

Algorithms and Data Structures

Other graph problems

dr Szymon Murawski

Comarch SA

May 29, 2019

Table of contents I

- 1 Minimum spanning tree
 - Kruskal's algorithm for MST
 - Prim's algorithm for MST
- 2 Euler path
- 3 Traveling salesman problem
- 4 Stable marriage problem

Course plan

- 1 Minimum spanning tree
 - Kruskal's algorithm for MST
 - Prim's algorithm for MST
- 2 Euler path
- 3 Traveling salesman problem
- 4 Stable marriage problem

Minimum spanning tree

- Minimum spanning tree (MST) of a graph is a subgraph, that is a tree, and connects all vertices in original graph
- MST will not contain redundant edges in graph
- For weighted graphs sum of edges in MST will be lower or equal to weight of every other spanning tree
- A graph can have multiple MSTs
- MST have multiple applications, for example when designing how to lay network cables in suburbs to minimize earthworks or optimize information distribution in some system.
- Algorithm for finding MST was first developed in 1926 to design efficient electric grid in Czech Republic
- A minimum spanning tree has exactly $V - 1$ edges

Course plan

- 1 Minimum spanning tree
 - Kruskal's algorithm for MST
 - Prim's algorithm for MST
- 2 Euler path
- 3 Traveling salesman problem
- 4 Stable marriage problem

Kruskal's algorithm

- Kruskal's algorithm for finding minimum spanning tree is a greedy algorithm
- First all edges are sorted according to their weight
- Next for every edge we check if adding it to MST would form a cycle. If not, then it is added to minimum spanning tree

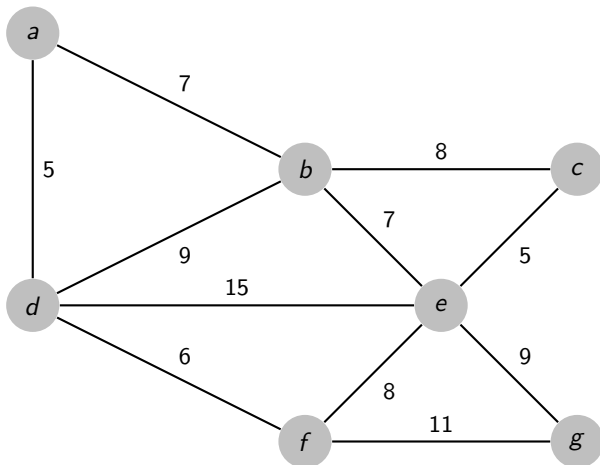
Pseudocode

```
1 KruskalMST(Graph){
2   Tree MST
3   sort Graph.Edges
4   for every edge in Graph.Edges
5     if FindRoot(edge.u) != FindRoot(edge.v)
6       MST.add(edge)
7       UNION(edge.u, edge.v)
8   return MST
9 }
```

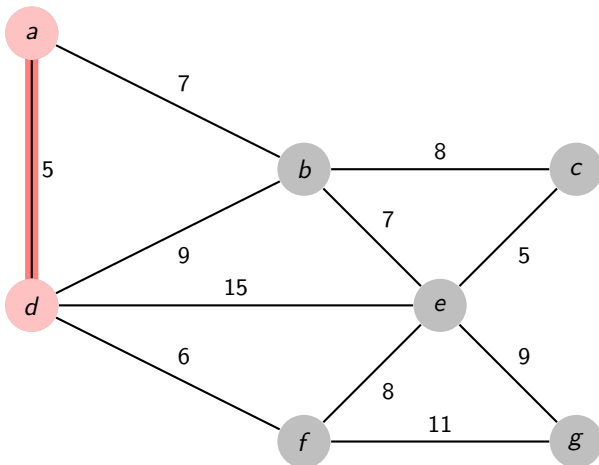
Kruskal's algorithm details

- Checking if adding an edge to minimum spanning tree would form a cycle is done by checking if root of vertex on one end of an edge is the same as for the vertex on other end
- Since initially no nodes are in MST, then the root of every node is just that node
- After we add an edge to MST we execute UNION function, to merge two trees into one.
- This procedure maintains the tree property emerging trees in a graph - every node in a tree has the same parent.

