Minimum weight spanning tree, TSP

László Papp

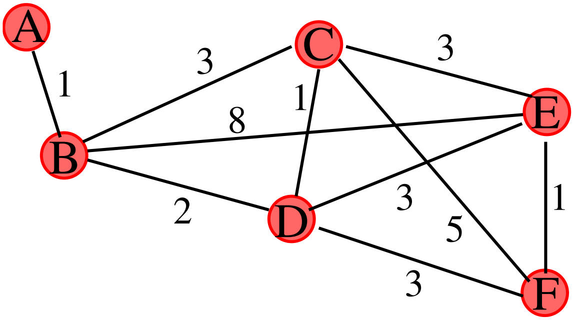
BME

17th of March, 2023

A combinatorial optimization problem:

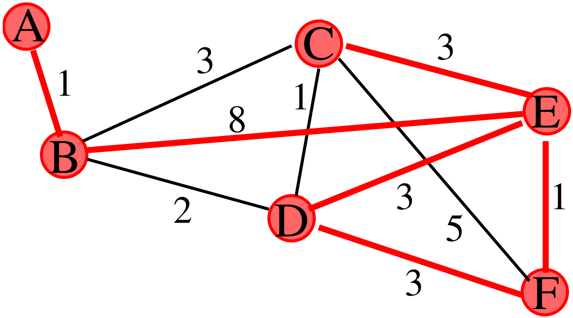
We have six towns and we want to build a telecommunication network such that we can send a message between any two cities over the network. Due to some reasons, we do not want to connect cables outside of the towns. We know in advance that which towns can be connected by a direct wire and how much is the cost of such a wire.

Task: Find the cheapest connected network!



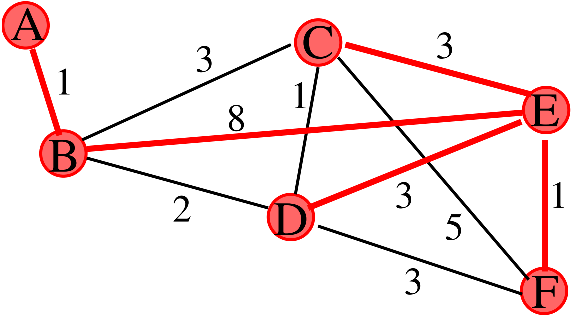
A not optimal solution

An optimal solution does not contain a cycle, because if we delete an edge contained in the cycle the network remains connected and the cost of the network decreases (the cost of each edge is positive).



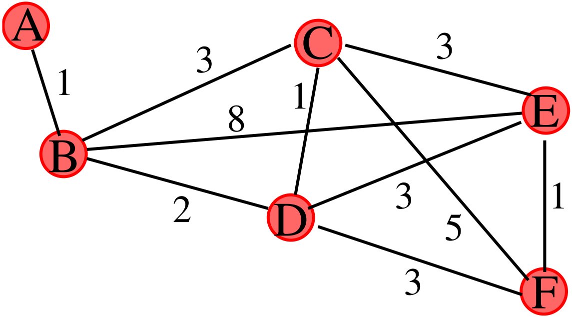
A not optimal solution

An optimal solution does not contain a cycle, because if we delete an edge contained in the cycle the network remains connected and the cost of the network decreases (the cost of each edge is positive).



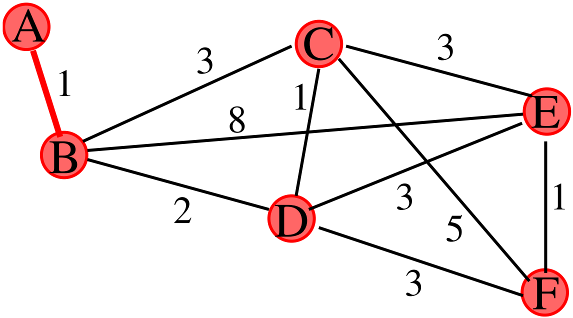
So we are looking for a spanning tree, but not all spanning trees are good enough.
The price of this network is $1 + 8 + 3 + 3 + 1 = 16$.

Finding an optimal solution



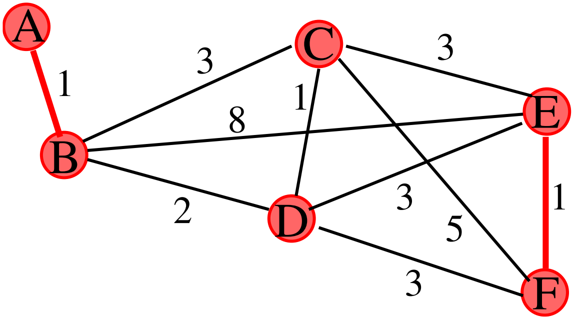
Each step we choose the cheapest edge which does not makes a cycle with the edges chosen earlier. This is a greedy algorithm, since at each step we choose the locally best option.

