

# Design and Analysis of Algorithms

José Dorronsoro  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid

## Contents

<b>1</b>	<b>Elementary Graph Algorithms</b>	<b>3</b>
1.1	Basic Concepts on Graphs	3
1.2	Minimum Distances on Graphs	4
1.3	All Pairs Shortest Paths	8
<b>2</b>	<b>Eulerian and Hamiltonian Circuits</b>	<b>10</b>
2.1	Eulerian Circuits	10
2.2	Hamiltonian Circuits and an Excursion on Complexity Theory	15
2.3	The Traveling Salesman Problem	18
<b>3</b>	<b>An Excursion on DNA Sequencing</b>	<b>21</b>
3.1	Hamilton, Euler and DNA Sequencing	21
3.2	From Reads to the Genome	25
<b>4</b>	<b>Depth First Search and Connectivity</b>	<b>28</b>
4.1	Depth First Search	28
4.2	Biconnected Graphs	30
4.3	DAGs and Topological Sort	33
<b>5</b>	<b>Minimum Spanning Trees</b>	<b>34</b>
5.1	The Algorithms of Prim and Kruskal	34
5.2	The Disjoint Set Abstract Data Type	36
5.3	Correctness of Prim and Kruskal	42
<b>6</b>	<b>Maximum Flows on Graphs</b>	<b>44</b>
6.1	Basic Definitions and Facts	44
6.2	Maximal Flows and Minimal Cuts	49

# 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  $w_{uv}$

### Storing an Unweighted Graph

- **Adjacency matrix:** Assume  $V = \{1, \dots, N\}$ . Then if  $(i, j) \in E$ ,  $m_{ij} = 1$ ; else,  $m_{ij} = 0$ 
  - 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:  $\Theta(|V|) + \Theta(|E|)$
- In all cases the cost is  $O(|V|^2)$  since for standard graphs

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

### 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  $G$  is 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

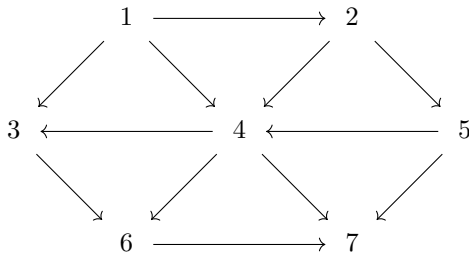
## 1.2 Minimum Distances on Graphs

### Minimum Distance Problems

- **Path** from  $u$  to  $v$ : a subset  $\pi = \{u = u_0, \dots, u_K = v\}$  with  $(u_i, u_{i+1}) \in E$
- **Length** of  $\pi$ :  $|\pi| = K = \#(\text{number of})$  edges
- First problem: given  $u$ , find a **shortest path** (i.e., a path with the smallest number of edges)  $\pi$  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 length strings, and make  $G$  “hang” from  $u$  discarding “repeated” edges
- **Example:**



