q (and not when it enters Q)
- We also need (again) a table $s[v]$ to check whether v has left Q and, hence, we do not consider it any longer
- This leads to **Dijkstra's** algorithm

Dijkstra's Algorithm

- Dijkstra's pseudocode is:

```
def dijkstra(u, G):
    s[ ] = F; p[ ] = None; d[ ] = inf        # 1
    Q = pq()
    d[u] = 0
    Q.put( (d[u], u) )
    while not Q.empty():                     # 2
        _, v = Q.get()                       # 3
        if not s[v]:
            s[v] = T
            for all z adjacent to v:         # 4
                if d[z] > d[v] + c(v, z):
                    d[z] = d[v] + c(v, z)
                    p[z] = v
                    Q.put( (d[z], z) )       # 5
    return d, p
```

Dijkstra's Algorithm II

- **Example**: First steps of Dijkstra's algorithm on the previous graph

	d	p	v	d	p	v	d	p	v
1	0	-	F	0	-	T	0	-	T
2	∞	-	F	10	1	F	10	1	F
3	∞	-	F	1	1	F	1	1	T
4	∞	-	F	∞	-	-	2	3	F
PQ	1_0			$3_1, 2_{10}$			$4_2, 2_{10}$		

Dijkstra's Cost

- The five commented numbers in the psc determine its cost
- The cost of (1) is clearly $O(N)$
- Using a PQ over a binary heap the cost of `Q.put, Q.get` is $O(\log |Q|)$

  - Q will contain at most all edges, so $|Q| = O(|E|)$
  - Thus, the cost of (3) over all iterations in (2) is $O(|E| \log |E|)$

- We enter (4) **once** per node; thus the total number of joint iterations in (2) and (4) is $|E|$
- Hence, the cost of (5) over all iterations is $O(|E| \log |E|)$
- Since $|E| = O(|V|^2)$, the overall cost is

$$\begin{aligned} O(|V|) + O(|E| \log |E|) &= O(|V|) + O(|E| \log |V|^2) \\ &= O(|V|) + O(|E| \log |V|) \end{aligned}$$

- This will be $O(|E| \log |V|)$ for most graphs, i.e., log linear in a graph's size

Observations on Dijkstra's Algorithm

- We allow that several instances of the same v be in Q
- We can stop the algorithm earlier using a counter of seen vertices (exercise)
  - But have to clear Q, so ...
- Dijkstra **works**: at the end $d[v]$ contains the minimum distances from u to v and we can get the minimum paths using $p[v]$
  - But **this has to be proved**
- Dijkstra is an example of the general **breadth first search** graph algorithm

Breadth First Search (BFS) v 2.0

- The pseudocode for general, PQ based BFS, is

```
def BFS(u, G):
    s[ ] = F; p[ ] = None; Q = pq()
    d[u] = 0; Q.put( (d[u], u) )
    doSomething(u)
    while not Q.empty():
        _, v = Q.get()
        if not s[v]:
            s[v] = T
            for all z adjacent to v:
                doSomething(z)      #perhaps change d[z]
                p[z] = v; Q.put( (d[z], z) )
    return p
```

- If needed, we add a driver to restart `BFS` at unseen nodes
- If the cost of `doSomething` is $O(1)$ and we work with a PQ over min heaps, the cost of BFS is $O(|E| \log |V|)$

1.3 All Pairs Shortest Paths

All Pairs Shortest Paths

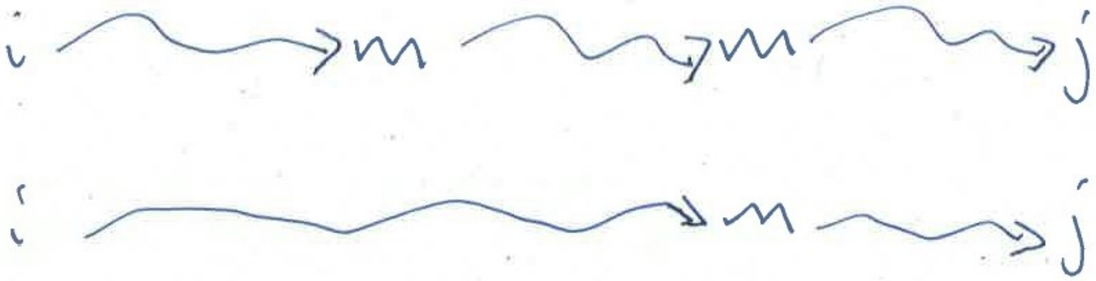
- If (G, c) is a weighted directed graph, we can consider in principle three minimum distance problems:
  - For u, v fixed, find **only** the minimum distance between u and v
  - For u fixed, find the minimum distance between u and **all other** $v \in V$
  - For **all** $u, v \in V$, find the minimum distance between u and v
- While the first problem seems easier, no algorithm for general graphs is better than the best one for the second problem
  - Notice that a minimal path from u to v is also minimal for all vertices in between
- We can solve the third problem iterating an algorithm for the second one over all $u \in V$
- For instance, iterating Dijkstra over all $u \in V$ has a cost $|V| \times O(|E| \log |V|) = O(|V||E| \log |V|)$
  - If G is dense, the cost is then $O(|V|^3 \log |V|)$

Improving on Dijkstra I

- Assume $V = \{1, \ldots, N\}$ and the cost c is nonnegative
- Denote by d_{ij} be the minimum distance between i and j
- We define d_{ij}^k be the minimum distance between i, j but where **the intermediate nodes are taken only from** $\{1, \ldots, k\}$
- It is clear that
$$d_{ij}^0 = c(i, j), \quad d_{ij}^N = d_{ij}$$
- It is clear that **no vertex is repeated on the optimal path** that gives d_{ij}^k

Improving on Dijkstra II

- Obviously, an optimal path between i and j with $\{1, \ldots, k\}$ as intermediate nodes may or may not contain k
- If it doesn't, we have
$$d_{ij}^k = d_{ij}^{k-1}$$
- If it does, we have
$$d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$$
  - A path from i to j **is optimal iff the partial subpaths** between i and k and k and j **are optimal**, i.e.,
$$d_{ij}^k = d_{ik}^k + d_{kj}^k$$
  - But a path **having another** k between i and k or between k and j **cannot be optimal**:



        - We can simply remove the subpath from k to k to get a better path
    - Thus, it is then obvious that
$$d_{ik}^k = d_{ik}^{k-1}, d_{kj}^k = d_{kj}^{k-1}$$

Floyd–Warshall Algorithm

- We can conclude $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$ and $d_{ij} = d_{ij}^N$
- Working with adjacency matrices, this suggest the following (quite bad) pseudocode

```
def FW_0(m_c):
    n = m_c.shape[0]
    d = np.empty( (n, n, n+1) )
    #d[i, j, k]: min dist from i to j with intermediate [1, ..., k]
    d[:, :, 0] = m_c
    for k in range(1, n+1):
        for i in range(n):
            for j in range(n):
                t = d[i, k, k-1] + d[k, j, k-1]
                d[i, j, k] = min(d[i, j, k-1], t)
    return d[:, :, n]
```

- The algorithm is ± obviously correct

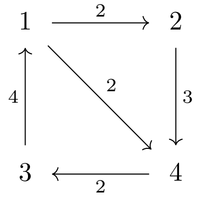
Floyd–Warshall Cost

- The time cost is $O(N^3)$, better than iterated Dijkstra for dense graphs
- The space cost is a first sight also $O(N^3)$ as we use N matrices $N \times N$; but in fact a single matrix D is enough, for
    - We first "retain" d_{ik}, d_{kj}
    - Then for i or $j \neq k$ we set $c = d_{ik} + d_{kj}$, and we can overwrite d_{ij} as $d_{ij} = \min\{d_{ij}, c\}$

- Exercise (easy): rewrite FW taking advantage of this
  - Is it now a good algorithm?
- Exercise (more difficult): how can we recover the optimal paths?
- Observation: FW is our first example of a problem solvable by a **Dynamic Programming (DP)** algorithm
  - An optimization problem with an **optimal substructure** (obvious: any optimization problem has it) that we are able to make **explicit**
  - The explicit substructure formula also shows FW to be correct

Applying Floyd–Warshall

- Example:



- We iteratively compute the intermediate matrices $D^k = (d^l_{ij})$, $k = 0, 1, ldots, N$
- Observe that going from D^{k-1} to D^k **we just copy** $d^k_{ik} = d^{k-1}_{ik}$**,** $d^k_{kj} = d^{k-1}_{kj}$

From D^0 to D^1

- We have

$$\begin{aligned} d^1_{23} &= \min\{d^0_{23}, d^0_{21} + d^0_{13}\} = \min\{\infty, \infty + \ldots\} = \infty \\ d^1_{24} &= \min\{d^0_{24}, d^0_{21} + d^0_{14}\} = \min\{3, \infty + \ldots\} = 3 \end{aligned}$$

  and so on to get

$$D^0 = \begin{pmatrix} 0 & 2 & \infty & 2 \\ \infty & 0 & \infty & 3 \\ 4 & \infty & 0 & \infty \\ \infty & \infty & 2 & 0 \end{pmatrix} \to D^1 = \begin{pmatrix} \mathbf{0} & \mathbf{2} & \infty & \mathbf{2} \\ \infty & \mathbf{0} & \infty & 3 \\ \mathbf{4} & 6 & \mathbf{0} & 8 \\ \infty & \infty & 2 & \mathbf{0} \end{pmatrix}$$