Finding an optimal solution



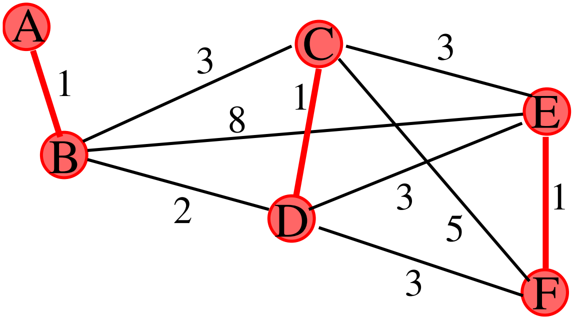
Each step we choose the cheapest edge which does not makes a cycle with the edges chosen earlier. This is a greedy algorithm, since at each step we choose the locally best option.

Finding an optimal solution



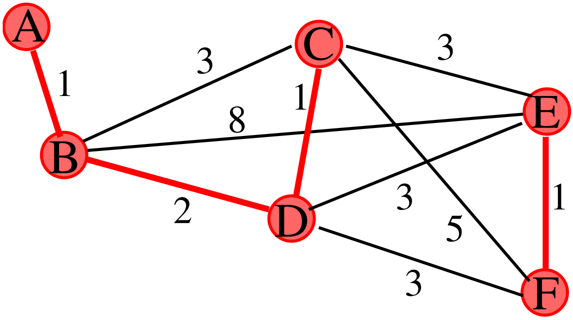
Each step we choose the cheapest edge which does not makes a cycle with the edges chosen earlier. This is a greedy algorithm, since at each step we choose the locally best option.

Finding an optimal solution



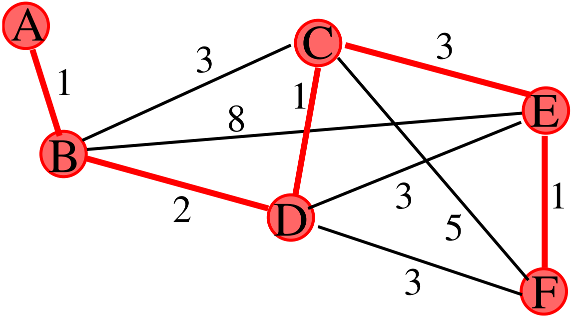
Each step we choose the cheapest edge which does not makes a cycle with the edges chosen earlier. This is a greedy algorithm, since at each step we choose the locally best option.

Finding an optimal solution



Each step we choose the cheapest edge which does not makes a cycle with the edges chosen earlier. This is a greedy algorithm, since at each step we choose the locally best option.

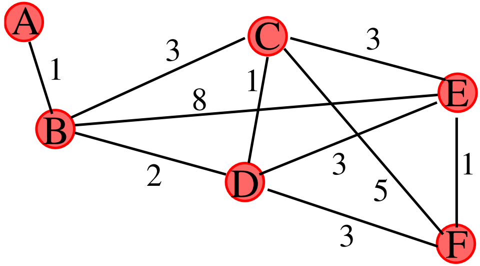
Finding an optimal solution



Each step we choose the cheapest edge which does not makes a cycle with the edges chosen earlier. This is a greedy algorithm, since at each step we choose the locally best option. So the price of a cheapest network is 8 and we also have obtained such a network.

Minimum weight spanning tree problem

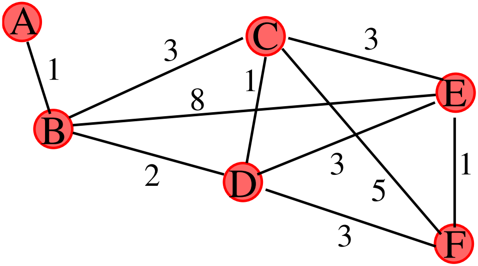
Let G be a graph and $w : E(G) \to \mathbb{R}$ be a real-valued function over the edge set of G. This function w tells us the weight (or cost) of the edges. If we take a subgraph H of G, then the weight (cost) of H is $\sum_{e \in E(H)} w(e)$.



Task: Find a spanning tree of G, whose weight is the smallest possible!