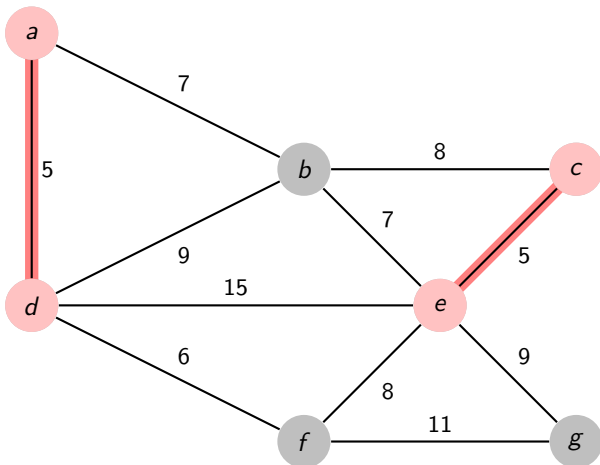
Kruskal's algorithm example



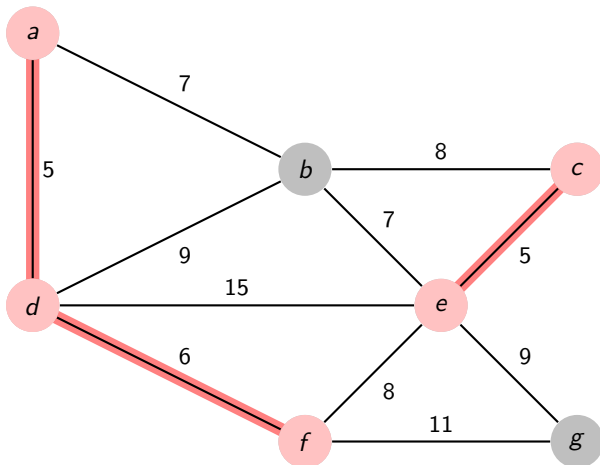
Kruskal's algorithm example



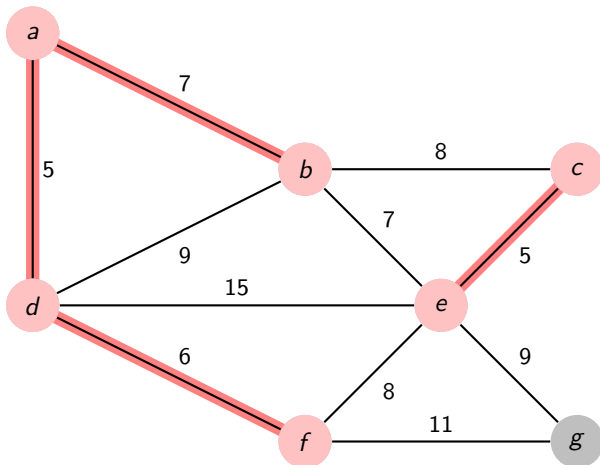
Kruskal's algorithm example



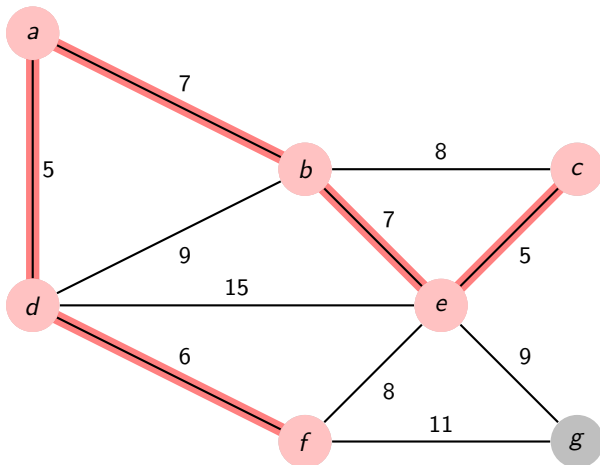
Kruskal's algorithm example



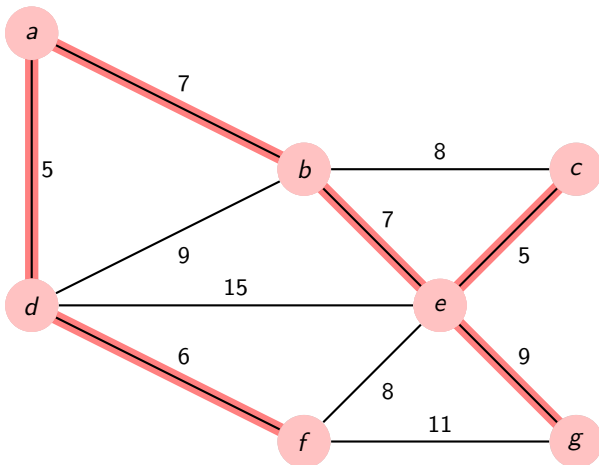
Kruskal's algorithm example



Kruskal's algorithm example



Kruskal's algorithm example



Course plan

- 1 Minimum spanning tree
 - Kruskal's algorithm for MST
 - Prim's algorithm for MST
- 2 Euler path
- 3 Traveling salesman problem
- 4 Stable marriage problem