- Once obtained, the tree layers give vertices at distances 1, 2, 3 and so on

### 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  $(w = p[v], v)$ ,  $(p[w], w)$ , 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[ ] = NULL; 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[ ] = NULL
    Q = q()
    doSomething(u); 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
```

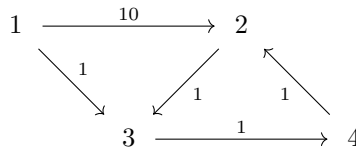
- Since we enter each list only once, if the cost of `doSomething` is  $O(1)$ , the cost of BFS is  $O(|E|)$ , i.e., linear,
- If needed, we add a driver to restart `BFS` at unseen nodes

### Minimum Distances on Weighted Graphs

- $G = (V, E)$  is a **weighted** graph if there is a function  $c : E \rightarrow R$ 
  - 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  $\pi$ :  $c(\pi) = c(\{u_0, \dots, u_K\}) = \sum_1^K c(u_{j-1}, u_j)$
- 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 ...

- Observation: working with the tree like representation of  $G$  is now trickier
- **Example:**



- 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**  $Q$ 
  - We also need (again) a table  $s[v]$  to check whether  $v$  has left  $Q$  and 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[ ] = NULL; 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

	d	p	v	d	p	v	d	p	v
1	0	-	F	0	-	T	0	-	T
2	$\infty$	-	F	10	1	F	10	1	F
3	$\infty$	-	F	1	1	F	1	1	T
4	$\infty$	-	F	$\infty$	-	-	2	3	F
PQ	1 <sub>0</sub>			3 <sub>1</sub> , 2 <sub>10</sub>			4 <sub>2</sub> , 2 <sub>10</sub>		

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

### 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[ ] = NULL; 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]
                if d[z] smaller afterwards:
                    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  $V = \{1, \dots, N\}$  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**  $\{1, \dots, k\}$
- 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  $\{1, \dots, k\}$  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[:, :, 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  $\pm$  obviously correct

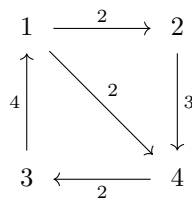
### Floyd–Warshall Cost

- The time cost is  $O(N^3)$ , better than iterated Dijkstra for dense graphs
- The space cost is at 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 **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:

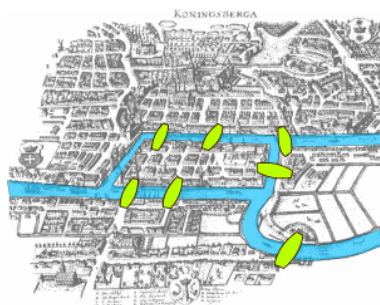


## 2 Eulerian and Hamiltonian Circuits

### 2.1 Eulerian Circuits

#### The Bridges of Königsberg

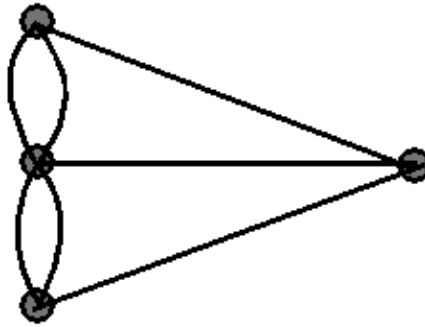
- The bridges of Königsberg (East Prussia) over the Pregel river circa 1700:



- 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

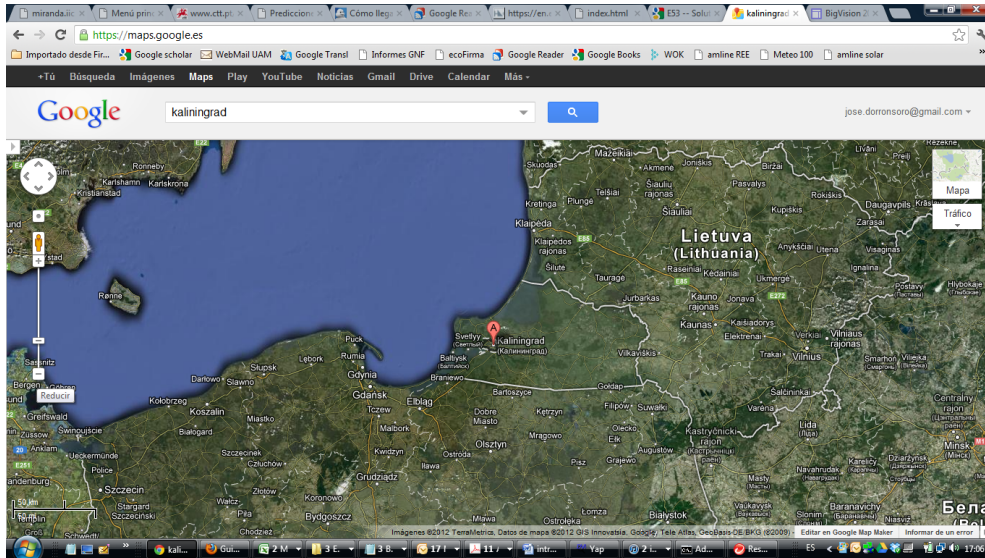
- Leonhard Euler showed in 1736 (*Solutio problematis ad geometriam situs pertinentis*) that such a circuit is not possible
- If  $G$  is an undirected graph, we define the **degree**  $\deg(w)$  of a node  $w$  as the number of edges that leave  $w$  (or that enter  $w$  or simply the size of  $T[w]$ )
- 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  $\pi$ , each time we enter  $w$  we subtract 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  $\pi$  we subtract 1 from  $\deg(u)$  and also when we leave  $u$ ; moreover, when we start and end  $\pi$  we also subtract 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  $G$

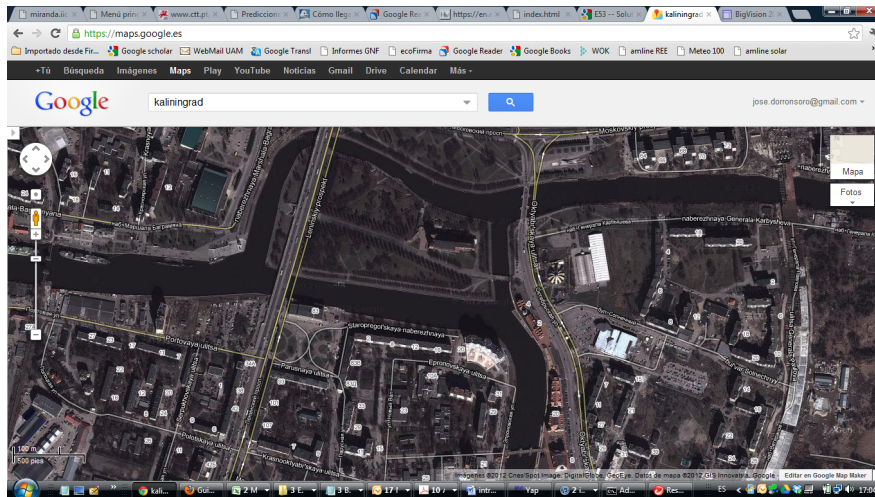
### The Bridges of Kaliningrad

- After World War II Königsberg becomes Kaliningrad (Russia)



### The Bridges of Kaliningrad II

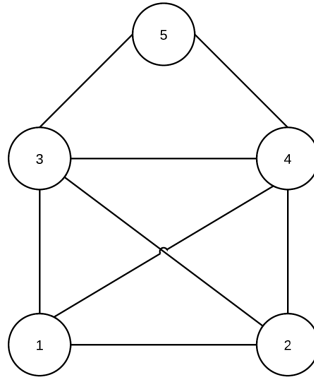
- And there only remain five bridges of the initial seven (and some more modern ones)



- But still, there is no EC

### 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



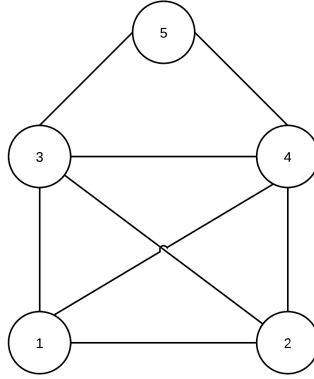
- It is very easy if we start at nodes 1 or 2 but impossible if we start from 3, 4 or 5

### Euler's Insight Again

- Now we want to find an **Eulerian path** (EP) and not a circuit
- Let  $\pi = \{(u = u_0, u_1), \dots, (u_{K-1}, u_K = v \neq u)\}$  be such an EP
- If  $w \neq u, v$  is a node in  $\pi$ , each time we enter  $w$  we subtract 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  $\pi$  we subtract 1 from  $\deg(u)$  and also when we leave  $u$ ; moreover, since we start  $\pi$  at  $u$ , we also subtract 1 from  $\deg(u)$ 
  - Thus, we must also have  $\deg(u)$  odd
- Similarly each time we enter  $v$  inside  $\pi$  we subtract 1 from  $\deg(v)$  and also when we leave  $v$ ; moreover, since we end  $\pi$  at  $v$ , we also subtract 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  $\pi$**

### 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

### 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  $u$  subtracting 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  $\pi$
- We remove  $E_\pi$  from  $E$  and from  $V$  the nodes  $w$  whose  $\deg \pi$  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  $G_1, \dots, G_K$  of  $G'$
- By induction there are ECs  $\pi_k$  in the  $G_k$  that start at nodes from  $\pi$  and we get an EC on  $G$  “collating”  $\pi$  and the  $\pi_k$

### Euler's Theorem for Paths

- **Corollary.** If  $G$  is a connected undirected graph, there is an EP  $\pi$  in  $G$  iff  $\deg(w)$  is even for all  $w \in V$  except for two vertices  $u$  and  $v$ . Moreover, then  $\pi$  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  $\pi'$  in  $G'$
- Let's write  $\pi'$  as  $\pi' = \{(v, z), \dots, (w, u), (u, v)\}$ , with the last edge the one we added to get  $G'$ .

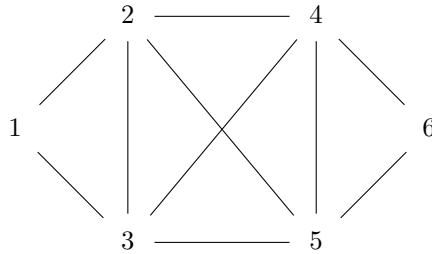
- Then removing this edge we get the EP  $\pi = \{(v, z), \dots, (w, u)\}$ .

### 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  $\pi_1 = \{(u_1, v_2), \dots, (v_{K-1}, v_K)\}$  subtracting 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  $\pi_1$  from  $E$  and all the  $w \in V$  for which  $\deg(w) = 0$  after  $\pi$ , 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  $\pi_1$  is an EC in  $G$
  - If however  $|V_1| > 0$ , there is a first  $u_2$  in  $\pi_1$  such that  $\deg_{G_1}(u_2) > 0$
  - We can thus **restart the above process on  $G_1$**  obtaining a new circuit  $\pi_2$  and a “remaining” graph  $G_2$
- If we repeat the preceeding and find circuits  $\pi_1, \dots, \pi_M$  until  $V_M = \emptyset$ , then we can “collate” the  $\pi_j$  circuits to get an EC  $\pi$  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  $O(|E|)$

## 2.2 Hamiltonian Circuits and an Excursion on Complexity Theory

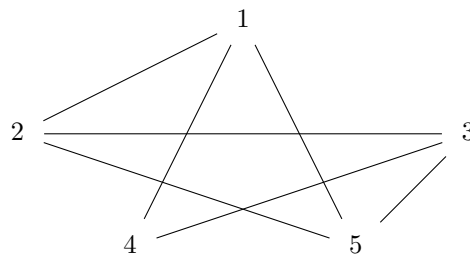
### Hamiltonian Circuits

- 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

### 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](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

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

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  

$$(x_{11} \text{ OR } !x_{12} \text{ OR } x_{13}) \text{ AND } (!x_{21} \text{ OR } x_{22} \text{ OR } !x_{23}) \text{ AND } \\ (x_{31} \text{ OR } !x_{32} \text{ OR } x_{33}) \text{ AND } \dots$$
- **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?

## 2.3 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  $\pi$  such that  $c(\pi) \leq C$ ?
- TSP is **NP-hard**: every problem in NP can be polynomially reduced to TSP
  - A NP-hard problem may not have to be in NP 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  $\text{tsp}(V, c)$  is a routine that returns the TSP solution for  $G$  with cost  $c$  and consider the following routine for HC:

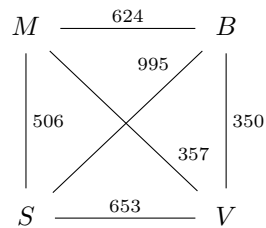
```
def tsp_2_hc(V, E):
    for any u, v in V:
        if (u, v) in E:
            c(u, v) = 1
        else:
            c(u, v) = 2