Minimum weight spanning tree problem

Let G be a graph and $w : E(G) \to \mathbb{R}$ be a real-valued function over the edge set of G. This function w tells us the weight (or cost) of the edges. If we take a subgraph H of G, then the weight (cost) of H is $\sum_{e \in E(H)} w(e)$.



Task: Find a spanning tree of G, whose weight is the smallest possible!

During our example problem, we have solved this task in a greedy way.

Kruskal's algorithm

Input: Graph G and a weight function $w : E(G) \to \mathbb{R}$.

1. Sort the edges of the graph to ascending order according to their weight: $w(e_1) \leq w(e_2), \leq \ldots \leq w(e_{|E(G)|})$.
2. Let F be the graph containing all the vertices of G but none of its edges and let $i := 1$.
3. If $E(F) \cup e_i$ does not contain a cycle then add e_i to $E(F)$.
4. If $i < |E(G)|$, then increase i by one and move to the 3rd step.

Output: F.

Claim

If a graph G is connected, then Kruskal's algorithm gives a minimum weight spanning tree of G.

Remark: We have run this algorithm previously.

Greedy algorithms

Definition: An algorithm is called **greedy** if at each choice it chooses the locally best option.

Kruskal's algorithm is a greedy algorithm, because at each step it tries to include the lightest (cheapest) edge.

Remark: Usually greedy steps and greedy algorithms do not lead to optimal solutions. We are going to see examples for this phenomenon later.

How fast is Kruskal's algorithm?

The time complexity of Kruskal's algorithm is $O(e \log(e))$ where e is the number of edges in the input graph. Sorting the edges according to their weight requires $\Theta(e \log(e))$ operations in the worst case. This is the main term here, but we will not give a reasoning for that.

Questions regarding the effectiveness of Kruskal's algorithm:

- How can we encode a graph?
- Is it much better than the brute-force method?

The brute-force method: Consider each spanning-tree of the graph, calculate its weight then choose the smallest one.

Note: This is not yet an algorithm because we have not specified how to find all the spanning trees.

How to encode (simple) graphs

Reminder: A graph $G = (V, E)$ is an ordered pair of sets, where V is the set of vertices and E is the set of edges containing pairs of vertices.

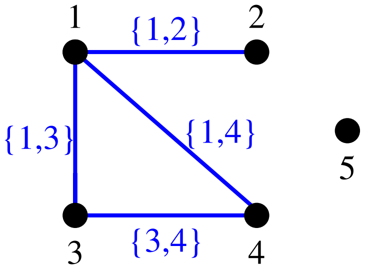
There are two major (+some other) methods to encode a graph:

Adjacency list: For each vertex we write down the set of adjacent vertices.

Example

For the given graph it is:

1 : 2, 3, 4;
2 : 1;
3 : 1, 4;
4 : 1, 3;
5 :



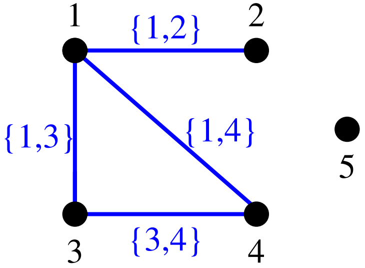
If the alphabet contains more symbols than the number of vertices (the lenght of the vertex labels are neglected), then the size of an adjacency list is $\Theta(e + n)$, where e and n denote the number of edges and the number of vertices, respectively.

How to encode graphs

Adjacency matrix: Each vertex has a corresponding column and a row. $A_{i,j}$ equals 1 if vertices i and j are adjacent and 0 otherwise. (If there are paralell edges then $A_{i,j}$ equals to the number of edges i $\{i, j\}$)

Example

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



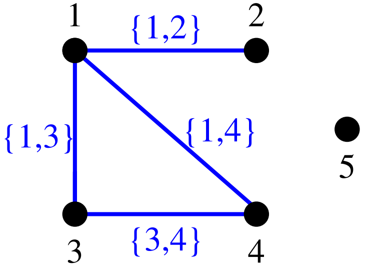
The size of the adjacency matrix is n^2.

How to encode graphs

Adjacency matrix: Each vertex has a corresponding column and a row. $A_{i,j}$ equals 1 if vertices i and j are adjacent and 0 otherwise. (If there are paralell edges then $A_{i,j}$ equals to the number of edges i $\{i, j\}$)

Example

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



The size of the adjacency matrix is n^2.

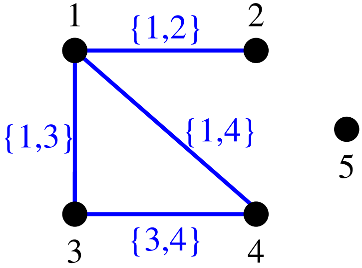
Question: Which encoding requires less space?