- And similarly we get D^2, D^3 and D^4

2 Minimum Spanning Trees

2.1 The Algorithms of Prim and Kruskal

Trees

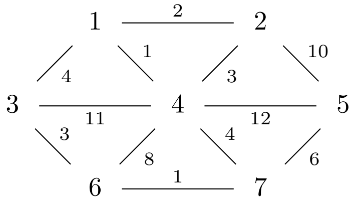
- An undirected graph $G = (V, E)$ is **connected** if for every pair $u, v \in V$ there is a path π in E from u to v
- A **cycle** π in a graph $G = (V, E)$ is a path that starts and ends at the same point
- A **tree** is an undirected connected graph that is also **acyclic**, i.e., there are no cycles in E
- A tree T is a **spanning tree** (ST) for $G = (V, E)$ if $T = (V, E_T)$ with $E_T \subset E$
- If G is weighted, the **cost** of an ST T is

$$c(T) = \sum_{(u,v)\in E_T} c(u, v)$$

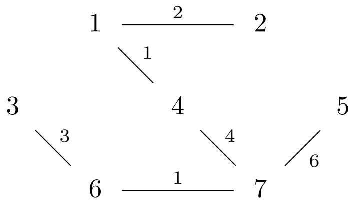
- $T = (V, E_T)$ is a **minimum spanning tree** (MST) for the undirected graph $G = (V, E)$ if for any other ST $T' = (V, E'_T)$ we have $c(T) \leq c(T')$

MST Examples

- On the graph



a first MST with cost 17 is



Prim's Algorithm

- Changing slightly Dijktsra's gives **Prim's** algorithm for finding MSTs

```
def prim(G, u):
    s[ ] = F; p[ ] = NULL; c_t[ ] = inf          # 1
    Q = pq()
    c_t[u] = 0; Q.put( (c_t[u], u) )
    while not Q.empty():                         # 2
        _, v = Q.get()                           # 3
        if not s[v]:
            s[v] = T
            for all z adjacent to v:             # 4
                if not s[z] and c_t[z] > c(v, z):
                    c_t[z] = c(v, z)
                    p[z] = v
                    Q.put( (c_t[z], z) )         # 5
    return p, c_t
```

- The second `if not s[z]` didn't appear in Dijkstra; do we need it here?

Observations on MSTs

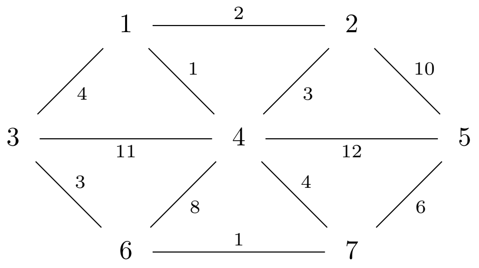
- There may be several minimum spanning trees in a graph
- We have $c(T) = \sum_{v \neq u} c(p[v], v)$
- The cost of Prim is $O(|E| \log |V|)$ if the PQ is built over a min heap
- **Prim works**: at the end $p[v]$ gives the edges $(p[v], v)$ of a MST E_T and $cE[v]$ its costs
  - But again this has to be proved
  - And we do not need to check `s[v] == T` (although it saves time) for if `z` already seen,
    `c_t[z] <= c(v, z),`
    since it is correct,
- Prim and Dijkstra are examples of a **greedy** algorithms

Greedy Algorithms

- A greedy algorithm tries to solve a **global optimization problem** by making **locally optimal choices** at each of its steps
- Simple example: the Nearest Neighbor algorithm for the **Traveling Salesman Problem** (TSP)
- In Dijkstra: we maintain a table $d[v]$ of **partially minimum distances** from u to v computed over a subset of all paths from u to v
- In Prim maintain a table $c_t[v]$ of **locally minimum edge costs** of a partial spanning subtree that is progressively grown from a starting node u
- The greedy strategy is quite natural, but Dijkstra and Prim are not obviously correct algorithms
  - A too simple greedy often results on wrong algorithms, with greedy TSP an example
- **Kruskal's** is another, clearer example of a greedy algorithm to obtain a MST

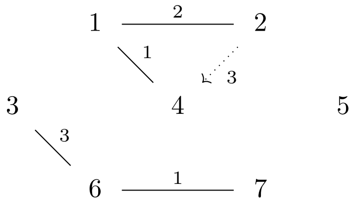
A First Look at Kruskal's Algorithm

- Main idea: sort the edges of E in a PQ by increasing costs and build a forest of partial STs
  - Starting from single node trees $T_u = (\{u\}, \emptyset)$ and
  - Adding edges from the PQ that do not produce cycles
- Example:



How to Apply Kruskal?

- Solving ties lexicographically, the sorted edges are
  `(1,4), (6,7), (1,2), (2,4), (3,6), (1,3), (4,7), (5,7), (4,6), ...`
- We add edges to the partial ST as



- But trying to add `(2,4)` we get a **cycle**, so we drop it and add next `(3,6)`
- We keep going until we get an MST

Elements of Kruskal's Algorithm

- To implement Kruskal we need a PQ, a way of storing the selected edges and a way to maintain the forest of partial subtrees and to detect cycles
- No problem with the PQ and we can simply gradually build the final MST graph
- At first sight maintaining trees and detecting cycles in them looks complicated and costly
- However, observe that (u, v) **gives a cycle iff** u **and** v **are in the same subset** $V_{T'}$ of the vertices of a tree T' in the Kruskal forest
  - `2` and `4` are in the set $\{$`1, 2, 4`$\}$
  - Thus we do not need to work with trees but with **subsets**
- We do this with a new abstract data type, the **Disjoint Set**

2.2 The Disjoint Set Abstract Data Type

Disjoint Set

- A **Disjoint Set** (DS) S over a universal set U is a dynamic family of disjoint subsets of U (i.e., a **partition** of U), each of which is **represented** by a certain element x and that has the following primitives:
  - `init_DS(U, S)`: receives the universal set U and returns the initial S as the fam**i**lity of atomic subsets $\{\{u\} : u \in U\}$
  - `find(x, S)`: receives an element $x \in U$ and returns the representative of the subset S_x of S that contains x
  - `union(x, y)`: receives two representatives x, y, computes their union $S_x \cup S_y$ and returns the representative of the subset $S_x \cup S_y$

Observations on the Disjoint Set

- The subsets of a Disjoint Set are never split; they can only change to bigger subsets
  - The Disjoint Set is never empty
- After `init_DS` we start with a partition with $|U|$ subsets;
  - Thus, the maximum number of unions is $|U| - 1$
- Even if we don't have yet a data structure for DS, its primitives allow us to write a first pseudocode for Kruskal

Kruskal's Algorithm

```
def kruskal(G):
    L = []                             #we save the MST in L
    init_DS(V, S)                      # 1

    Q = pq()
    for all (u, v) in E:
        Q.put( (c(u, v), (u, v)))      # 2

    while not Q.empty:                 # 3
        _, (u, v) = Q.get()            # 4
        x = find(u, S)
        y = find(v, S)                 # 5
        if x != y:
            L.append( (u, v) )         # 6
            union(x, y, S)             # 7

    return L
```

Observations on Kruskal's Algorithm

- We simply store E_T on a list L (when writing a program this may change)
- The algorithm may return a faulty ST, for instance if G is not connected
  - We can control this introducing a counter c and increasing it when a new edge is added to L
  - c should have the value $|V| - 1$ when the PQ is empty
  - Exercise: add it
- The maximum number of unions is $|V| - 1$
- Even if we achieve a efficient implementation of `union` and `find`, the cost of Kruskal will be at least $O(|E| \log |V|)$ because of building the PQ in (1)

A Data Structure for DS

- Our second data structure stores DS as trees (not to be confused with the conceptual subtrees that Kruskal builds)
- The representative x of a subset S is at the root of the subset tree T_S

- The cost of `union(x, y, S)` is then just $O(1)$, as we simply make, say, T_{S_y} a child subtree of the x root
- To implement `find(u, S)` we need a fast way to first locate the tree of u and then to go from the u node to the root
- This can be easily done if we place the subsets on a table `p[ ]`:
  - `p[u]` is the index of the father of `u`
  - `p[x]=-1` for a root `x`, i.e., a representative

Union and Find over Trees

- To initialize the DS we simply need `p[i]=-1` for all i
- The simplest pseudocode for `find` is

```
def find(u, p):
    while p[u] != -1:
        u = p[u]
    return u
```

- The pseudocode for `union` is

```
def union(x, y, p):
    p[y] = x    #join second tree to first
    return x
```

Improving Union

- Since the cost of `find` is $O($ height $(T_x))$ it is clear that we should join the shorter tree into the taller one
- For this we need to keep a tree's height h
  - We simply can change $p[x]$ at the root x from -1 to $-h$
- We then change the pseudocode for `union` as

```
def union_height(x, y, p):
    if p[y] < p[x]:           #T_y is taller
        p[x] = y; return y
    elif p[y] > p[x]:         #T_x is taller
        p[y] = x; return x
    else:                     #T_x, T_y have the same lenght
        p[y] = x; p[x] -= 1; return x
```

- We also change the while condition on `find` to

```
while p[u] >= 0:
```

The Cost of Find

- **Proposition.** *If prof(T) denotes the depth of a DS tree T, we have prof(T)* $\leq \lg |T|$
- **Proof Sketch:**
  - Use induction on $|T|$, with an obvious base case $|T| = 1$

- Assume it true for $|T'| < |T|$ and that we join T_y into T_x
- If $\text{prof}(T_y) < \text{prof}(T_x)$,

  $$\text{prof}(T_x \cup T_y) = \text{prof}(T_x) \leq \lg |T_x| \leq \lg |T_x \cup T_y|$$

  and the same argument works when $\text{prof}(T_x) < \text{prof}(T_y)$,
- If $\text{prof}(T_y) = \text{prof}(T_x)$ and $|T_y| \leq |T_x|$,

  $$\begin{aligned} \text{prof}(T_x \cup T_y) &= 1 + \text{prof}(T_y) \leq 1 + \lg |T_y| = \lg \ 2|T_y| \\ &\leq \lg |T_x \cup T_y| \end{aligned}$$

Improving Find

- Thus, the cost of `find(x, p)` is also $O(\log |S_x|) = O(\log N)$
- We thus get the same cost as before but with a much simpler data structure.
- Moreover, we can further improve on this
- Observe that when finding the representative of u we also find the **representative of all the v between u and the root of its tree**
- We can thus change `find` to update `p[v]` for al v between u and the root
- In other words, we can **compress the path** from u to the root