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

- $\text{tsp\_2\_hc}$  solves HC for  $\pi$  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



- And distances given by a  $4 \times 4$  Numpy matrix

### 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  $\pi_{NN}$  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*

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

*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  $T^*$

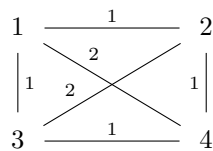
- Since  $T_1$  is an MST, we have  $c(T_1) \leq c(T^*) \leq c(\pi^*)$
- And using the Euclidean distance property, we then we conclude that

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

- 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



## 3 An Excursion on DNA Sequencing

### 3.1 Hamilton, Euler and DNA Sequencing

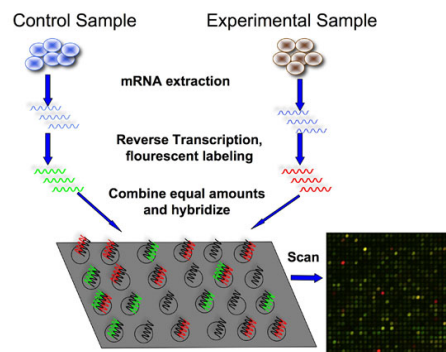
#### DNA Sequencing

- **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](http://bitesizebio.com/7206/introduction-to-dna-microarrays)