How to encode graphs

Adjacency matrix: Each vertex has a corresponding column and a row. $A_{i,j}$ equals 1 if vertices i and j are adjacent and 0 otherwise. (If there are paralell edges then $A_{i,j}$ equals to the number of edges i $\{i, j\}$)

Example

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



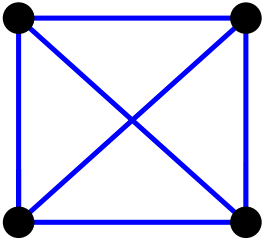
The size of the adjacency matrix is n^2.

Question: Which encoding requires less space?
Answer: Usually the adjacency list.

Complete graphs

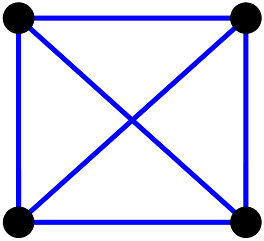
Definition A **complete graph** is a simple graph where any two vertices are adjacent. The complete graph having n vertices is denoted by K_n.

Question: How many edges does K_n have?



Complete graphs

Definition A **complete graph** is a simple graph where any two vertices are adjacent. The complete graph having n vertices is denoted by K_n.



Question: How many edges does K_n have?

Answer: $\frac{n(n-1)}{2}$, because: From each vertex $n-1$ edges go to the other vertices. If we sum it for all vertices, then we obtain $n(n-1)$. We have counted each edge twice, at both of its endpoints. To compensate this we divide this number by 2.

Remark: Any simple graph having n vertices is a subgraph of K_n. Therefore it has at most $\frac{n(n-1)}{2}$ edges.

Corollary: The size of a simple graph's adjacency list is in $O(n^2)$.

Time complexity of the the brute-force algorithm

The brute force algorithm checks each spanning tree.
In K_n there are n^{n-2} spanning trees. (If you are interested in the proof, search for Cayley's formula.)

Time complexity of the the brute-force algorithm

The brute force algorithm checks each spanning tree.
In K_n there are n^{n-2} spanning trees. (If you are interested in the proof, search for Cayley's formula.)

That is so much. The function $f(n) = n^{n-2} \notin O(2^n)$. It grows faster than any exponential function.

So the brute-force algorithm runs so slow if the input graph is a complete graph, but what if it is something else?

Time complexity of the the brute-force algorithm

The brute force algorithm checks each spanning tree.
In K_n there are n^{n-2} spanning trees. (If you are interested in the proof, search for Cayley's formula.)

That is so much. The function $f(n) = n^{n-2} \notin O(2^n)$. It grows faster than any exponential function.

So the brute-force algorithm runs so slow if the input graph is a complete graph, but what if it is something else?

Answer: It is too slow for most of the possible input graphs. So avoid it!

Time complexity of the the brute-force algorithm

The brute force algorithm checks each spanning tree.
In K_n there are n^{n-2} spanning trees. (If you are interested in the proof, search for Cayley's formula.)

That is so much. The function $f(n) = n^{n-2} \notin O(2^n)$. It grows faster than any exponential function.

So the brute-force algorithm runs so slow if the input graph is a complete graph, but what if it is something else?

Answer: It is too slow for most of the possible input graphs. So avoid it!

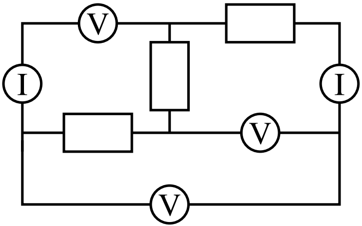
Remark: Checking all the possible solutions, evaluating the objective function for each of them and picking the best one is usually a bad idea. It can be done in finite time, but in most of the cases it takes way to much time. This is the reason why we are looking for smart algorithms!

Summary for Kruskal's algorithm

- It finds a minimum weight spanning tree.
- It runs in $O(e \log e)$ time, which is just a little bit more than the O(e) steps which is required to read the adjacency list.
- It is much faster than the brute-force algorithm.
- It is a greedy algorithm.

An application: Normal trees

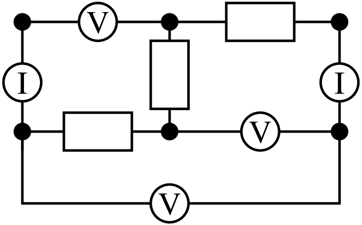
Assume that we have an electric circuit with three type of components: resistors, voltage sources and current sources.