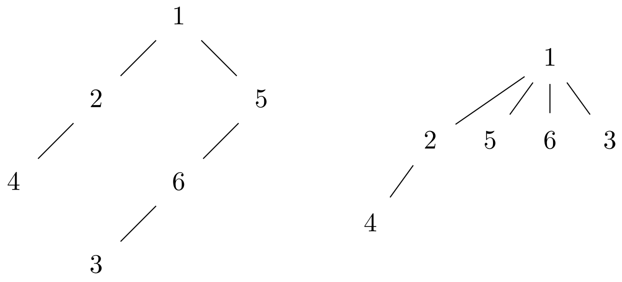
Path Compression

- Recall that after finding the representative of u, we also know it for all the other nodes between u and the root of the tree
- We thus improve `find` as follows:

```
def find_cc(u, p):
    # find the representative
    z = u
    while p[z] >= 0:
        z = p[z]

    # compress the path from u to the root
    while p[u] >= 0:
        y = p[u]
        p[u] = z
        u = y
    return z
```

The effect of `find_cc`

- Left: tree state after `find(3)`; right: state after `find_cc(3)`



Path Compression and Union by Rank

- The problem is now that, after `find`, we no longer have in $-p[x]$ the tree's height
- We do nothing about this other than calling $-p[x]$ the tree's **rank**
- We change nothing on `union` although it is no longer a union by height but a **union by rank**
- However the joint cost of unions and finds considerably improves
- **Proposition:** *If on a DS with N elements we do L unions by rank and $M = \Omega(N)$ path compression finds, the overall cost is*

$$O(L + M \lg^* N)$$

The $\lg^*$ Function

- We define $\lg^* H = K$ if K is the smallest integer such that after K binary logs we have

$$\lg(\ldots \lg(\lg H) \ldots) \leq 1$$

- For instance $\lg^* 65536 = \lg^* 2^{16} = 4$, but then

$$\lg^* 2^{65536} = 1 + \lg^* 2^{16} = 5$$

- Now 2^{65536} is a huge number:
  - Find out how many digits its decimal expression has (easy)
  - Then try to write it using millions, billions, googols and so on! ;-)
- For practical purposes $\lg^* H = O(1)$

Back to Kruskal's Algorithm

- Assume we work with union by rank and path compression and go back to Kruskal's pseudocode

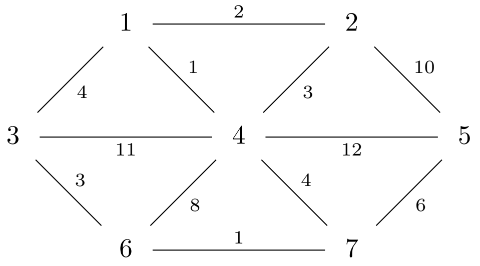
```
def kruskal(G):
    l_mst = []                      # we save the MST in L
    p = init_DS(V)                  # 1
    Q = pq()

    for all (u, v) in E:
        Q.put( (c(u, v), (u, v)))   # 2

    while not Q.empty:              # 3
        _, (u, v) = Q.get()         # 4

        x = find(u, p)              # 5
        y = find(v, p)

        if x != y:
            l_mst.append( (u, v) )  # 6
            union(x, y, p)          # 7

    return l_mst
```

The Cost of Kruskal's Algorithm

- Clearly the cost of (1) is $O(|V|)$ and that of (2) is $O(|E| \log |V|)$
- The cost of (4) accumulated over (3) is again $O(|E| \log |V|)$
- Since the single cost of (6) and (7) is $O(1)$ and only happens when `x!=y`, their accumulated costs are $O(|V|)$
- Finally, since we must do at least one `findPC` for each node, the total number is $\Omega(N)$ and, therefore, the cost of (5) accumulated over (3) is $O(|E| \lg^* |V|)$, that is, essentially $O(|E|)$
- Summing things up, the cost of Kruskal is $O(|E| \lg |V|)$, dominated by the PQ operations
- In particular the DS operations do not penalize the algorithm

Applying Kruskal's Algorithm

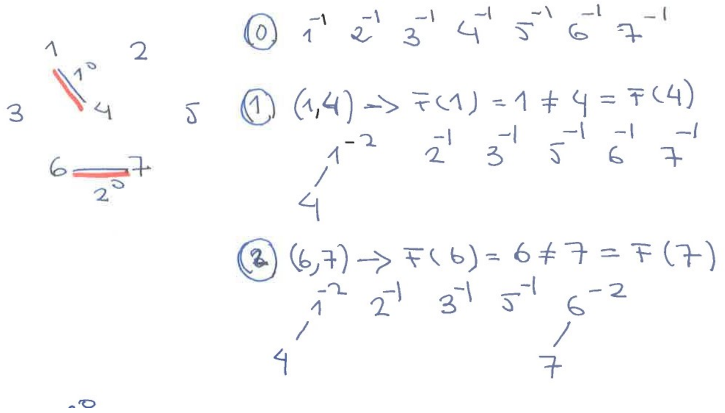
- **Example:**



- The PQ is `(1,4), (6,7), (1,2), (2,4), (3,6), (1,3), (4,7), (5,7), (4,6), (2,5), (3,4), (4,5)`

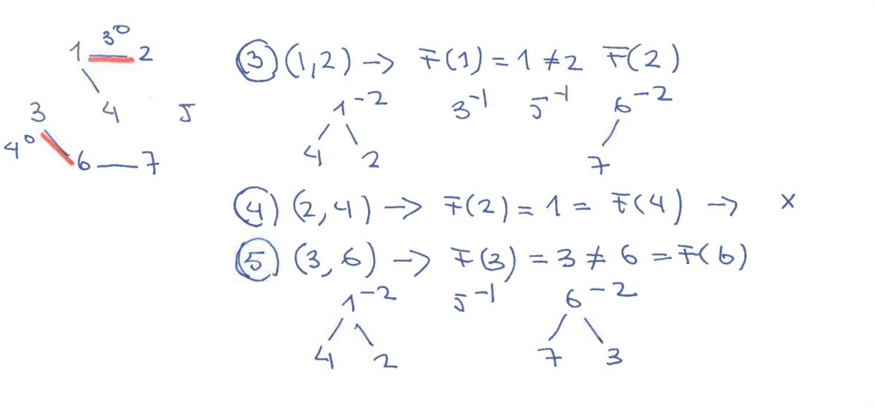
Applying Kruskal's Algorithm (II)

- We maintain separately the Kruskal forest and the DS forest



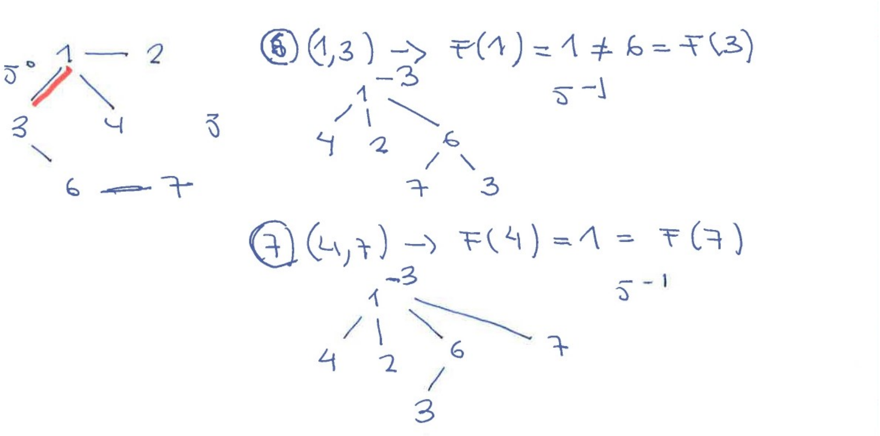
Applying Kruskal's Algorithm (III)

- We process the remaining edges from the PQ


1 3° 2
3 4 5
4° 6 7
③ (1,2) -> F(1) = 1 ≠ 2 F(2)
1 -2 3 -1 5 -1 6 -2
4 2 7
④ (2,4) -> F(2) = 1 = F(4) -> x
⑤ (3,6) -> F(3) = 3 ≠ 6 = F(6)
1 -2 5 -1 6 -2
4 2 7 3

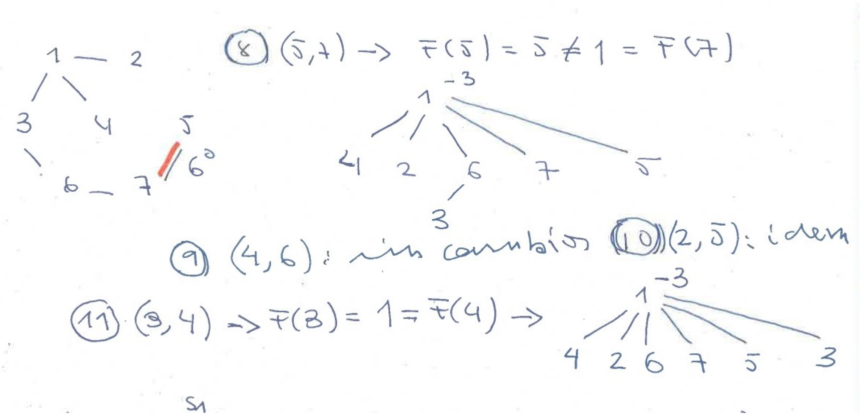
Applying Kruskal's Algorithm (IV)

- We process the remaining edges from the PQ



Applying Kruskal's Algorithm (V)

- We process the remaining edges from the PQW until it is empty
- The MST may not change but the DS forest may


8 (5,7) -> F(5) = 5 ≠ 1 = F(7)
-3
9 (4,6): sin cambios
10 (2,5): idem
11 (3,4) -> F(3) = 1 = F(4) ->
-3

2.3 Correctness of Prim and Kruskal

Cuts and Minimal Crossings

- Assume we have an undirected weighted graph $G(V, E)$ with cost c
- A **cut** P of G is a partition of V into two disjoint subsets $P = (S, V - S)$
- An edge (u, v) **crosses** P if either $u \in S$ and $v \in V - S$ or viceversa
- A subset $A \subset E$ **preserves** P if no edge in A crosses P
- An edge (u, v) that crosses P is **minimal** w.r. to P if $c(u, v) \leq c(w, z)$ for any other edge (w, z) that crosses P