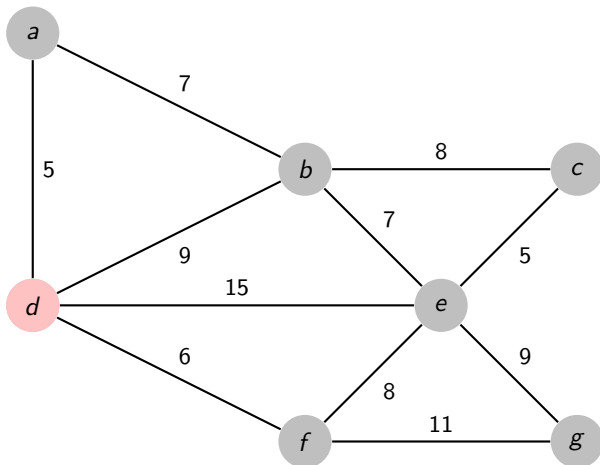
Prim's algorithm

- 1 Initialize MST with a single vertex chosen arbitrary from graph
- 2 For all edges that connect a vertex from MST to vertex not yet in MST choose one with minimum weight and add it to MST
- 3 Repeat above step until all vertices are in MST

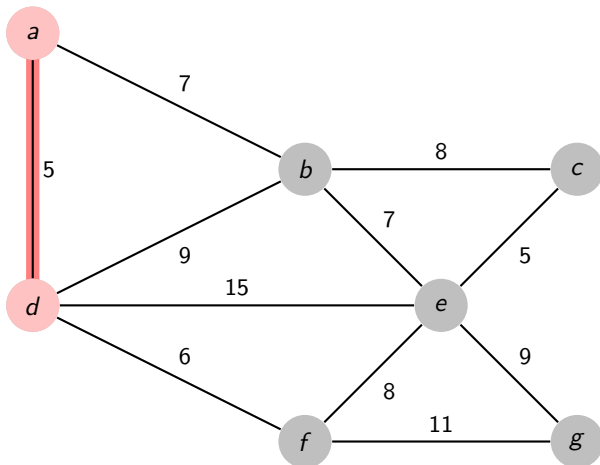
Pseudocode

```
1 PrimMST( Graph )
2   Tree MST
3   startingVertex = Graph.Vertices[0];
4   MST.add(startingVertex)
5   PriorityQueue EdgesQueue
6   EdgesQueue.add( startingVertex.Neighbours() ) //Priority is
   weight of edge
7   while (MST.Size != Graph.Size)
8     currEdge = Edges.Queue.dequeue() //Dequeues weight with
   lowest weight
9     if ! MST.Contains(currEdge.end) //End vertex of edge
10      MST.Add(currEdge)
11      EdgesQueue.add( currEdge.end.Neighbours() )
12   return MST
```