### Microarray Hybridization

- Put all the possible length  $\ell$  probes, i.e., DNA subsequences of a fixed length  $\ell$ , 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  $\ell$  of the fragment
- This way we get all possible length  $\ell$  subsequences that make the fragment but they are **unordered**

### $\ell$ -mers and the Spectrum

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

$$sp(S, 3) = \{\text{TAT}, \text{ATG}, \text{TGG}, \text{GGT}, \text{GTG}, \text{TGC}\}$$
- 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  $\ell$ -mers  $s_1, s_2$  is the longest length 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  $s_{i_1}, \dots, s_{i_K}$  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  $S = \text{[TATGGTGC]}$  and the unordered 3-spectrum  

$$sp(S, 3) = \{\text{GGT}, \text{TAT}, \text{TGG}, \text{TGC}, \text{ATG}, \text{GTG}\}$$

### 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  $\ell$ -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  $\pi = \{(u = u_0, u_1), \dots, (u_{K-1}, u_K = u)\}$  is an Eulerian circuit on  $G$
- If  $w \neq u$  is a node in  $\pi$ , each time we enter  $w$  we subtract 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  $\pi$  we subtract 1 from  $in(u)$  and also from  $out(u)$  when we leave it

Moreover, when we start we subtract 1 from  $out(u)$  and also subtract 1 we subtract 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 necessary and sufficient condition to have an EC in a directed  $G$  is that  $in(v) = out(v)$  for all  $v \in V$



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

### Eulerian Sequencing

- **Example:** consider again  $S = \text{[TATGGTGC]}$  and  
 $\text{sp}(S, 2) = \{\text{TA}, \text{AT}, \text{TG}, \text{GG}, \text{GT}, \text{GC}\}$

## 3.2 From Reads to the Genome

### The Shortest Superstring Problem (SSP)

- Given a string set  $\{s_1, \dots, s_M\}$ , find the **shortest superstring**  $S$  that contains the  $M$  substrings  $s_j$
- Recall that the **overlap**  $\omega_{i,j}$  between strings  $s_i, s_j$  is the length of the longest suffix of  $s_i$  that is also a prefix of  $s_j$
- Notice that the length of the shortest string containing first  $s_i$  and then  $s_j$  is  $|s_i| + |s_j| - \omega_{i,j}$  where  $|s|$  denotes the length of  $s$
- If we add another string  $s_k$  after  $s_j$ , the extra added length is  $|s_k| - \omega_{j,k}$
- Thus, if we get  $S'$  collating the ordering  $\{s_{i_1}, \dots, s_{i_M}\}$  then

$$\begin{aligned} |S'| &= |s_{i_1}| + |s_{i_2}| - \omega_{i_1, i_2} + |s_{i_3}| - \omega_{i_2, i_3} + \dots \\ &= \sum_{j=1}^M |s_j| - \sum_{j=1}^{M-1} \omega_{i_j, i_{j+1}} \end{aligned}$$

### The Longest Path Problem (LPP)

- As an example, if we collate this way the strings  
 $\text{'ATGGTAG'}$ ,  $\text{'GTAGACTA'}$ ,  $\text{'CTAGGTATT'}$   
 we get the sequence  $\text{'ATGGTAGACTAGGTATT'}$  of length  $17 = 7 + 8 + 9 - 4 - 3$
- Clearly  $|S'|$  will be minimal iff  $\sum_{i=1}^{M-1} \omega_{i_j, i_{j+1}}$  is maximal
- Consider the complete graph  $G$  over  $V = \{s_1, \dots, s_M\}$  and with cost  $\omega_{i,j}$
- Solving the Shortest Superstring Problem is thus equivalent to solving the **Longest Path Problem (LPP)** in  $(G, \omega)$ : to find a cycle-free path of maximal length

### LPP and Hamiltonian Paths

- A Hamiltonian path (HP) is a path that passes over all the  $|V|$  vertices once
- It can be shown that HC reduces polynomially to HP (the argument is easy but not totally obvious)
- Consider LPP-d, the decision version of LPP: given a graph  $G$  and  $c$ , to decide whether there is an acyclic path in  $G$  with length  $\geq c$
- It is easy to see that LPP-d is in NP
- It is also easy to see that HP reduces to LPP-d: there is a HP in  $G$  iff LPP-d over  $(G, |V| - 1)$  returns 1
- As a consequence, if HC is NP-complete, so is HP and so is LPP-d
- In general, if  $P_1$  is NP-complete and it reduces polynomially to  $P_2$ ,  $P_2$  is also NP-complete

### How to Create New NP-complete Problems

- Assume  $T$  is the reduction operator from  $P_1$  to  $P_2$  with cost  $O(|I|^p)$  when applied to an input  $I$  of  $P_1$
- If  $P'$  is another problem that reduces polynomially to  $P_1$  via  $T'$ ,  $T \circ T'$  **reduces  $P'$  to  $P_2$  with polynomial cost**
- In fact, if  $I = T'(I')$  with  $I'$  an input of  $P'$  with size  $n$  and  $T'$  has cost  $O(n^q)$ , we have  $|I| = O(n^q)$ , for the size of the outputs of  $T'$  cannot be larger than the cost of computing them
- But the cost of applying  $T \circ T'$  to  $I'$  is  $O(n^q + (n^q)^p) = O(n^{pq})$
- Thus, if a NP problem reduces polynomially to  $P_1$ , so it does to  $P_2$
- Thus, if  $P_1$  is NP-complete, so is  $P_2$
- Thus, if we manage to prove HC to be NP-complete, so is HP and so is LPP-d. And so is TSP-d