A Meta MST Algorithm

- Consider the following meta–algorithm to find MSTs

```
def metaMST(G, c):
    L = [ ]                 #we see L as a subset of E

    while len(L) < |V|:
        find a cut P that preserves L
        select (u, v) minimal w.r. to P
        L.append( (u,v) )

    return L
```

- Notice that `metaMST` is also a kind of greedy meta–algorithm
  - At each step a minimal edge is added to the MST list

Prim as an Example of metaMST

- Recall that Prim works with a table $v[\,]$ of nodes "seen" and that the nodes still in Q are ordered by their cost at insertion
- Assume that a node v has been extracted from Q just before is marked as seen, and take

  - $P = (\{seen\ nodes\}, \{others\})$
  - $L = \{(p[w], w) : w \in \{seen\ nodes\}\}$

- Then we have

  1. L preserves P for if $(p[w], w) \in L$, both w and $p[w]$ are seen
  2. $(p[v], v)$ crosses P, for v is unseen but $p[v]$ was processed when v entered Q, i.e., it is seen by now
  3. If other (w, z) crosses P we have $v[w] = T$, $v[z] = F$ and, hence, $z \in Q$ since it is adjacent to the already seen node w
  4. Since we extract v but not z, $c(p[v], v) \leq c(w, z)$ and, thus, $(p[v], v)$ is minimal

- Hence, Prim is a particular case of metaMST

Kruskal as an Example of metaMST

- Assume that we are about to add the edge (u, v) and let

  - $L = \{(w, z)\}$ be the edges already selected
  - $P = (S_u, V - S_u)$ where S_u is the subset of the tree T_u that contains u

- Then we have

  1. L preserves P for the subtrees are disjoint and if $(w, z) \in L$, they are in the same subtree T, which cannot happen if $w \in S_u$ and $z \in V - S_u$
  2. (u, v) crosses P: if not, it would make a cycle in T_u
  3. Any other (w, z) crossing P must connect different subtrees and cannot make a cycle
  4. Thus (w, z) must still be in Q: it can only be been discharged if w and z were in the same subtree
  5. Thus, $c(u, v) \leq c(w, z)$ and (u, v) is minimal w.r. P

- Hence, Kruskal is a particular case of metaMST

A Kruskal metaMST step

- In the previous example, assume we are going to add $(1, 3)$

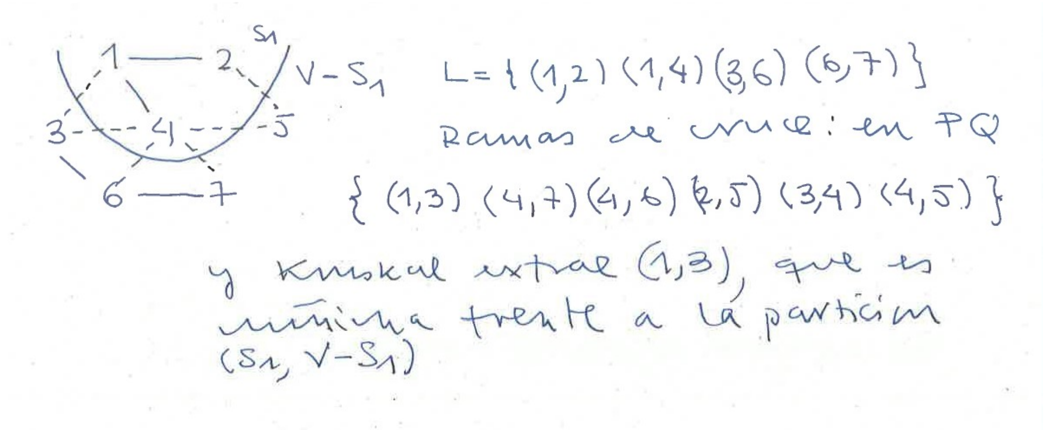
- The partition, the preserving edges and the crossing ones are

Correctness of metaMST I

- Thus, if metaMST is correct, Prim and Kruskal will also be correct

- **Proposition.** *Let $G = (V, E)$ be a undirected, connected, weighted graph and assume $A \subset E$ verifies $A \subset E_T$ for some MST T. Then,* **if A preserves some P and (u, v) is minimal w.r. to P, we have $A \cup \{(u, v)\} \subset E_{T'}$ for some MST T'**

- **Proof sketch:**



L = { (1,2) (1,4) (3,6) (6,7) }

Ramas de cruce: en PQ

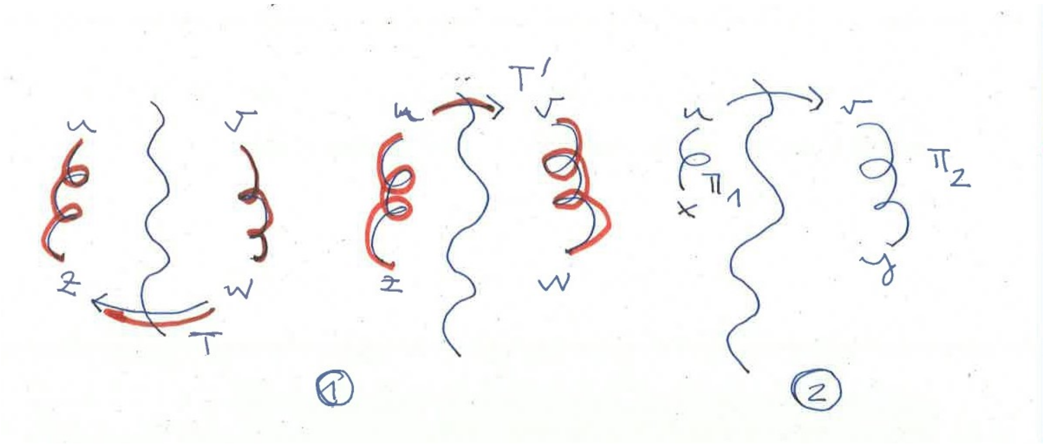
{ (1,3) (4,7) (4,6) (2,5) (3,4) (4,5) }

y Kruskal extrae (1,3), que es mínima frente a la partición (S1, V−S1)

- Assume $T = (V, E_T)$; then $\pi = E_T \cup \{(u, v)\}$ is a cycle with an edge (w, z) that crosses P
- Define $T' = (V, E_{T'})$ with $E_{T'} = (E_T - \{(w, z)\}) \cup \{(u, v)\}$; clearly $c(T') \leq c(T)$ and to prove T' a tree
- Since $V_{T'} = V$, we just check T' is connected

Correctness of metaMST II

- **Proof sketch:**



Correctness of metaMST III

- **Proof sketch II:**
  - Assume x, y at different sides of P with x, u and y, v in the same sides
  - There are paths π_1 from x to u and π_2 from v to y in T and hence T'
  - Then $\pi = \pi_1 \cup \{(u, v)\} \cup \pi_2$ is a path in T' from x to y

  - Thus T' is connected, $c(T') \leq c(T)$ and $V_{T'} = V$
  - Thus T' is an MST

Loop Invariants

- The proposition says that **after each iteration the selected edges are part of a MST**
- This is an example of a **loop invariant**:

  A condition that remains true after each loop and that "leads" the algorithm towards a correct solution
- The standard way to prove the correctness of an iterative algorithm is to find an adequate loop invariant for its iterations
- Example: loop invariants for InsertSort or BubbleSort
  - InsertSort: after iteration i, $i = p+1, \ldots, u$, the subtable $T[p], \ldots, T[i]$ is sorted
  - BubbleSort: after iteration i, $i = u, \ldots, p+1$, the subtable $T[i], \ldots, T[u]$ is sorted

Correctness of metaMST II

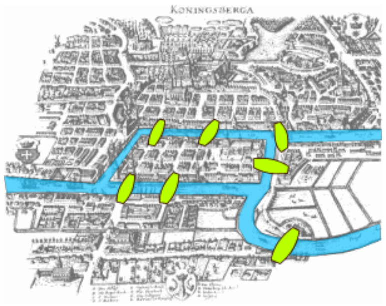
- **Corollary.** *`metaMST` returns a MST*
- **Proof sketch:**
  - We just exploit the loop invariant provided by the previous proposition
  - Let $L_0 = \emptyset \subset L_1 \subset \ldots \subset L_{N-1}$ be the successive subsets `metaMST` produces
  - If L_j is a subset of some MST, the proposition shows that so is L_{j+1}
  - But obviously L_0 is a subset of some MST and, thus, so is L_{N-1} and since it has $N-1$ edges, (V, L_{N-1}) is a MST
- **Corollary** *Prim and Kruskal also return MSTs*

3 Eulerian and Hamiltonian Circuits

3.1 Eulerian Circuits

The Bridges of Königsberg

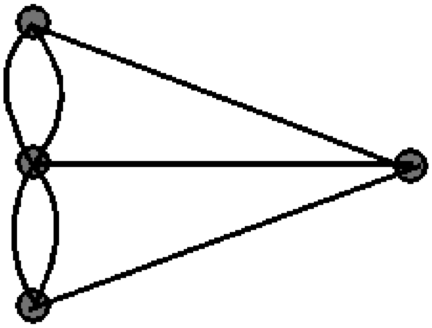
- The bridges of Königsberg (East Prussia) over the Pregel river circa 1700:



- The problem: find a promenade that crosses all bridges but only once
- Exercise: google `pregel graph`

The Bridges of Königsberg as a Graph Problem

- We can depict the bridges of Königsberg as a multigraph (i.e., we allow for multiple edges between two nodes)



- The problem: find a circuit that passes **through all edges but only once**
- Such a circuit in a multigraph is called an **Eulerian circuit** (EC)

Euler's Insight

- Leonhard Euler showed in 1736 (*Solutio problematis ad geometriam situs pertinentis*) that such a circuit is not possible
- If G is an undirected graph, we define the **degree** $deg(w)$ of a node w as the number of edges that leave w (or that enter w or simply the size of $T[w]$)
- Assume that $\pi = \{(u = u_0, u_1), \dots, (u_{K-1}, u_K = u)\}$ is an EC for G
- If $w \neq u$ is a node in π, each time we enter w we substract 1 from $deg(w)$ and also when we leave w
  - Since at the end we have passed by all the edges of w, **we must have at the beginning** $deg(w)$ **even**
- Similarly each time we enter u inside π we substract 1 from $deg(u)$ and also when we leave u; moreover, when we start and end π we also substract 1 from $deg(u)$
  - Thus, **we must also have** $deg(u)$ **even**