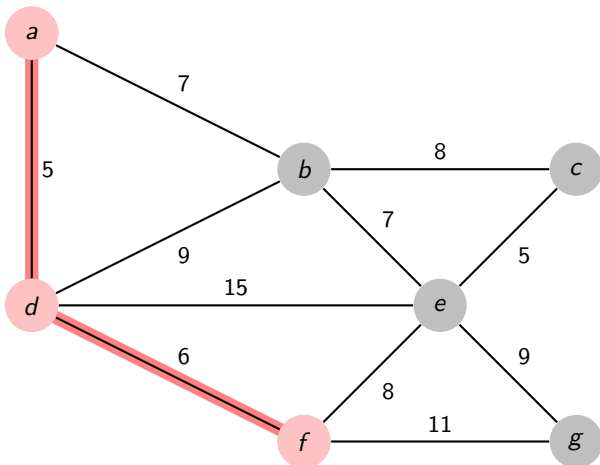

Prim's algorithm example



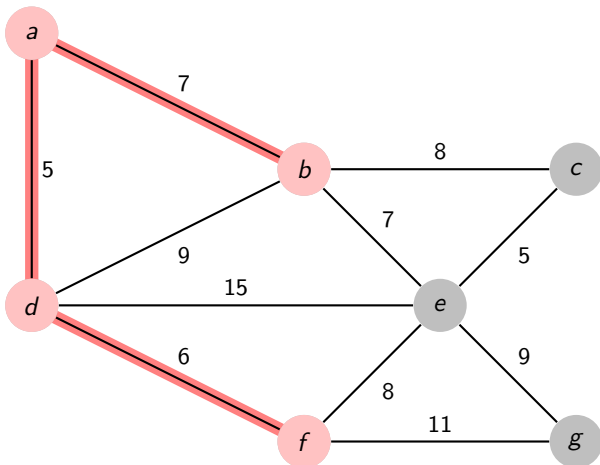
Prim's algorithm example



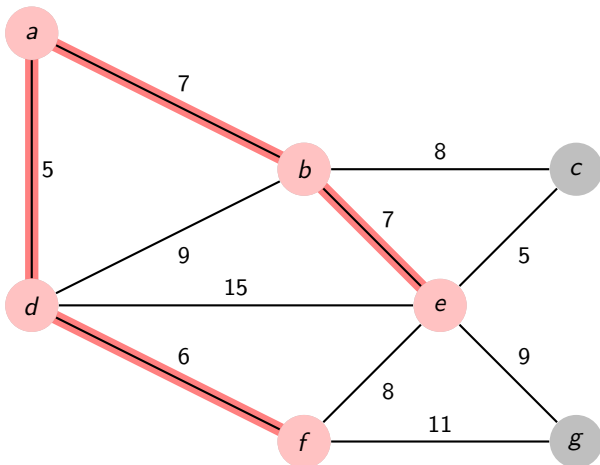
Prim's algorithm example



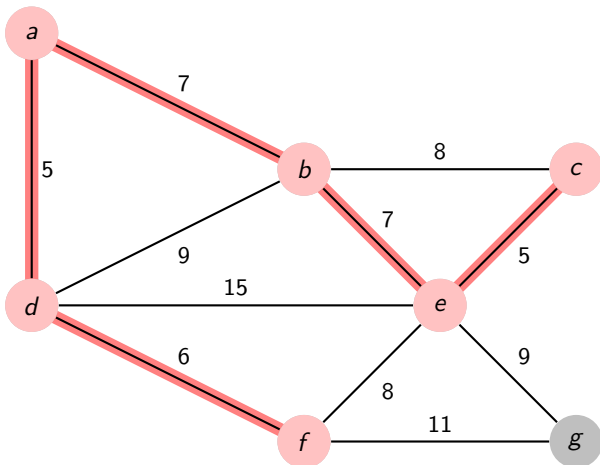
Prim's algorithm example



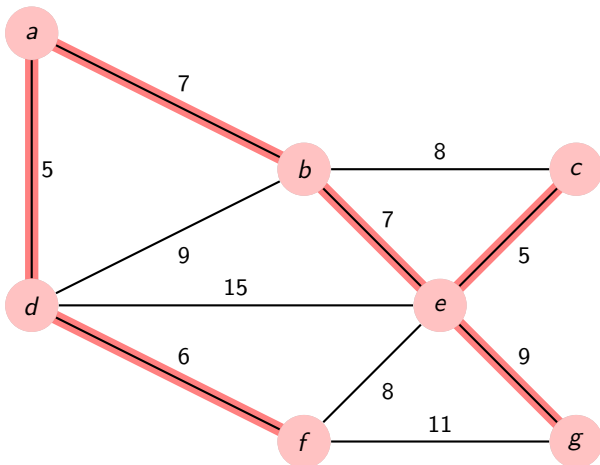
Prim's algorithm example



Prim's algorithm example



Prim's algorithm example



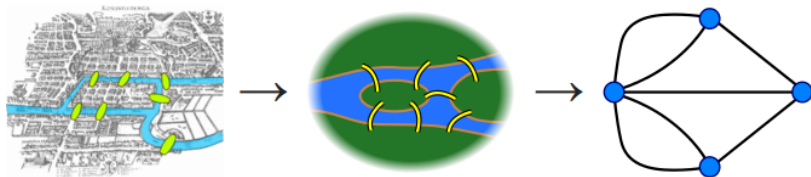
Kruskal's and Prim's algorithms analysis

- Both algorithms find minimum spanning tree using greedy programming approach
- Complexity of both algorithms is the same: $O(E \lg V)$
- By using advanced data structures (Fibonacci Heaps) Prim's algorithm can be optimized to run in $O(E + \lg V)$.
- Concluding, if we have dense graph, where there are much more edges than vertices, Prim's algorithm should be used. In typical situations it's better to use Kruskal's, because it's simpler.

Course plan

- 1 Minimum spanning tree
 - Kruskal's algorithm for MST
 - Prim's algorithm for MST
- 2 Euler path
- 3 Traveling salesman problem
- 4 Stable marriage problem

Eulerian path



- Eulerian path (tour) is a path in graph that visits every edge exactly once
- A graph has Eulerian path if degree of every vertex is even, or there are exactly two vertices with odd degree
- In the second case we start at one of the odd-degree vertex and end tour in second vertex
- To find Eulerian path we can use Fleury's algorithm

Fleury's algorithm

- This algorithm is not the fastest, but it's elegant
- The idea for this algorithm is to avoid disconnecting the graph - we cannot burn bridges.
- A bridge is an edge that is the only connection between two parts of the graph. Once we traverse it we cannot go back!

Algorithm

- Choose a starting vertex (odd one or random)
- Until there are unvisited edges follow them one by one
- While making a choice which edge to follow always choose non-bridge over bridge edge - don't disconnect the graph!