We create a graph from the electrical circuit: The vertices are the equipotential surfaces and the edges are the components. A **normal tree** of the circuit is a spanning tree which contains all the voltage sources but none of the current sources.

An application: Normal trees

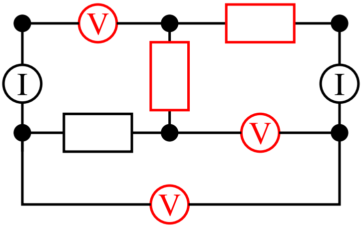
Assume that we have an electric circuit with three type of components: resistors, voltage sources and current sources.



We create a graph from the electrical circuit: The vertices are the equipotential surfaces and the edges are the components. A **normal tree** of the circuit is a spanning tree which contains all the voltage sources but none of the current sources.

An application: Normal trees

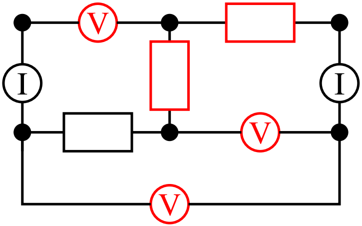
Assume that we have an electric circuit with three type of components: resistors, voltage sources and current sources.



We create a graph from the electrical circuit: The vertices are the equipotential surfaces and the edges are the components. A **normal tree** of the circuit is a spanning tree which contains all the voltage sources but none of the current sources.

The use of normal trees

We know the properties of the electronic components: resistence of resistors, supplied voltage of voltage sources and supplied current of current sources. We want to determine the voltage and current across each component by using Kirchoff's circuit laws. Sometimes this cannot be done, because there are infinitely many solutions.

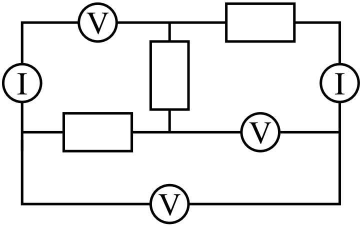


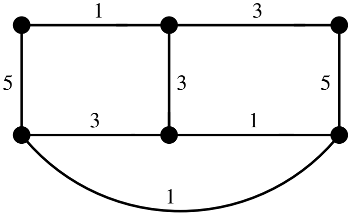
Claim:
If the circuit does not have a normal tree, then the Kirchoff's laws does not give a unique solution.

Finding a normal tree

We assign weight to the components by the following rule:

- Voltage source 1
- Resistor 3
- Current source 5

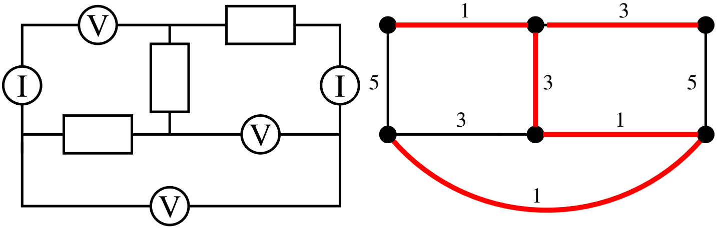




Finding a normal tree

We assign weight to the components by the following rule:

- Voltage source 1
- Resistor 3
- Current source 5

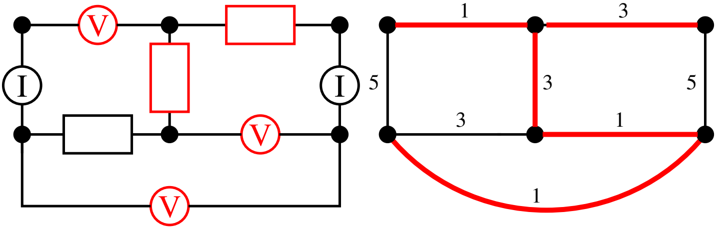


We search a minimum weight spanning tree by Kruskal's algorithm. If it contains all the voltage sources and none of the current sources, then it is a normal tree. Otherwise the circuit does not have a normal tree.

Finding a normal tree

We assign weight to the components by the following rule:

- Voltage source 1
- Resistor 3
- Current source 5



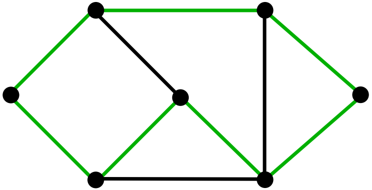
We search a minimum weight spanning tree by Kruskal's algorithm. If it contains all the voltage sources and none of the current sources, then it is a normal tree. Otherwise the circuit does not have a normal tree.