There Are No ECs in Königsberg

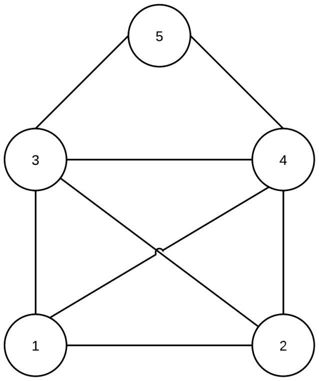
- It follows from the previous analysis that a necessary condition to have an EC is that $deg(v)$ is even for all $v \in V$

- Since all the nodes in the previous multigraph have odd degrees, Euler concluded that no Eulerian circuit is possible in Königsberg
- As we shall see later, Euler also proved that the condition is sufficient:

  If $deg(v)$ is even for all nodes v of an undirected graph G, then there is an Eulerian circuit in G

Drawing Houses Without Lifting the Pen

- A child's game is to try to draw the house below without lifting the pen from the sheet



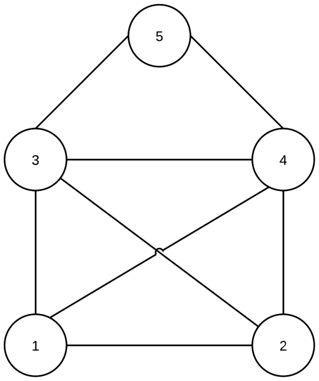
- It is very easy if we start at nodes 1 or 2 but impossible if we start from 3, 4 or 5

Euler's Insight Again

- Assume that $\pi = \{(u = u_0, u_1), \ldots, (u_{K-1}, u_K = v \neq u)\}$, $u \neq v$, is such an **Eulerian path** (EP)
- If $w \neq u, v$ is a node in π, each time we enter w we substract 1 from $deg(w)$ and also when we leave w;
  - Since at the end we have passed through all the edges of w, we must have at the beginning $deg(w)$ even
- Similarly each time we enter u inside π we substract 1 from $deg(u)$ and also when we leave u; moreover, since we start π at u, we also substract 1 from $deg(u)$
  - Thus, we must also have $deg(u)$ odd
- Similarly each time we enter v inside π we substract 1 from $deg(v)$ and also when we leave v; moreover, since we end π at v, we also substract 1 from $deg(v)$
  - Thus, we must also have $deg(v)$ odd
- Thus, **a necessary condition to have an EP is that $deg(w)$ is even for all w except the first node u and the final one v of π**

Back to Drawing Houses

- Since $deg(1) = deg(2) = 3$ we can find an EP for the house drawing if we start at either 1 or 2



- But since $deg(3) = deg(4) = deg(5)$ even, it is impossible to draw an EP for the house starting at them
- And there is no EP in Königsberg either.

Euler's Theorem for Circuits

- **Theorem.** *If* $G = (V, E)$ *is a connected undirected* **multigraph***, there is an EC in G iff* $deg(u)$ *is even for all* $u \in V$

  Proof sketch: We argue by induction on $|V|$

  - The theorem is obviously true if $|V| = 2$ and assume it also to be true for any $G' = (V`, E')$ such that $|V'| < |V|$
  - We start walking from a node u substracting from deg at each node until we arrive at v such that $deg(v) = 0$ after we enter v and, thus, cannot leave it
  - It is easy to see that $v = u$ and we have found a cycle π
  - We remove E_π from E and from V the nodes w whose deg π has exhausted and let $G' = (V', E')$ be the resulting graph
  - Since $|V'| \leq |V| - 1$ and $deg_{G'}(w) = deg_G(w) - deg_\pi(w)$ is even, we can apply induction on the connected components $G_1, \ldots, G_K$ of G'
  - By induction there are ECs π_k in the G_k that start at nodes from π and we get an EC on G "collating" π and the π_k

Euler's Theorem for Paths

- **Corollary.** *If G is a connected undirected graph, there is an EP* π *in G iff* $deg(w)$ *is even for all* $w \in V$ *except for two vertices* u *and* v*. Moreover, then* π *starts at* u *and ends at* v *or viceversa*

  Proof sketch: We just show the condition to be sufficient:

  - Consider $G' = (V, E' = E \cup \{(u, v)\})$, i.e., we add an extra edge (u, v) to E

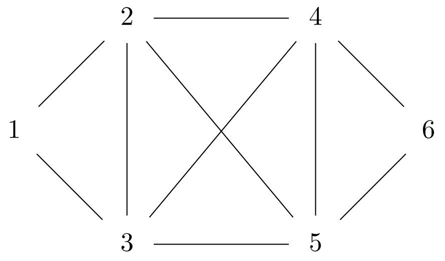
  - Since $deg_{G'}(u) = deg_G(u) + 1$, $deg_{G'}(v) = deg_G(v) + 1$ and $deg_{G'}(w) = deg_G(w)$ for all other w, all the G' degrees are even and there is an EC π' in G'
  - Let's write π' as $\pi' = \{(v, z), \ldots, (w, u), (u, v)\}$, with the last edge the one we added to get G'.
  - Then removing this edge we get the EP $\pi = \{(v, z), \ldots, (w, u)\}$.

How to Find an EC

- Assuming an EC exists, the basic idea is simply to follow the proof's argument
- We start at any u_1 and build $\pi_1 = \{(u_1, v_2), \ldots, (v_{K-1}, v_K)\}$ substacting 1 from $deg(w)$ each time we enter or leave w and where we stop because after entering v_K we have $deg(v_K) = 0$
  - It is then clear that $u_1 = v_K$, and
- Let $G_1 = (V_1, E_1)$ the graph obtained after removing π_1 from E and all the $w \in V$ for which $deg(w) = 0$ after π, i.e., for which $deg_\pi(w) = deg_G(w)$
  - Clearly u_1 at least will be removed, i.e., $|V_1| < |V|$
  - If $|V_1| = 0$, clearly π_1 is an EC in G
  - If however $|V_1| > 0$, there is a first u_2 in π_1 such that $deg_{G_1}(u_2) > 0$
  - We can thus **restart the above process on** G_1 obtaining a new circuit π_2 and a "remaining" graph G_2
- If we repeat the preceeding and find circuits $\pi_1, \ldots, \pi_M$ until $V_M = \emptyset$, then we can "collate" the π_j circuits to get an EC π for G

How to Find an EC II

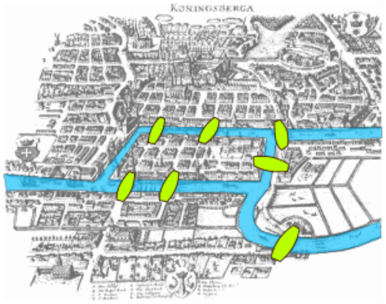
- **Example:**



- We do not write a pseudocode (good exercise!) but it is clear that its cost will be $O(|E|)$

EC Steps

- The steps to find an EC are



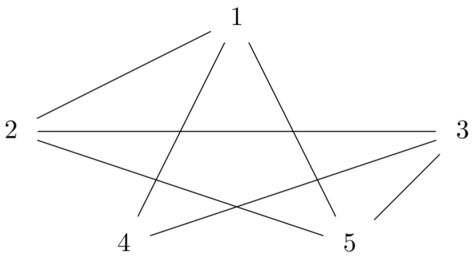
3.2 Hamiltonian Circuits and an Excursion on Complexity Theory

Hamiltonian Circuits

- If G is an undirected connected graph, a **Hamiltonian circuit** (HC) is a circuit on G that visits **only once each node** other than the initial
- Finding HCs may be trivial in some cases, such as complete graphs
- There are also sufficient conditions for special graphs
- But for general graphs, while finding ECs has an $O(|E|)$ cost, finding HCs is much costlier
- In fact, essentially the only general algorithm is an exhaustive search with backtracking

Hamiltonian Circuits II

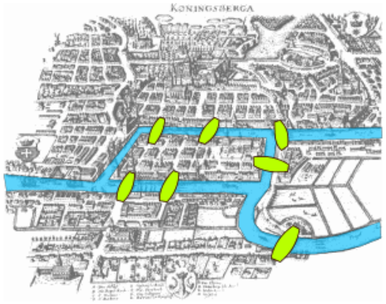
- **Example:**



- Since the number of node orderings is $N!$, the search's cost can be very high
- Actually, finding HCs in general graphs is an example of an **NP–complete** problem

Backtracking Search

- An example of a HC search



P and NP I

- We will make a brief (and light) excursion on Complexity Theory
- We consider **decision problems** $\mathcal{P}$ for which there is a set of solution inputs $S_{\mathcal{P}}$, i.e., the decision on an input I is 1 iff $I \in S_{\mathcal{P}}$
  - To decide whether a graph has an EC or HC is a decision problem but notice that an algorithm does not have to actually find an EC or HC to solve them
  - Optimization problems can be partially reduced to decision problems using a bound C: change *find the optimum* by *find a solution with cost* $\leq C$
- For an input I we can consider its size $|I|$ to be the number of bits needed to store it
- We say that $\mathcal{P}$ **is in the class** P if there is an algorithm A with cost polynomial on $|I|$ that solves $\mathcal{P}$, i.e., $A(I) = 1$ iff $I \in S_{\mathcal{P}}$
  - Note that to be in class P does not mean that A is efficient: if its cost is $O(|I|^{1000})$, $\mathcal{P}$ is in P

P and NP II

- Decision–EC is in P: we check in linear time whether or not there are ECs in G by counting degrees and checking that they are even
- An algorithm $C(I, S)$ is a **certifier** for $\mathcal{P}$ if
  - For every input $I \in S_{\mathcal{P}}$ there is another, different input $S = S(I)$ to C such that $C(I, S) = 1$
  - If $I \notin S_{\mathcal{P}}$, then $C(I, S) = 0$ no matter which S is used
- S is a kind of certificate (solution?) that the certifier validates
  - For the EC or HC problems, S can just be a possible EC or HC
- We say that $\mathcal{P}$ **is in the class** NP if there is a certifier C that runs in polynomial time on the sizes $|I|$ and $|S|$