Fleury's algorithm pseudocode

```
1 EulerianPath(Graph)
2   startVertex = FindStartVertex(Graph)
3   EulerianPathRecursive(Graph, startVertex)
4
5 FindStartVertex(Graph)
6   startVertex = Graph.vertices[0]
7   foreach vertex in Graph.vertices
8     if vertex.degree is odd
9       startVertex = vertex
10    break
11   return startVertex
```

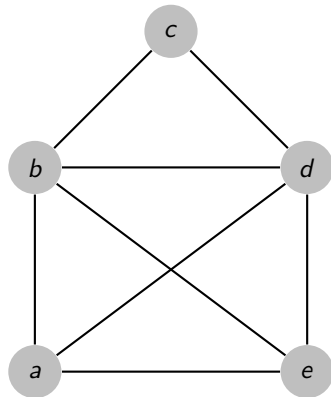
Fleury's algorithm pseudocode continued

```

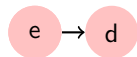
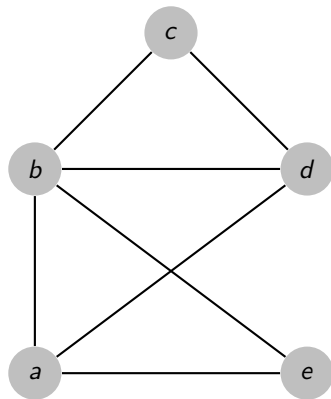
12 EulerianPathRecursive(Graph, startVertex)
13   foreach neighbour in Graph.neighbours(startVertex)
14     if Graph.areConnected(startVertex, neighbour) && (startVertex
        .degree == 1 || !Graph.IsBridge(startVertex, neighbour))
15       print startVertex + "-" + neighbour
16       Graph.RemoveEdge(startVertex, neighbour)
17       EulerianPathRecursive(neighbour)
18
19 IsBridge(start, end)
20   count1 = DFS(start) // calculate how many vertices are reachable
        from start vertex
21   Graph.RemoveEdge(start, end)
22   count2 = DFS(start) // how about when we delete the edge?
23   Graph.AddEdge(start, end)
24   if count1 > count2
25     return true
26   return false

```

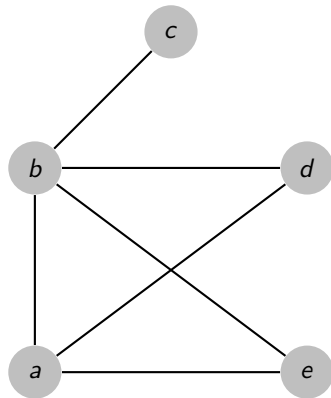
Fleury's algorithm example



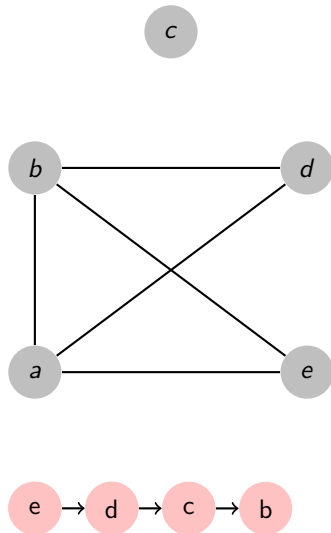
Fleury's algorithm example



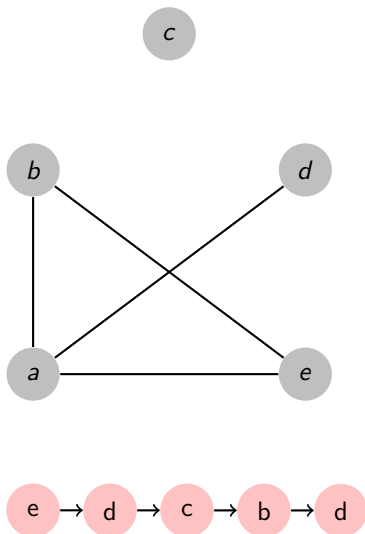
Fleury's algorithm example



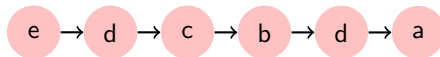
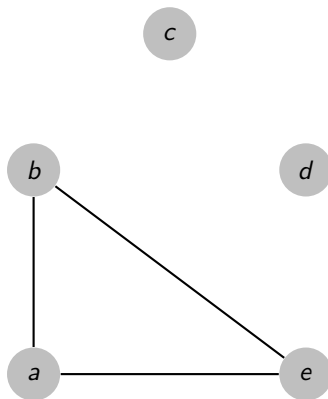
Fleury's algorithm example