Hamiltonian cycle, Hamiltonian path

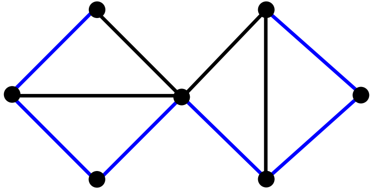
Definition: H is a **Hamiltonian cycle** of graph G if H is a cycle, $H \subseteq G$ and it contains all vertices of G.

Definition: P is a **Hamiltonian path** of graph G if P is a path, $H \subseteq G$ and it contains all vertices of G.

Examples:



Hamiltonian cycle



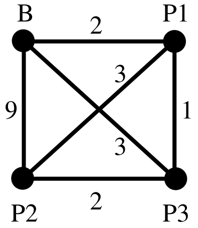
Hamiltonian path

Traveling salesman problem (TSP)

Real-world problem:

We have $n-1$ pubs in a town, a brewery and a truck. We want to supply the pubs with beer by using the truck. We know the pairwise distance between the n places. We want to distribute the beer traveling the least distance. How to do it?

As a mathematical problem: A complete graph with a weight function on its edge set is given. We are searching for a Hamiltonian cycle of minimum weight in the graph (called as an optimal tour or shortest tour).



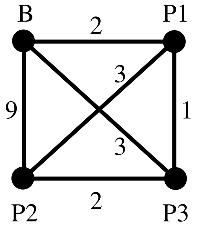
How to encode our real-world problem as the mathematical problem?

Traveling salesman problem (TSP)

Real-world problem:

We have $n-1$ pubs in a town, a brewery and a truck. We want to supply the pubs with beer by using the truck. We know the pairwise distance between the n places. We want to distribute the beer traveling the least distance. How to do it?

As a mathematical problem: A complete graph with a weight function on its edge set is given. We are searching for a Hamiltonian cycle of minimum weight in the graph (called as an optimal tour or shortest tour).



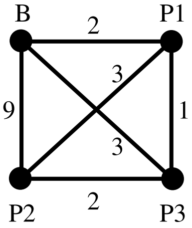
How to encode our real-world problem as the mathematical problem? The vertex set contains the pubs and the brewery and the weight function over the edge set is their pairwise distance.

TSP has application in many areas, for example: logistics, DNA sequencing, etc.

Trying to solve TSP with a greedy algorithm (nearest neighbor)

The algorithm which we are going to use is the following:

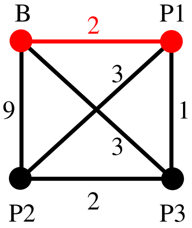
1. Start from a random vertex v and a subgraph H containing only vertex v. Remember this vertex, so let $u := v$.
2. Choose the lightest edge which is incident to v and its other endpoint is not contained in H. Add this edge with its endpoint to H and let v be the recently added vertex.
3. If H does not contain all the vertices of the input graph, then repeat step 2. Otherwise add the edge whose endpoints are v and u.



Trying to solve TSP with a greedy algorithm (nearest neighbor)

The algorithm which we are going to use is the following:

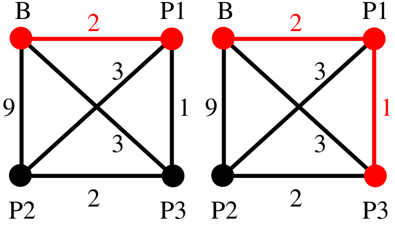
1. Start from a random vertex v and a subgraph H containing only vertex v. Remember this vertex, so let $u := v$.
2. Choose the lightest edge which is incident to v and its other endpoint is not contained in H. Add this edge with its endpoint to H and let v be the recently added vertex.
3. If H does not contain all the vertices of the input graph, then repeat step 2. Otherwise add the edge whose endpoints are v and u.



Trying to solve TSP with a greedy algorithm (nearest neighbor)

The algorithm which we are going to use is the following:

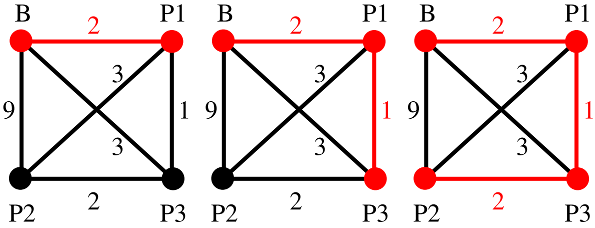
1. Start from a random vertex v and a subgraph H containing only vertex v. Remember this vertex, so let $u := v$.
2. Choose the lightest edge which is incident to v and its other endpoint is not contained in H. Add this edge with its endpoint to H and let v be the recently added vertex.
3. If H does not contain all the vertices of the input graph, then repeat step 2. Otherwise add the edge whose endpoints are v and u.