  - For instance, if $I = G$ and S is a possible CH, we can check it in polynomial time;
  - Thus HC belongs to NP

P and NP III

- Clearly $P \subset NP$: if $\mathcal{P} \in P$ and A solves it, set $C(I, S) = A(I)$; then
  - If $I \in S_{\mathcal{P}}$, we can simply use an empty certificate and set $C(I, \emptyset) = A(I)$
  - If $I \notin S_{\mathcal{P}}$, we will have $C(I, S) = A(I) = 0$ no matter the S presented
- Big question: $P = NP$?
- If yes, there would be a polynomial time algorithm for HC
- It is one of the Millenium Problems of the Clay Mathematics Institute with a 1M $ prize
  - For more details see `http://www.claymath.org/millennium/P_vs_NP`
- General opinion: $P \neq NP$
- Reason: NP**–complete problems**

NP–complete Problems

- We say that $\mathcal{P}_1$ is **reducible** to $\mathcal{P}_2$ if there is a map

$$T : \{ \text{ inputs of } \mathcal{P}_1 \} \to \{ \text{ inputs of } \mathcal{P}_2 \}$$

  such that I_1 has a solution for $\mathcal{P}_1$ iff $T(I_1)$ has a solution for $\mathcal{P}_2$

  - Or: $I \in S_{\mathcal{P}_1}$ iff $T(I) \in S_{\mathcal{P}_2}$
- Thus, **if** A **is an algorithm that solves** $\mathcal{P}_2$**, then** $A \circ T$ **solves** $\mathcal{P}_1$:

$$I \in S_{\mathcal{P}_1} \text{ iff } T(I) \in S_{\mathcal{P}_2} \text{ iff } A(T(I)) \equiv A \circ T(I) \equiv 1$$

- If T has polynomial cost, we say that $\mathcal{P}_1$ is **polynomially reducible** to $\mathcal{P}_2$
- We say that problem $\mathcal{P}$ is NP**–complete** if any other $\mathcal{P}' \in NP$ is polynomially reducible to $\mathcal{P}$
- Notice that **if we show for just one** NP**–complete problem** $\mathcal{P}$ **that** $\mathcal{P} \in P$**, then we have proved that** $P = NP$

Is There Any NP**–complete Problem?**

- At first sight the NP–complete definition seems very strict so a natural question is whether there any such problem
- Answer: yes, and in fact many!! HC is such a problem
- The first (basically) NP–complete problem found is 3–SAT

- Given a Boolean expression B written using only AND, OR, NOT operators, and parentheses, the **satisfiability problem (SAT)** is to decide whether there is some assignment of T and F to the variables that will make B true
- The k–SAT problem deals with expressions in **conjunctive normal form** (i.e., as a sequence of OR clauses joined by AND) with k variables or their negation per clause

Cook's Theorem

- Example: 3–SAT deals with expressions like

```
(x11 OR !x12 OR x13)AND (!x21 OR x22 OR !x23)AND
(x31 OR !x32 OR x33)AND ...
```

- **Cook's Theorem (1971)**: 3–SAT is NP–complete
  - However, 2-SAT $\in P$
- More to read: Chapter 5 of H. Wilff's book **Algorithms and Complexity**
- Much more to read: M.R. Garey and D.S. Johnson. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. W.H. Freeman, 1979.
- But are P, NP and NP–complete problems just academic curiosities?

3.3 The Traveling Salesman Problem

The Traveling Salesman Problem

- TSP: Given a weighted complete graph G, find a HC (trivial) with minimum cost
- It is an optimization problem with obvious practical interest: many persons have to solve it every morning
  - Decision version: given a weighted complete graph G and a bound C, is there a HC π such that $c(\pi) \leq C$?
- TSP is **NP–hard**: every problem in NP can be polynomically reduced to TSP
  - A NP–hard problem may not have to be NP-complete (e.g., the halting problem) or to be a decision problem
  - Also, TSP–decision for general graphs is NP–complete
- But TSP–decision is also NP–complete for "real world" problem versions, such as for cities in the plane with Euclidean distances
- Many related problems of great practical interest in planning, logistics or DNA sequencing are also NP–complete

From TSP to HC

- Fact: **HC is polynomially reducible to TSP**

- Assume `tsp(V, c)` is a routine that returns the TSP solution for G with cost c and consider the following routine for HC:

```
def tsp_2_hc(V, E):
    for any u, v in V:
        if (u, v) in E:
            c(u, v) = 1
        else:
            c(u, v) = 2

    p = tsp(V, c)
    if cost(p) == |V|:
        return p
```

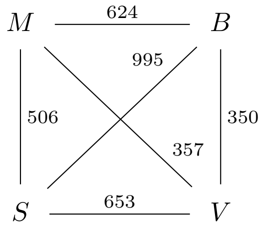
- `tsp_2_hc` solves HC for π is a HC on G iff $c(\pi) = |V|$
- Thus TSP has not only practical but also theoretical interest

A TSP Example

- Simple example:

```
cities = ["madrid", "barcelona", "sevilla", "valencia"]
```

- The (complete) graph is



- The solution is `M, V, B, S, M` check!!
- More examples in Traveling Salesman Algorithms

A Greedy TSP Solution

- Simple greedy approach: Nearest–Neighborhood (NN) TSP, that simply visits the nearest unseen city

```
def nn_tsp_circuit(distance_matrix, node_ini=0):

    num_cities = distance_matrix.shape[0]
    circuit = [node_ini]

    while len(circuit) < num_cities:
        current_city = circuit[-1]

        # sort cities in ascending distance from current
        options = list(np.argsort(distance_matrix[ current_city ]))

        # add first city in sorted list not visited yet
        for city in options:
            if city not in circuit:
                circuit.append(city)
                break

    return circuit + [node_ini]
```

What Can We Do About TSP?

- On average, NN gives a path that is about 25% longer than the optimum
- But one can set up special instances of TSP where NN gives the worst route
- If c satisfies the triangle inequality $c(u, v) \leq c(u, z) + c(z, v)$ for any z, we have

  $$c(\pi_{NN}) = O(\log |V|) \times c^*,$$

  with π_{NN} the NN solution and c^* the optimal cost
- TSP has great practical importance, but there is no cost effective **exact** algorithm for general graphs
- So, it may be very hard to find the best route to, say, deliver mail (at least in big cities)

Approximation Algorithms

- Alternative: **approximate** algorithms
- **Definition:** Given an optimization problem $\mathcal{P}$, an **approximate algorithm** for $\mathcal{P}$ **with bound** $\lambda \geq 1$ is an algorithm A that for every input I returns a solution $s_A(I)$ such that

  $$c^*(I) \leq c(s_A(I)) \leq \lambda c^*(I)$$

  with $c^*(I)$ the optimal cost for $\mathcal{P}$ on I
- NN is not exactly an approximate algorithm for TSP, since its bound is $O(\log |V|)$ and depends on $|V|$

Approximation Algorithms for TSP

- **Proposition:** *If the cost function is Euclidean, i.e., it verifies*

  $$c(u, v) \leq c(u, w) + c(w, v) \textit{ for all } u, v, w \in V,$$

  then there is an approximate algorithm for TSP with $\lambda = 2$
- Algorithm:

```
def euclideanTSP(g, c):
  find a MST t on g
  duplicate its edges to obtain a graph g_1

  #now each node in g_1 has degree 2 and there is an EC
  find a EC p_1 in g_1

  short--cut seen edges in p_1 to get HC p
  return p
```

Approximation Algorithms for TSP

- **Proof sketch:** Let T_1, p_1 and p be the MST, the Eulerian and the returned circuits in the previous algorithm

  Let p^* be an optimal HC and remove an edge on p^* to get a spanning tree T^*

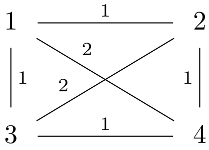
  - Since T_1 is an MST, we have $c(T_1) \leq c(T^*) \leq c(\pi^*)$
  - And using the Euclidean distance property, we then we conclude that

$$c(\pi) \leq c(\pi_1) = 2c(T_1) \leq 2c(\pi^*)$$

- The **Christofides** algorithm improves this to $\lambda = 1.5$ (see this article in Wired for more about the algorithm)
- To learn more: Johnson, McGeoch, The Traveling Salesman Problem: A Case Study in Local Optimization
  - Or the movie The Travelling Salesman

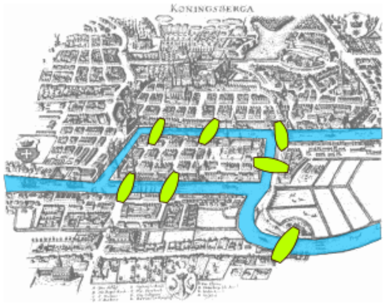
Approximation Algorithms for TSP II

- **Example**



Applying The Algorithm

- The steps to find an EC are



4 An Excursion on DNA Sequencing

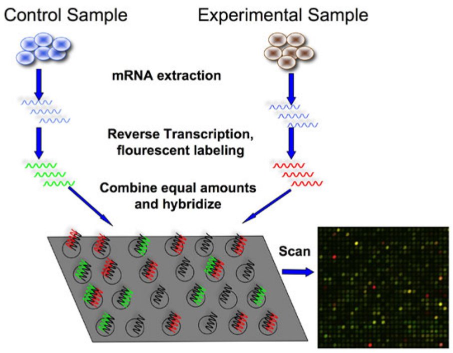
4.1 Hamilton, Euler and DNA Sequencing

DNA Sequencing

- **Note:** this is a very, very light description of DNA Sequencing
- Goal: decompose a gene into a sequence of four letters $\{A, C, G, T\}$ that correspond to DNA **bases**
- **Shotgun sequencing** follows a four step process:
  - Blast the gene into random short fragments (“reads”) of 100–500 bases
  - Identify read subsequences by hybridizing them on a DNA microarray
  - Reconstruct each read from these subsequences
  - Reconstruct the entire gene from the reads
- First two steps: biochemistry
- Third step: Hamiltonian or (better) Eulerian circuits
- Fourth step: compute the Shortest Superstring Problem solving TSP (plus more algorithms and a lot of biochemistry)