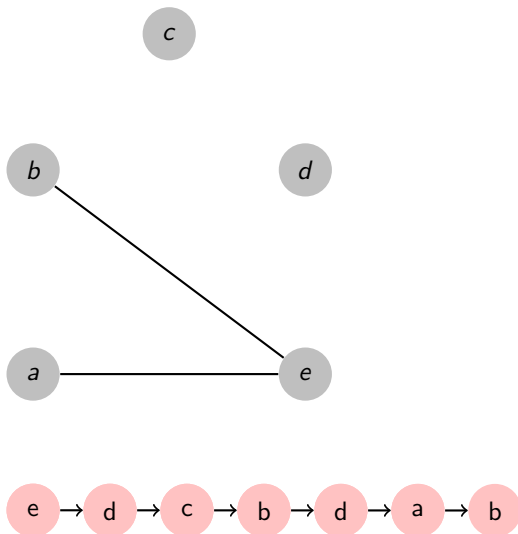
Fleury's algorithm example



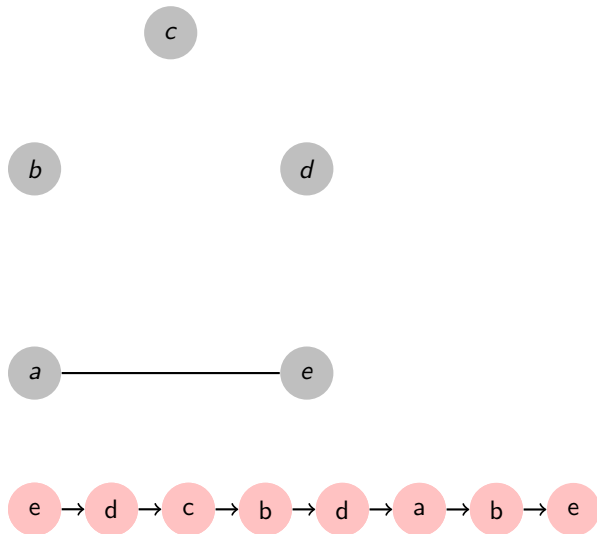
Fleury's algorithm example



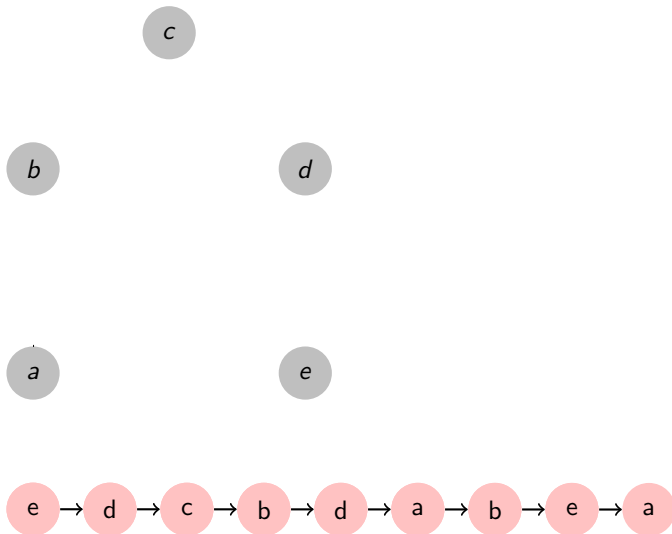
Fleury's algorithm example



Fleury's algorithm example



Fleury's algorithm example



Fleury's algorithm analysis

- This algorithm runs in $O(E^2)$
- There are more efficient algorithms for finding an Eulerian path, but this one is quite elegant
- Similar problem is Eulerian cycle - in addition of it being an Eulerian path, it must also start and end at the same vertex
- There is an $O(E)$ algorithm for finding such a cycle - Hierholzer's algorithm
- Finding an Euler cycle/path let's us draw the graph without lifting the pencil

Course plan

- 1 Minimum spanning tree
 - Kruskal's algorithm for MST
 - Prim's algorithm for MST
- 2 Euler path
- 3 Traveling salesman problem**
- 4 Stable marriage problem

Traveling salesman problem



Given a list of cities and roads connecting them, what is the shortest route that visits each city and ends at the starting point?

Traveling salesman problem

- This problem is known as traveling salesman problem
- It has many direct applications:
 - Planning a route for delivery companies
 - Efficiently moving a telescope to watch different objects on sky
 - Minimizing cost of soldering circuit-board
- It also appears as a subproblem in many areas, like DNA sequencing.
- In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *road* represents travelling times or cost, or a similarity measure between DNA fragments.