Trying to solve TSP with a greedy algorithm (nearest neighbor)

The algorithm which we are going to use is the following:

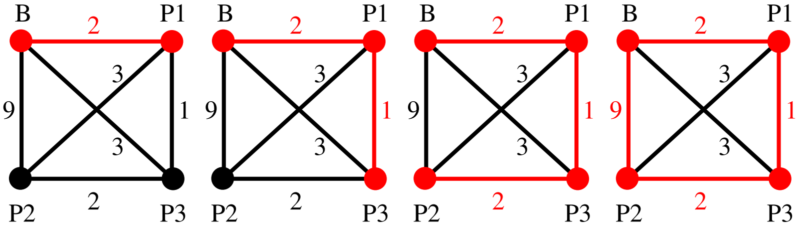
1. Start from a random vertex v and a subgraph H containing only vertex v. Remember this vertex, so let $u := v$.
2. Choose the lightest edge which is incident to v and its other endpoint is not contained in H. Add this edge with its endpoint to H and let v be the recently added vertex.
3. If H does not contain all the vertices of the input graph, then repeat step 2. Otherwise add the edge whose endpoints are v and u.



Trying to solve TSP with a greedy algorithm (nearest neighbor)

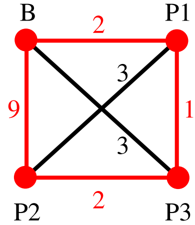
The algorithm which we are going to use is the following:

1. Start from a random vertex v and a subgraph H containing only vertex v. Remember this vertex, so let $u := v$.
2. Choose the lightest edge which is incident to v and its other endpoint is not contained in H. Add this edge with its endpoint to H and let v be the recently added vertex.
3. If H does not contain all the vertices of the input graph, then repeat step 2. Otherwise add the edge whose endpoints are v and u.



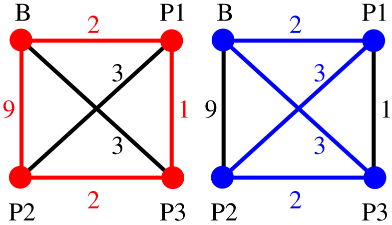
Analyzing the algorithm

The weight of the solution given by this greedy algorithm is $2 + 1 + 2 + 9 = 14$.



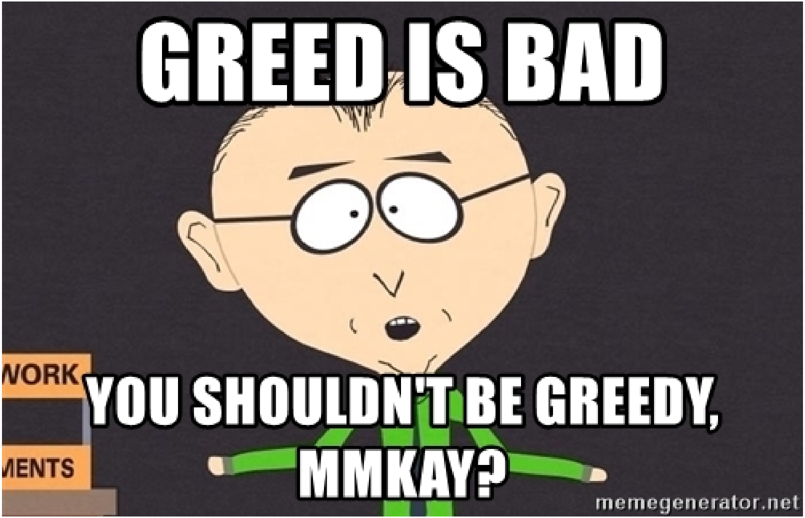
Analyzing the algorithm

The weight of the solution given by this greedy algorithm is $2 + 1 + 2 + 9 = 14$.



The weight of the optimal solution is $2 + 3 + 2 + 3 = 10$.
So this greedy algorithm does not give us the optimal solution.

Conclusion: Greedy algorithms usually give bad results.



Except when you are searching for a minimum weight spanning tree...

How to solve the TSP?

Question: Is there an algorithm which gives us the optimal solution?

How to solve the TSP?

Question: Is there an algorithm which gives us the optimal solution?

Of course there are some. For example calculating the weight of each Hamiltonian cycle and choosing the minimal works. This is a brute force algorithm. However there are $(n-1)!/2$ Hamiltonian cycles in a complete graph, therefore it takes so much time.