Sequencing by Hybridization II

- Scheme of the process:



From *bitesizebio.com/7206/introduction-to-dna-microarrays*

Microarray Hybridization

- Put all the posible lenght ℓ probes, i.e., DNA subsequences of a fixed lenght ℓ, into the spots of a microarray
- Put a drop of fluorescently labeled DNA into each microspot of the array

- The DNA fragment hybridizes with those microspots that are complementary to a certain substring of length ℓ of the fragment
- This way we get all possible lenght ℓ subsequences that make the fragment but they are **unordered**

ℓ–mers and the Spectrum

- We call the sequence on each one of the probes an **ℓ–mer**
- The **ℓ–spectrum** $sp(S, \ell)$ of a sequence S is the set of all the ℓ–mers from S
- For instance, `S = [TATGGTGC]` we have

  ```
  sp(S, 3)= {TAT, ATG, TGG, GGT, GTG, TGC}
  ```

- We have $|sp(S, \ell)| \leq |S| - \ell + 1$
- After hybridization, the hybridized probes in the microarray give us **an unordered version of** $sp(S, \ell)$ that we have to correct to recover S
- The overlap $\omega(s_1, s_2)$ between two ℓ–mers s_1, s_2 is the longest leght of a suffix of s_1 that is also a prefix of s_2
- We clearly have $\omega(s_1, s_2) \leq \ell - 1$ and if s_2 follows s_1 in S, we must have $\omega(s_1, s_2) = \ell - 1$

Sequencing by Hamiltonian Paths

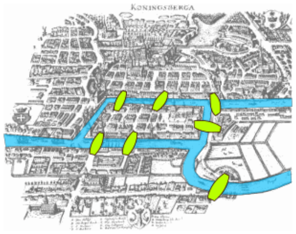
- We can reconstruct the sequence S by finding an ordering $s_{i_1}, \ldots, s_{i_K}$ of $sp(S, \ell)$ such that $\omega(s_{i_j}, s_{i_{j+1}}) = \ell - 1$
- This suggests to define the graph $G_\ell(S) = (V_\ell, E_\ell)$ where
  - $V_\ell = sp(S, \ell)$ and
  - $(s, s') \in E_\ell$ iff $\omega(s, s') = \ell - 1$
- Notice that reconstructing S is equivalent to pass once through all the nodes of $G_\ell(S)$
- In other words, we can reconstruct S by finding a Hamiltonian path in $G_\ell(S)$

Sequencing by Hamiltonian Paths II

- Example: consider `S = [TATGGTGC]` and the unordered 3–spectrum

  ```
  sp(S, 3)= {GGT, TAT, TGG, TGC, ATG, GTG}
  ```

- The adjacency list and graph are



- By inspection, the HC and the recovered sequence are

Sequencing by Eulerian Paths

- The obvious problem of HP sequencing is the lack of efficient algorithms to solve the HP problem
- The alternative is to try to have ℓ–mers on the edges instead of on nodes
- If $s \in sp(S, \ell)$ and s_1 is its $\ell - 1$ prefix and s_2 its $\ell - 1$ suffix, we can consider s as the edge connecting nodes s_1 and s_2
  - Now we have $\omega(s_1, s_2) = \ell - 2$
- We define now the graph $G_{\ell-1} = (V_{\ell-1}, E_{\ell-1})$ where
  - $V_{\ell-1} = sp(S, \ell - 1)$ and
  - $(s, s') \in E_{\ell-1}$ iff they are respectively prefix and suffix of an $s \in sp(S, \ell)$
- Notice that now reconstructing S is equivalent to pass once over all the edges of $G_{\ell-1}$
- In other words, we can reconstruct S by finding a EP in $G_{\ell-1}$

Eulerian Circuits on Directed Graphs

- However, notice that $G_{\ell-1}$ is a **directed** graph and we have to adapt the Eulerian circuit/path theory to these graphs
- In an directed graph $G(V, E)$ we have to distinguish between incident and adjacent edges
- For any $u \in V$, we say that (u, v) is an **adjacent** (outgoing) edge and (w, u) an **incident** (incoming) edge
- The **indegree** $in(u)$ of u is the number of incoming edges to u
- The **outdegree** $out(u)$ is the number of outgoing edges from u

Eulerian Circuits on Directed Graphs II

- Assume that $\pi = \{(u = u_0, u_1), \ldots, (u_{K-1}, u_K = u)\}$ is an Eulerian circuit on G
- If $w \neq u$ is a node in π, each time we enter w we substract 1 from $in(w)$ and also from $out(w)$ when we leave w
- Since at the end we have passed through all the edges of w, we must have at the beginning $in(w) = out(w)$
- Similarly each time we enter u inside π we substract 1 from $in(u)$ and also from $out(u)$ when we leave it

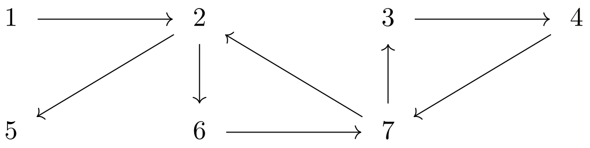
  Moreover, when we start we substract 1 from $out(u)$ and also substract 1 we substract 1 from $in(u)$ when we finish
- Thus, we must also have $in(u) = out(u)$

Euler‘s Theorem for Directed Graphs

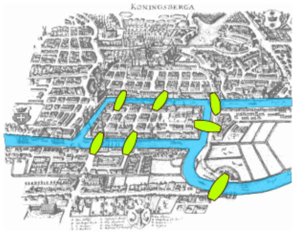
- **Euler’s Theorem.** *A neccesary and sufficient condition to have an EC in a directed G is that $in(v) = out(v)$ for all $v \in V$*
- **Corollary.** *A neccesary and sufficient condition to have an Eulerian path $\pi = \{(u = u_0, u_1), \ldots, (u_{K-1}, u_K = v\}$ in a directed graph G is that we have $in(w) = out(w)$ for all $w \in V$ different from u and v and also $in(v) = out(v) + 1$, $in(u) = out(u) - 1$*
- Essentially the same $O(|E|)$ algorithm we saw for undirected graphs can be applied to directed ones
- Thus we can efficiently sequence genomic reads

Applying Euler on Directed Graphs

- Consider the graph

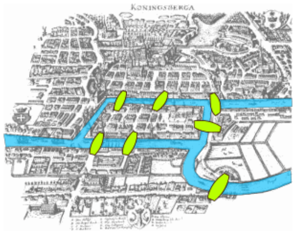


- The adjacency list and the first exploration give



Applying Euler on Directed Graphs II

- The second step yields



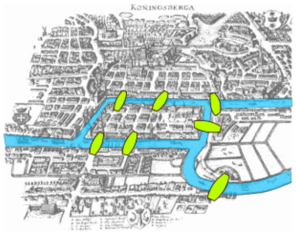
- And the final EC is

Eulerian Sequencing

- Example: consider again `S = [TATGGTGC]` and

  `sp(S, 2)= {TA, AT, TG, GG, GT, GC}`

- The adjacency list and graph are



- By inspection, the EC and the recovered sequence are

5 Depth First Search and Connectivity

5.1 Depth First Search

Breadth First Search (BFS)

- Recall the general pseudocode for BFS

```
def BFS(u, G):
    s[ ] = False; p[ ] = NULL; Q = pq()
    d[u] = 0; Q.put( (d[u], u) )
    do_something(u)
    while not Q.empty():
        _, v = Q.get()
        if not s[v]:
            s[v] = True
            for all z adjacent to v:
                do_something(z) #perhaps change d[z]
                if d[z] smaller:
                    p[z] = v; Q.put( (d[z], z) )
    return p
```

- If the cost of `do_something` is $O(1)$ and we work with a PQ, the cost of BFS is $O(|E| \log |V|)$ (which can be improved using more sophisticated PQ implementations)

- If we only need simple queue, we get a linear cost $O(|E|)$

- If needed, we add a driver to restart `BFS` at unseen nodes

Depth First Search (DFS)

- The alternative to BFS is recursive DFS

```
def DFS(u, G):
    s[u] = True
    do_something_before_DFS(u)
    for all w adjacent to u:
        if s[w] == False:
            p[w] = u
            DFS(w, G)
    do_something_after_DFS(u)
```

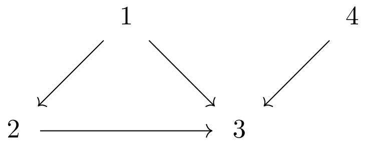
- The table `p[ ]` defines the DFS tree (or forest)

Depth First Search II

- We may have to restart DFS if not all nodes have been processed, for which we need a driver for DFS

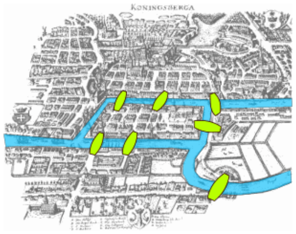
```
def driver_DFS(G):
    s[ ] = False; p[ ] = NULL
    for all u in V:
        if s[u] == False:
            DFS(u,G)
```

- If doing something has cost $O(1)$, the joint cost of `driver_DFS` and `DFS` is clearly $O(|E|)$
- An example:



Applying DFS

- The DFS evolution is



Edge Classification by DFS

- DFS induces a classification on the edges of G
  - **Tree edges**: (u, v) where $u = p[v]$
  - **Back (ascending) edges**: (u, v) where $v = p[\ldots p[u] \ldots]$ (one or more p)
  - **Forward (descending) edges**: (u, v) where $u = p[\ldots p[v] \ldots]$ (with at least 2 p)

  - **Cross edges**: any other $(u, v) \in E$

- If G is undirected and (u, v) is a forward edge, then (v, u) is a back edge
  - Thus, we will not distinguish then between forward and back edges
- We prove next that if G is undirected there are no cross edges