Naive solution

- Naive solution would be to calculate all the possible permutations and choose one with lowest possible length
- How many possibilities there exist for n cities? So first we need to choose one of n cities, then one of $n - 1$ cities, then $n - 2 \dots$
- In the end there are $n!$ solutions, so the complexity of this algorithm is also $O(n!)$
- This is very bad, $n!$ is very "quick" function, so even for relatively small number of cities there are just too many options for this algorithm to be feasible
- For 60 cities there are $60! \approx 8 * 10^{81}$ different possible routes
- There are about 10^{78} atoms in the universe

Dynamic programming

- What about applying dynamic programming approach?
- Every subpath of a path of minimum distance is itself of minimum distance - this is optimization property that we can use in dynamic programming
- Held-Karp algorithm:
 - 1 Pick a starting city c
 - 2 Calculate the simplest subproblem - getting from previous to last city to the last city c
 - 3 Then calculate more advanced problem of getting from the third to last city to the last city c
 - 4 Repeat n times
- Complexity of this algorithm: $O(n^2 \times 2^n)$
- Better than naive solution, but still very bad - in real world applications it will never be used

P vs NP

- In fact Traveling Salesman Problem belongs to a class of NP-Complete problems
- Problem belongs to NP-Complete class if it is at as "hard" as any other problem in NP-Complete and the solution can be verified in polynomial time
- There is one funny property - all the problems in NP-Complete class are closely related and we can translate one problem to other in the same class. So if we find a polynomial algorithm for just one of the problems then all problems in NP-Complete will be solvable in polynomial time
- Right now no polynomial-time algorithm has been found for any NP-Complete problem.
- It is still not proven that there is not an polynomial-time algorithm for NP-Complete class. This is the famous $P = NP$ problem, if you solve it you will earn 1 million dollars and eternal fame

P and NP problems

- NP problems are deceptively similar to P problems and on the surface it might seem that a polynomial algorithm should exist
- Some examples:
 - Searching **shortest** path between two vertices in a graph can be done in $O(VE)$ time. Searching for **longest** path requires non-polynomial time algorithm.
 - Looking for a path in graph that visits every **edge** once can be done in $O(E)$ (Euler path). Looking for a path that visits every **vertex** (Hamilton path) is however NP-Complete.
 - Graph coloring is applying a color to every vertex in such a way, that vertices of the same color are not connected. Checking if a graph can be colored using **two** colors is polynomial problem. Checking if graph can be colored using **three** (or more) colors belongs to NP-Complete class

Approximate algorithms for TSP

- Finding optimal solution for TSP is impractical, as was shown before
- Therefore an approximation solution must be found
- There are many algorithms for finding suboptimal paths for TSP, they differ in how complex they are, how fast they can provide a result and how close to optimal solution the result would be
- Most of them use some kind of heuristics, some try to take advantage of the properties of the system
- Amazingly humans can produce quite good solutions pretty fast - this lead to some studies in cognitive science regarding which exact heuristics human brain uses and could they be translated to computer algorithms

Random and Greedy algorithms for TSP

Random path for TSP

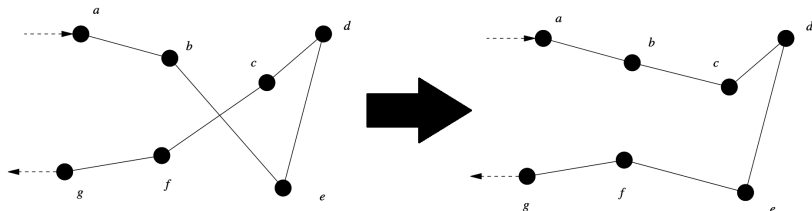
- This algorithm works by selecting a starting city and then choosing at random next city to visit from the set of unvisited cities
- Result is produced very fast, but the resulting path is for sure not optimal
- Still, this can be a benchmark for other algorithms

Greedy TSP

- This algorithm follows greedy algorithm principle - always make a decision that is the best at a given moment
- Translating it to TSP - start with a city and then travel to the closest unvisited city; repeat the process until all cities are visited
- This algorithm produces a path that is on average 25% longer than the optimal path, but the result is obtained very fast
- There are however cases in which this algorithm will always provide the worst path, but it is the case for any greedy algorithm

2-opt algorithm for TSP

- In this algorithm we iteratively improve our solution by making shortcuts
- For the case of 2-opt algorithm, if we find a path that crosses over itself we rearrange it so it does not
- We need to start with some initial path, it might be random or even result of other algorithm (like greedy). Then optimize using 2-opt until no more optimizations could be done
- Instead of switching only 2 edges, we can rearrange 3 edges at a time (3-opt) or more (K-opt algorithm). Generalization of this strategy is Lin-Kernighan heuristic, where at every step an algorithm decides how many edges should be reordered