Note that $(n-1)!/2 \notin O(2^n)$. It is increasing much faster than the exponential function. For example $(20-1)!/2 = 6 \cdot 10^{16}$.

How to solve the TSP?

Question: Is there an algorithm which gives us the optimal solution?

Of course there are some. For example calculating the weight of each Hamiltonian cycle and choosing the minimal works. This is a brute force algorithm. However there are $(n-1)!/2$ Hamiltonian cycles in a complete graph, therefore it takes so much time.

Note that $(n-1)!/2 \notin O(2^n)$. It is increasing much faster than the exponential function. For example $(20-1)!/2 = 6 \cdot 10^{16}$.

Unfortunately no polinomial time algorithm is known which solves TSP.

How to solve the TSP?

Question: Is there an algorithm which gives us the optimal solution?

Of course there are some. For example calculating the weight of each Hamiltonian cycle and choosing the minimal works. This is a brute force algorithm. However there are $(n-1)!/2$ Hamiltonian cycles in a complete graph, therefore it takes so much time.

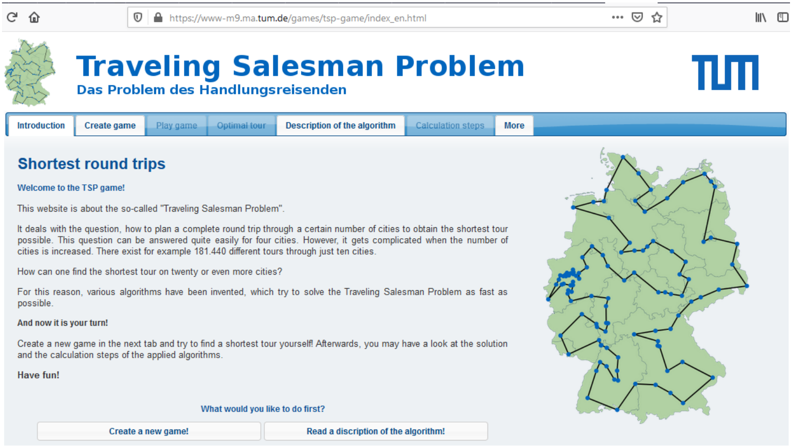
Note that $(n-1)!/2 \notin O(2^n)$. It is increasing much faster than the exponential function. For example $(20-1)!/2 = 6 \cdot 10^{16}$.

Unfortunately no polinomial time algorithm is known which solves TSP.

Later we will see some methods which can be used to attack the TSP.

Play with the TSP

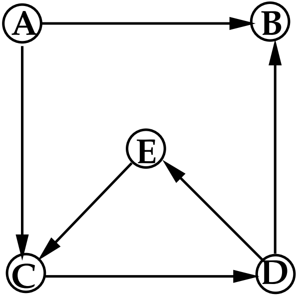
If you want to play with the travelling salesman problem, then visit https://algorithms.discrete.ma.tum.de/graph-games/tsp-game/index_en.html.



Directed graphs or digraphs

In a directed graph each edge has an orientation, so the corresponding pair has a fixed order. The first element is the **tail** and the second element is the **head** of the edge.

$V(\vec{G}) = \{A, B, C, D\}$
$E(\vec{G}) =$
$\{(A, C), (A, B), (C, D),$
$(D, B)\, (D, E), (E, C)\}$

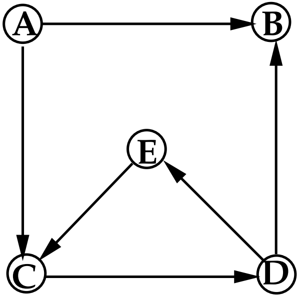


So in $(A, \vec{C})$ A is the tail and C is the head. If $(A, \vec{C})$ is an edge, then we say that C is an **out-neighbor** of A and A is an **in-neighbor** of C.

In an undirected graph A is a **neighbor** of B if they are adjacent, equivalently (A, B) is an edge.

Directed graphs, more definitions

Definition: A vertex u is a **source** if there is no edge whose head is u. Similarly v is a **sink** if there is no edge whose tail is v.



Example: In this digraph A is a source and B is a sink.

Definition: In a directed graph a $(v_0, \vec{e_1}, v_1, \vec{e_2}, v_2, \ldots, c_{k-1}, \vec{e_k}, v_k)$ path (cycle) is a **directed path (directed cycle)** if $\vec{e_i} = (v_{i-1}, v_i)$.

Example: $(E, \{E, C\}, C, \{C, D\}, D, \{D, E\}, E)$ is a directed cycle in the above digraph.