### Another Look at SSP

- Alternatively, assume we decompose two sequences  $s$  and  $t$  as

$$s = \text{pre}_{st} + \text{over}_{st}; \quad t = \text{over}_{st} + \text{suf}_{st}$$

with  $\text{pre}$ ,  $\text{over}$  and  $\text{suf}$  the prefix, overlap and suffix of  $s$  and  $t$  and  $+$  the concatenation operator

- For instance, for  $s = \text{ATGGTAG}$ ,  $t = \text{GTAGACTA}$ , we have  $\text{tpre}_s = \text{ATG}$ ,  $\text{tover}_s = \text{GTAG}$ ,  $\text{tsuf}_s = \text{ACTA}$
- Then the shortest superstring  $s \circ t$  of  $s$  and  $t$  is

$$\text{pre}_{st} + \text{over}_{st} + \text{suf}_{st} = s + \text{suf}_{st} = \text{pre}_{st} + t$$

- Now, concatenating  $s_i, s_j, s_k$  to get  $s_{ijk}$ , we have  $s_{ijk} = s_i \circ (s_j \circ s_k) = s_i \circ s_{jk}$ , i.e.,

$$s_{ijk} = s_i \circ s_{jk} = \text{pre}_{ij} + s_{jk} = \text{pre}_{ij} + \text{pre}_{jk} + s_k$$

- But  $s_k = \text{pre}_{k\ell} + \text{over}_{k\ell}$  for any other  $s_\ell$

### SSP and TSP

- Thus, choosing  $\ell = i$ ,  $s_{ijk} = s_i \circ (s_j \circ s_k) = \text{pre}_{ij} + \text{pre}_{jk} + \text{pre}_{ki} + \text{over}_{ki}$
- Therefore, setting  $\rho_{i,j} = |\text{pre}_{ij}|$  and  $\omega_{i,j} = |\text{over}_{ij}|$  we have

$$|s_{ijk}| = \rho_{i,j} + \rho_{j,k} + \rho_{k,i} + \omega_{k,i}$$

- Therefore, collating  $\{s_{i_1}, \dots, s_{i_M}\}$  to get a possible SS  $S'$ , we have

$$|S'| = \sum_1^{M-1} \rho_{i_j, i_{j+1}} + \rho_{i_M, i_1} + \omega_{i_M, i_1}$$

- Consider the complete graph  $G$  defined by the  $M$  sequences  $s_k$  with cost  $\rho_{ij}$
- Then, if  $\rho^*$  is the optimal cost of TSP on  $G$  we have

$$\begin{aligned} |S'| &= \sum_1^{M-1} \rho_{i_j, i_{j+1}} + \rho_{i_M, i_1} + \omega_{i_M, i_1} \geq \rho^* + \omega_{i_M, i_1} \\ &\geq \rho^* + \min_j \{\omega_{i_j, i_{j+1}}\} \geq \rho^* \end{aligned}$$

### Solving SSP and LPP

- Thus, the solution of TSP on the complete graph over the  $s_j$  with cost  $c(i, j) = \rho_{i,j}$  gives a lower bound on the length of the shortest superstring
- The relationship between SPP and TSP hints that SPP-d is also NP-complete (and it is so)
- Both SSP and LPP are NP-hard
- There are approximation algorithms for SSP with a 2.5 bound
- But LPP is much harder to approximate

- For directed acyclic graphs (DAGs) LPP can be solved in linear time using the Topological Order algorithm

### Back to Genome Sequencing

- Recall the four steps of the process:
  - Blast the gene into individual short fragments (“reads”)
  - Identify read subsequences by hybridizing them on a DNA microarray
  - Reconstruct each read from these subsequences
  - Reconstruct the gene from the reads
- Biochemistry provides the first two and Computer Science (CS) can be applied to the last two steps and will give a plausible genome
- But: is it the one we started with?
- Not obvious in principle: much more Biochemistry needed!!

## 4 Depth First Search and Connectivity

### 4.1 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

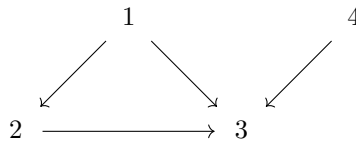
- We may have to restart DFS if not all nodes have been processed
- We need a driver for DFS

```

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

```

- An example:



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

### 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  $v = p[\dots p[u] \dots]$  (one or more  $p$ )
  - **Forward (descending) edges:**  $(u, v)$  where  $u = p[\dots p[v] \dots]$  (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 intervals  $I_u = (d_u, f_u)$ ,  $I_v = (d_v, f_v)$ . 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

## 4.2 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  $\pi$  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  $V = V_1 \cup \dots \cup V_K$ , 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  $s[u] = \text{True}$  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  $\pi$  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  $G^T$  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

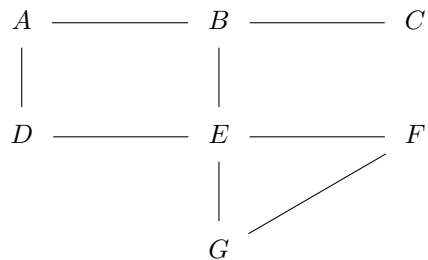
$$G' = (V - \{u\}, E - \{(u, z) \in E\})$$

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  $E$



- If we apply DFS ...