Simulated annealing

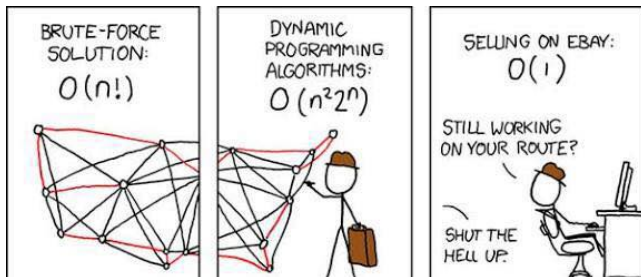
- Annealing process in metalurgy: heat metal to very high temperature and slowly cool it down. Thanks to this process internal structure of metal changes and also it's physical (or even chemical) properties.
- We can simulate this process in our computers, hence the name "simulated annealing"
- Simulated annealing for TSP:
 - 1 Start with some initial route
 - 2 Make a random change to the route
 - 3 If result route is shorter accept the change
 - 4 If not, still accept it with some probability correlated to the temperature of the system
 - 5 Decrease the temperature and go back to step 2
- At first our system is "hot", so it changes rapidly. With decreasing temperature probability of introducing suboptimal changes also decreases
- However we need to allow for those suboptimal changes to happen, else we will not escape the local minimum

Last words on TSP

Visualization

<https://www.youtube.com/watch?v=q6fPk0-eHY>

- While it might look impractical to wait hours or even days for a solution that is just 1% better, there are cases where even slight improvement leads to millions of dollars in savings
- Traveling salesman problem is one of the most popular questions during interviews - be prepared!













Course plan

- 1 Minimum spanning tree
 - Kruskal's algorithm for MST
 - Prim's algorithm for MST
- 2 Euler path
- 3 Traveling salesman problem
- 4 Stable marriage problem

Stable marriage problem

- Imagine two equal sized groups of men and women, where every person keeps a list of preferred partners from the second group
- Can we marry them in such a way, that there are no two people of opposite sex who would both rather have each other than their current partners?

Boys	Girls
 1: CBEAD	 A : 35214
 2 : ABECD	 B : 52143
 3 : DCBAE	 C : 43512
 4 : ACDBE	 D : 12345
 5 : ABDEC	 E : 23415

Stable marriage problem

- Stable marriage problem applications in many fields:
 - Matching users and servers on web
 - Devising a system for donor-recipient chain of organ transplants
 - Assigning employees to employers
 - Assigning actors/directors to movies
- There is an algorithm that provides a stable solution: Gale–Shapley algorithm.
- This algorithm iteratively upgrades it's result, until everyone is married:
 - Each unengaged man proposes to a woman that is the highest on his list of preferences to whom he did not yet proposed (even if that woman is already engaged)
 - Every woman chooses a partner from all the proposals that is the highest on her list of preferences
 - Process is repeated until every man and woman are engaged











Gale–Shapley pseudocode

```

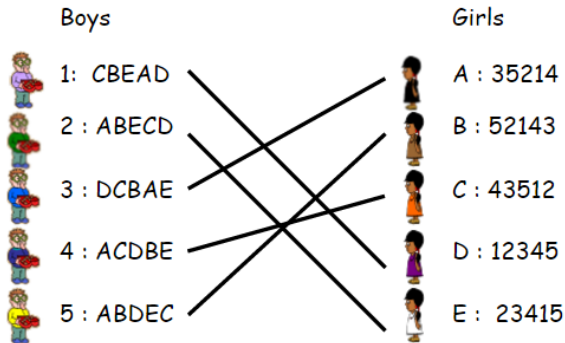
1 Initialize all men and women to free
2 while there exist a free man M who still has a woman to propose
   to
3 {
4     W = M's highest ranked woman to whom he has not yet proposed
5     if W is free
6         (M, W) become engaged
7     else some pair (M', W) already exists
8         if W prefers M to M'
9             (M, W) become engaged
10            M' becomes free
11        else
12            (M', W) remain engaged
13 }

```

Gale–Shapley example

Boys	Girls
	 A : 35214
	 B : 52143
	 C : 43512
	 D : 12345
	 E : 23415

Gale-Shapley example



Gale-Shapley analysis

- Algorithm runs in $O(n^2)$ time, where n is number of participants
- Algorithm guarantees stability of marriages - there are no two people that would at the same time prefer the other person over their current partner
- The solution is man-optimal - all men in the system cannot get better partner than the one calculated by the algorithm
- That's not the case for women, they might be paired down
- What happens if we allow cheating when describing preferences? Man earn nothing or even could lose if the cheat with their list of preferences. Women however can get better partner!
- Variations of this problem:
 - Stable roommate problem - we have one big group of people that we need to pair up, without dividing into men/women
 - Hospital/residents problem - a resident can choose one hospital, but one hospital can hire many residents
- Lloyd Shapley, Alvin E. Roth earn Nobel Prize in Economics in 2012 for applying this algorithm to Hospital/residents problem, resulting in potentially many lives saved!