Parenthesis Theorem

- Assume we have a counter c in DFS and consider 2 time–stamps:
  - **Discovery**: $d_u = c; c+ = 1$, updated when DFS **starts** on u
  - **Finish**: $f_u = c; c+ = 1$, updated when DFS **ends** on u
- Obviously $d_u < f_u$
- **Parenthesis Theorem.** *For a graph G and $u, v \in V$, consider the intevals $I_u = (d_u, f_u)$, $I_v = (d_v, f_v)$. Assuming $d_u < d_v$ we either have $I_v \subset I_u$, or $I_u \cap I_v = \emptyset$*
- **Proof sketch:** Assume $d_u < d_v$;
  - If $f_u < d_v$, obviously $I_u \cap I_v = \emptyset$
  - And if $f_u > d_v$, DFS recursively started on v before finishing with u; thus the recursion on v must finish before that of u and $f_v < f_u$
  - Thus, $I_v \subset I_u$

No Cross Edges in Undirected Graphs

- **Corollary.** *If G is undirected there are no cross edges*
- **Proof sketch:** Take $(u, v) \in E$:
  - Assume $d_u < d_v$; then we have $f_v < f_u$ for v is adjacent to u
  - If $s[v] = F$ when we arrive at v, then (u, v) is a tree edge
  - And if $s[v] = T$ when we arrive at v, we have processed $L[v] \Rightarrow$ we have processed (v, u), that must be a back edge
  - Thus, (u, v) is a forward edge
- Thus, in no case is (u, v) a cross edge

5.2 Biconnected Graphs

Undirected Graph Connectivity

- Recall that an undirected graph $G = (V, E)$ is connected if for every pair $u, v \in V$ there is a path π in E from u to v
- **Connected component**: a maximal connected subgraph of G
- If $G_i = (V_i, E_i)$ are the connected components of G, the V_i are a **partition** of V and the E_i of E

- If we order the vertices of G as $V = V_1 \cup \ldots \cup V_K$, then the adjacency matrix M is **block diagonal** with the blocks M_k being the adjacency matrices of the G_k
- BFS can be used to give the connected components of G through the table $p[\,]$ just counting how many nodes u verify $s[u] = True$ and restarting BFS if it is $< |V|$
- DFS and its driver can also be used to give the connected components of G through the table $p[\,]$

An Aside: Directed Graph Connectivity

- A directed graph $G = (V, E)$ is **weakly connected** if its extension to an undirected graph is connected
- A directed graph $G = (V, E)$ is **strongly connected** if for every pair $u, v \in V$ there is a path π in E from u to v
- DFS is also used in **Tarjan's Algorithm** to obtain the strong components of a graph
- Tarjan's algorithm basically obtains the strong components computing DFS's ending times on G and applying again DFS to the transpose graph G^τ in the order inverse to the ending times

Articulation Points

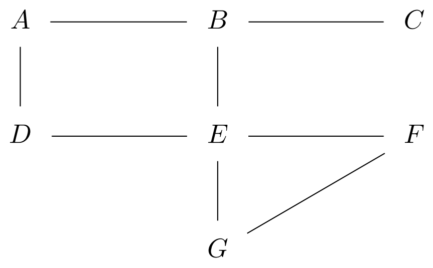
- If G is undirected and connected, a **cut vertex** or **articulation point (AP)** is a vertex u such that

  $$G' = (V - \{u\}, E - \{(u, z) \in E\})$$

  is no longer connected
- An undirected and connected graph G is **biconnected** if it has no articulation points
- Biconnected graphs are desirable in computer networks, as they are more robust against router failures
- Q: how we detect APs?

How to Detect APs?

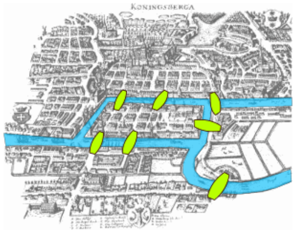
- An example: the graph below has two APs, B and E



- We next apply DFS.

DFS To Detect APs

- We show DFS evolution of the adjacency list and the edges on the DFS tree:



DFS To Detect APs II

- From this “top-down” view of the graph we can more easily detect APs:
  - A is not AP: it does not unhook any vertex
  - B is AP: it unhooks C
  - C is not AP: it has no children
  - E is AP: it unhooks F and G
  - D is not AP: it has no children
  - F is not AP: G can reach E without F
  - G is not AP: it has no children it does not unhook any vertex

DFS and Articulation Points

- The example shows that the DFS tree gives a “top–bottom” view of a graph in which
  - APs other than the root disconnect lower parts of the graph
  - An AP at the root disconnects subtrees
- We can use two auxiliary tables to detect articulation points that can be computed by DFS
  - The **order** table $o[\,]$ that contains the order in which DFS arrives at a node u.
  - The **ascent** table $a[\,]$ which is defined as $a[u] = \min\{o[v]\}$ where v is any node that can be accessed from u by
    - Going “down” through 0, 1 o more tree edges, and then
    - Going“up” through a single back edge

The DFS Auxiliary Tables

- The o, o', a', a tables for the previous example are

	o	o'	a'	a
A	1	∞	1	1
B	2	∞	1	1
C	3	∞	∞	3
D	5	1	∞	1
E	4	∞	1	1
F	6	∞	4	4
G	7	4	∞	4

Detecting Articulation Points

- Clearly if we remove a non root node u from the DFS tree, it will disconnect one of its children v unless v can go "above" $o[u]$ using back edges,
- In other words, u **will be an AP if for some child** v **we have** $o[u] \leq a[v]$
  - Notice that a larger number means a "lower" node
- Since there are no cross edges on the DFS tree, **the root node will be an AP if it has two or more children**
- It is also clear that these sufficient conditions are also neccessary
  - A single root node cannot be an AP
  - If all children of u bypass it, u cannot be an AP

Computing $o[\,]$ and $a[\,]$

- We compute $o[u]$ **before** DFS explores u's adjacency list
- We can use two auxiliary tables to compute the table $a[\,]$
- The **direct ascent** table $o'[u]$ that contains the order of highest node accessible from u by an ascending edge

$$o'[u] = \min\{o[v] : (v, u) \text{ is a back edge}\}$$

  $o'[u]$ can be computed **before** DFS looking at the w adjacent to u s.t. $s[w] == True$

- The **ascent by children** table $a'[u]$ that contains the order of highest node accessible from any of the children of u

$$a'[u] = \min\{a[v] : u = p[v]\}$$

  $a'[u]$ can be computed **after** the recursive call to DFS returns

- We then have $a[u] = \min\{o[u], o'[u], a'[u]\}$

Computing $o[\,]$ and $a[\,]$ II

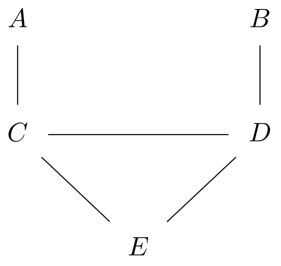
- Assume the DFS driver has initialized $o[\,]$ and $a[\,]$ to ∞ and a counter c to 0
- We compute $o[\,]$ and $a[\,]$ recursively as follows

```
def ap_tables(u, G):
    s[u] = True; o[u] = c; a[u] = o[u]; c += 1
    for all w adjacent to u: # direct ascent
        if s[w] == True and w != p[u] and o[w] < a[u]:
            a[u] = o[w]
    for all w adjacent to u:
        if s[w] == False:
            p[w] = u; ap_tables(w, G)
    for all w adjacent to u: # ascent by children
        if p[w] == u and a[u] > a[w]:
            a[u] = a[w]
```

- The cost of `ap_tables` is clearly $O(|E|)$

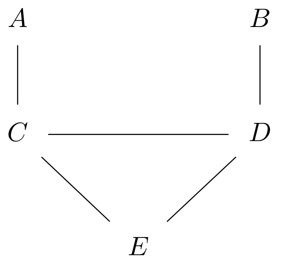
Algorithm Application

- A second example:



Algorithm Application II

- We compute o, o' before DFS and a', a after DFS



Analyzing the Tables

- A is not AP: it has no children

- B is not AP: it has no children
- C is AP: root with 2 children
- D is AP: $a[B]4 \geq 3 = o[D]$
- E is not AP: it has no children

5.3 DAGs and Topological Sort

Directed Acyclic Graphs

- A **directed acyclic graph** (DAG) is a directed graph without cycles
- **Proposition:** G is a DAG iff there are no ascending edges in G
  - If (v, u) is ascending, there is a path from u to v in the DFS forest, and adding (v, u) results in a cycle
  - Assume π is a cycle and let $u \in V_\pi$ be the first node processed in DFS and assume (v, u) in E_π

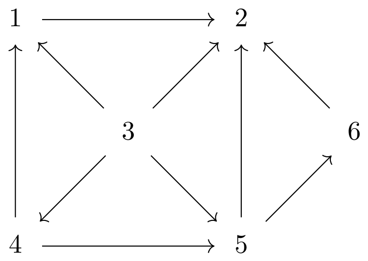
    Then it can be shown that v descends from u in the DFS forest and, thus, (v, u) is ascending
- DFS can be used to detect cycles in a graph modifying our previous AP algorithm
- DAGs can be used to model many problems of interest

Topological Sort

- Recall: $\leq$ is a **total order** if either $u \leq v$ or $v \leq u$ or both
- A **topological sort** in a DAG $G = (V, E)$ is a total ordering of its vertices s.t. if $(u, v) \in E$, then $u \leq v$
- If G is a DAG, a topological sort can be obtained
  - Applying DFS starting at u with $inc[u] = 0$ (there is always one) and
  - Adding a vertex u at the beginning of a linked list after DFS ends its process
- We end up with a topological sort of G:
  - Since DFS ended at v **after** processing of the vertices w adjacent to v, then these w are in the list **after** v
- The cost of TS on DAGs is thus $O(|E|)$

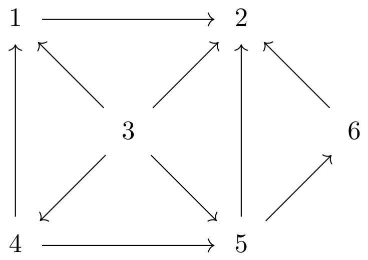
Applying Topological Sort

- An example:



Applying Topological Sort II

- We apply DFS computing the discovery and finish times



- TS can also be obtained by reversed finish times