### 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 or more tree edges, and then
  - \* Going “up” through a single back edge

### 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 necessary
  - 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  $\infty$  and a counter  $c$  to 0
- We compute  $o[]$  and  $a[]$  recursively as follows



```

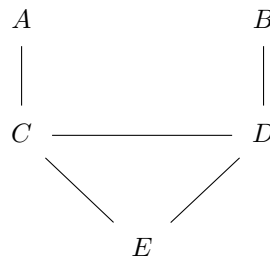
def ap_tables(u, G):
    s[u] = True; o[u] = c; a[u] = o[u]; c += 1
    for all w adjacent to u: # direct ascent
        if s[w] == True and w != p[u] and o[w] < a[u]:
            a[u] = o[w]
    for all w adjacent to u:
        if s[w] == False:
            p[w] = u; ap_tables(w, G)
    for all w adjacent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

- An example:



## 4.3 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  $\pi$  is a cycle and let  $u \in V_\pi$  be the first node processed in DFS and assume  $(v, u)$  in  $E_\pi$   
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

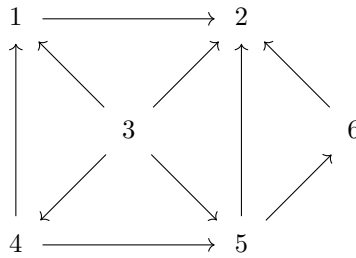
### Topological Sort

- 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:



## 5 Minimum Spanning Trees

### 5.1 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  $\pi$  in  $E$  from  $u$  to  $v$
- A **cycle**  $\pi$  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  $c(T) \leq c(T')$

### Prim's Algorithm

- Changing slightly Dijkstra'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

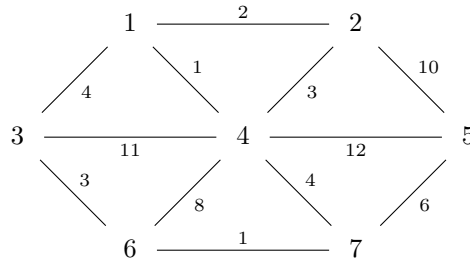
- There may be several minimum spanning trees in a graph
- We have  $c(T) = \sum_{v \neq u} c(p[v], v)$
- The cost of Prim is  $O(|E| \log |V|)$  if the PQ is built over a min heap
- **Prim works:** at the end  $p[v]$  gives the edges  $(p[v], v)$  of a MST  $E_T$  and  $cE[v]$  its costs
  - But again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



### 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 store the edges on a linked list (or simply build the graph of the final MST)
- 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**
  - Thus we do not need to work with trees but with **subsets**
- We do this with a new abstract data type, the **Disjoint Set**

## 5.2 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 family 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)                  # 5
        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 First Data Structure for DS

- We assume  $V = \{1, \dots, N\}$
- A simple idea is keep each subset in a list with the representative in the first node
- We also construct a pointer (dict?) table  $T[\ ]$  where  $T[i]$  points to the list that contains  $i$

- The cost of `find` is clearly  $O(1)$
- To implement `union(x, y, S)` we can concatenate the list  $T[y]$  after the list  $T[x]$  and then make sure that for each  $j$  in  $T[y]$  we have  $T[j] = T[x]$
- However this is not satisfactory as the cost of the union is then
  - $|T[x]|$  (to find the end point) plus
  - $|T[y]|$  (to reset the pointers of  $V_{T(y)}$ )

### Improving the First Data Structure

- We can improve things if we keep track of the last node of each  $T[i]$  and also on the sizes  $|T[x]|$ .
- This way we find the last element of  $T[x]$  with cost  $O(1)$  and then join the smaller subset to the largest one
- Working with pure linked lists, notice that we have to keep track of the size and the last element only at the first node of  $T[x]$ 
  - The extra cost is then  $O(1)$
- The cost of `union(x, y, S)` is then  $\min(|T[x]|, |T[y]|)$  and if done well, it is always  $O(\log N)$
- But we need to work with 3-field nodes in the lists, which in most nodes just keep outdated information
- We try to do something better

### A Second 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  $O(\text{height}(T_x))$  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  $\text{prof}(T)$  denotes the depth of a DS tree  $T$ , we have  $\text{prof}(T) \leq \lg |T|$

- **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 all  $v$  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
```

### 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

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

- 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

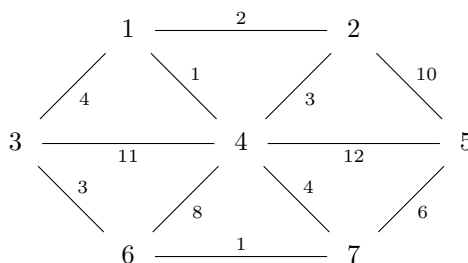
    return l_mst
```

### 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  $x \neq y$ , 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:



### 5.3 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. If  $(w, z)$  also crosses  $P$ , it 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

**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:**
  - 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 we have to check it connected
  - Assume  $x, y$  at different sides of  $P$ ; since  $T$  is connected, there is a path  $\pi$  from  $x$  to  $y$  in  $E_T$  and it crosses  $P$  at  $(w, z)$  (if two edges of  $T$  cross  $P$ ,  $T$  would be a cycle)
  - We can decompose  $\pi$  as  $\pi = \pi_{x,w} \cup \{(w, z)\} \cup \pi_{z,y}$
  - Then  $\pi' = \pi_{x,w} \cup \pi_{w,u} \cup \{(u, v)\} \cup \pi_{v,z} \cup \pi_{z,y}$  is a path on  $E_{T'}$

**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  $i$ ,  $i = p + 1, \dots, u$ , the subtable  $T[p], \dots, T[i]$  is sorted
  - BubbleSort: after iteration  $i$ ,  $i = u, \dots, p + 1$ , the subtable  $T[i], \dots, T[u]$  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  $L_0 = \emptyset \subset L_1 \subset \dots \subset L_{N-1}$  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*

## 6 Maximum Flows on Graphs

### 6.1 Basic Definitions and Facts

#### Flow Networks

- A **network flow** is a directed graph  $G = (V, E)$  such that
  - There is a **source**, a single node  $F$  such that  $in[F] = 0$
  - There is a **sink**, a single node  $S$  such that  $out[S] = 0$
  - There is a nonnegative **capacity** function  $c : E \rightarrow \mathbb{R}$
- We extend  $c$  to  $V \times V$  as an **augmented capacity** with  $c(w, z) = 0$  if  $(w, z) \notin E$
- **Example:**

#### Flows in Networks

- A **flow** in the flow network  $G$  is a real function on  $V \times V$  such that
  - For all  $w, z$ ,  $f(w, z) \leq c(w, z)$  (**boundedness**)
  - For all  $w, z$ ,  $f(w, z) = -f(z, w)$  (**antisymmetry**)

- For all  $w$  other than  $F, S$ ,  $\sum_z f(w, z) = 0$  (**conservation**)

- **Example:**

- If  $(u, v) \notin E$  and  $(v, u) \notin E$ , then  $f(u, v) = f(v, u) = 0$ , for

$$0 = -c(u, v) \leq -f(u, v) = f(v, u) \leq c(v, u) = 0;$$

thus, we can ignore these edges

### Maximum Flow Problem

- For  $u \neq F, S$  set  $V_u^+ = \{v : f(u, v) > 0\}$ ,  $V_u^- = \{v : f(u, v) < 0\}$ ; then, by conservation and antisymmetry

$$0 = \sum_{V_u^+} f(u, v) + \sum_{V_u^-} f(u, v) = \sum_{V_u^+} f(u, v) - \sum_{V_u^-} f(v, u);$$

- Thus, for any node the incoming flow equals the outgoing flow, i.e., no flow is “lost” in  $u$
- The **value** of a flow  $f$  is  $|f| = \sum_{u \neq F} f(F, u)$
- By conservation  $|f| = \sum_{v \neq S} f(v, S)$
- In the **maximum flow** problem we want to find a flow in  $G$  with  $|f|$  maximum

### Residual Capacity

- If  $f$  is a flow in the flow network  $G, c$ , we define  $f$ ’s **residual capacity** on  $V \times V$  as  $c_f(u, v) = c(u, v) - f(u, v)$
- The **residual network** of  $f$  is the flow network  $G_f = (V, E_f)$  where  $(u, v) \in E_f$  iff  $c_f(u, v) > 0$
- We can see  $G_f$  as the manoeuvres we can do on  $G$  to augment  $|f|$
- **Example:**

### Augmenting Path

- A path  $\pi$  from  $F$  to  $S$  on  $G_f$  is an **augmenting path** (AP)
- Notice that if  $(u, v) \in \pi$ , then  $c_f(u, v) > 0$

- The **augment**  $a_\pi$  of  $\pi$  is  $a_\pi = \min\{c_f(u, v) : (u, v) \in \pi\}$
- We can exploit such a  $\pi$  to augment  $|f|$
- **Example:**

### Augmented Flow

- If  $\pi$  is augmenting for  $f$ , we define the **augmented** flow  $f'$  as

$$\begin{aligned} f'(u, v) &= f(u, v) + a_\pi \text{ if } (u, v) \in \pi \\ &= f(u, v) - a_\pi \text{ if } (v, u) \in \pi \\ &= f(u, v) \text{ if } (u, v) \text{ and } (v, u) \notin \pi \end{aligned}$$

- Assuming  $f'$  is also a flow, we have then  $|f'| = |f| + a_\pi$  for if  $(F, u)$  is the first edge of  $\pi$ ,

$$\begin{aligned} |f'| &= \sum_v f'(F, v) = f'(F, u) + \sum_{v \neq u} f'(F, v) \\ &= f(F, u) + a_\pi + \sum_{v \neq u} f(F, v) \\ &= |f| + a_\pi \end{aligned}$$

### Finding Augmented Flows

- **Example:**

**The Augmented Flow is a Flow**

- Notice that  $f'$  does not change  $f$  for  $(u, v), (v, u) \notin \pi$
- Thus, to prove  $f'$  is a flow, we only check boundedness and antisymmetry on  $(u, v) \in \pi$  and conservation on  $u$ , leaving  $(v, u)$  as an exercise
- Boundedness: by definition of  $c_f$  we have

$$f'(u, v) = f(u, v) + a_\pi \leq f(u, v) + c_f(u, v) = c(u, v)$$

- Antisymmetry: we have

$$\begin{aligned} f'(u, v) &= f(u, v) + a_\pi = -(-f(u, v) - a_\pi) \\ &= -(f(v, u) - a_\pi) = -f'(v, u) \end{aligned}$$

by antisymmetry of  $f$

**The Augmented Flow is a Flow II**

- Conservation: assume we have  $\pi = \{\dots, (w, u), (u, v), \dots\}$ ; then

$$\begin{aligned} \sum_{z \neq u} f'(u, z) &= f'(u, w) + f'(u, v) + \sum_{other} f'(u, z) \\ &= f(u, w) - a_\pi + f(u, v) + a_\pi + \sum_{other} f(u, z) \\ &= \sum_{z \neq u} f(u, z) = 0 \\ &= 0 \end{aligned}$$

by definition of  $f'$  and conservation on  $f$

**Ford–Fulkerson Meta–Algorithm**

- If there is augmenting path for  $f$ ,  $|f|$  can be augmented, but if there is no augmenting path?
- **Theorem:** *The flow  $f$  is maximal iff there is no augmenting path in  $G_f$*

We prove it later

- This leads to the **Ford–Fulkerson** meta–algorithm

```

flow FF(graph G, capacity c)
  // we assume G and c extended to  $V \times V$ 
   $\mathbf{f} = 0$ ;  $\mathbf{c}_f = \mathbf{c}$ ;  $\mathbf{G}_f = \mathbf{G}$ ;
  while there is an AP  $\pi$  in  $\mathbf{G}_f$ :
    compute  $\mathbf{a}_\pi$ ;
     $\mathbf{f} = \text{update}(\mathbf{f}, \pi, \mathbf{a}_\pi)$ ;
    compute new  $\mathbf{G}_f$ ;
  return  $\mathbf{f}$ ;

```

- Notice that FF does not yet give us an algorithm

### Edmonds–Karp Algorithm

- We have to define how we obtain an augmenting path
- Simplest way: solve an unweighted minimum distance problem in  $G_F$  starting from  $F$ ;
  - If  $p[S] \neq \text{NULL}$ , we get an AP
- This leads to the **Edmonds–Karp** algorithm

```

flow EK(graph G, capacity c)
  // we assume G and c extended to  $V \times V$ 
   $f = 0; c_f = c; G_f = G;$ 
   $p[\ ] = \text{minDist}(G_f, F);$ 
  while  $p[S] \neq \text{NULL}$ :
     $a = \text{augment}(f, p[\ ]);$  (1)
     $f = \text{update}(f, p[\ ], a);$  (1)
    compute new  $G_f$ ; (1)
     $p[\ ] = \text{minDist}(G_f, F);$  (2)
  return  $f$ ;

```

### The Cost of Edmonds–Karp

- Computing  $a$  and updating  $f$  has a cost  $O(|V|)$  as  $\pi$  has at most  $|V|$  edges
- Applying the function  $\text{minDist}$  has a cost  $O(|E|)$  as we solve a minimum distance problem in an unweighted graph
- Thus, the cost of each loop iteration is  $O(|V|)$  in the (1) steps and  $O(|E|)$  in the (2) step
- It can be shown that the number of EK iterations is  $O(|E| |V|)$ ;
- Thus, we have  $\text{cost}_{EK} = O(|E|^2 |V|)$
- Hence, it is  $O(|V|^5)$  for dense graphs

## 6.2 Maximal Flows and Minimal Cuts

### Cuts

- A **cut** on  $G = (V, E)$  is any partition  $(V_1, V_2 = V - V_1)$  of  $V$  such that  $F \in V_1, S \in V_2$
- The **capacity**  $c(V_1, V_2)$  of a cut  $(V_1, V_2)$  is

$$c(V_1, V_2) = \sum_{u \in V_1, v \in V_2} c(u, v)$$

- The **flow**  $f(V_1, V_2)$  across a cut  $(V_1, V_2)$  is

$$f(V_1, V_2) = \sum_{u \in V_1, v \in V_2} f(u, v)$$



### Cut Properties

- **Property 1:** For any cut and any flow, we clearly have  $f(V_1, V_2) \leq c(V_1, V_2)$
- **Property 2:** We have  $|f| = f(\{F\}, V - \{F\})$  by the definition of  $|f|$ , i.e.,  $|f|$  is the flow across the cut  $(\{F\}, V - \{F\})$
- **Property 3:** For any flow  $f$ , any cut  $(V_1, V_2)$  and any  $v \in V_2$ , we have

$$f(V_1, V_2) = f(V_1 \cup \{v\}, V_2 - \{v\})$$

by conservation at  $v$

- **Property 4:** For any flow  $f$  and any cut  $(V_1, V_2)$ ,  $|f| = f(V_1, V_2)$ , for we simply start at Property 2 and keep on adding nodes until we go from  $\{F\}$  to  $V_1$

### Maximal Flows and Minimal Cuts

- Assume that for a flow  $f$  and a cut  $(V_1, V_2)$  we have  $f(V_1, V_2) = c(V_1, V_2)$ ; then
  - The flow is **maximal**, for if  $f'$  is another flow,

$$f'(V_1, V_2) \leq c(V_1, V_2) = f(V_1, V_2) = |f|$$

- The cut is **minimal**, for if  $(V'_1, V'_2)$  is another cut

$$c(V_1, V_2) = f(V_1, V_2) = f(V'_1, V'_2) \leq c(V'_1, V'_2)$$

In fact more can be said

- **Max Flow/Min Cut Theorem:** *If  $f$  is a flow in the flow network  $(G, c)$ , the following are equivalent:*
  1. *The flow is maximal*
  2. *There is no augmenting path in  $G_f$*
  3. *There is a (minimal) cut  $(V_1, V_2)$  such that  $f(V_1, V_2) = c(V_1, V_2)$*

### Proof of the Max Flow/Min Cut Theorem

- We prove it by showing  $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$
- $1 \Rightarrow 2$  is clear for if there would be an augmenting path, we could augment  $f$
- We have seen that  $3 \Rightarrow 1$  in Property 1
- For  $2 \Rightarrow 3$ , set  $V_1 = \{v : v \text{ reachable from } F \text{ in } G_f\}$ ; then
  - Since  $S \notin V_1$ , we have a partition
  - If  $f(V_1, V_2) < c(V_1, V_2)$ , there must be an edge  $(u, v)$  with  $u \in V_1, v \in V_2$  and  $f(u, v) < c(u, v)$ , i.e.,  $c_f(u, v) > 0$

- But since we can reach  $u$  from  $F$  in  $G_f$ , we could also reach  $v$ , which is a contradiction with  $v \in V_2$
- Thus, we must have  $f(V_1, V_2) = c(V_1, V_2)$

### Consequences

- Clearly for any flow  $f$  we have  $|f| \leq \phi^* = \sum c(F, u)$
- If  $c(u, v)$  are integers, FF or EK iterations give flows  $f_i$  such that  $|f_i| < |f_i| + 1 \leq |f_{i+1}|$
- Therefore, FF and EK cannot make more than  $\phi^*$  iterations
- Thus, they are correct for they arrive to a maximal flow in a finite number of steps
- However, one can build network flows for non integer  $c$  for which EK do not end in a finite number of steps, approaching instead the maximum flow value  $|f^*|$  asymptotically