6 Graph algorithms

6.1 Graphs and their representation

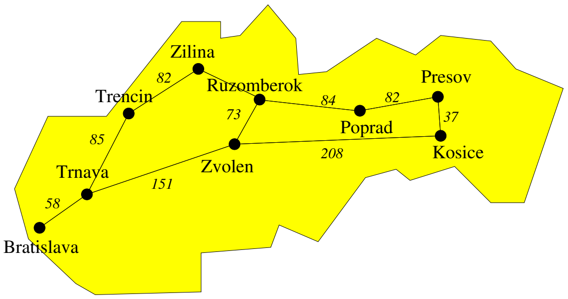
[BB 5.4-5.5 or CLRS2 B.4,22.1]

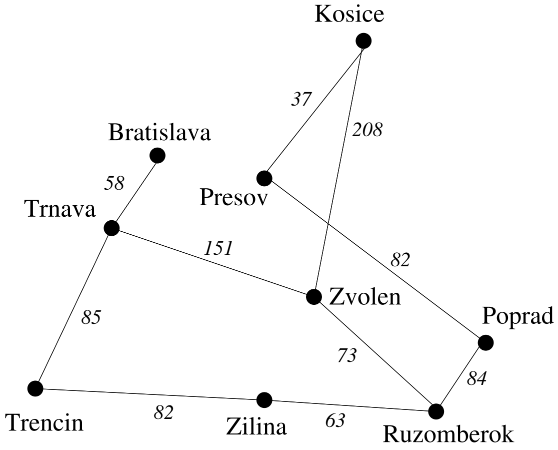
Basic definitions

- **Graph** G is a pair (V, E), where V is a finite set (set of vertices) and E is a finite set of pairs from V (set of edges). We will often denote $n := |V|$, $m := |E|$.
- Graph G can be **directed**, if E consists of ordered pairs, or **undirected**, if E consists of unordered pairs. If $(u, v) \in E$, then vertices u and v are **adjacent.**
- We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called **weighted**.
- **Degree** of a vertex v is the number of vertices u for which $(u, v) \in E$ or $(v, u) \in E$ (denote $deg(v)$). The number of **incoming edges** to a vertex v is called **in-degree** of the vertex (denote $indeg(v)$). The number of **outgoing edges** from a vertex is called **out-degree** (denote $outdeg(v)$).

Examples:

- Map of Slovakia with driving distances: the cities are vertices, highways between the cities are edges. We can assign weights to the edges based on, for example, driving distances.





Note that the position of the vertices in the drawing of the graph does not necessarily have any relation to the real geographical position of the cities (either relative or absolute).

This graph is weighted undirected.

Note: We only consider simple graphs – the ones which do not have loops (edges which begin and end in the same vertex) and multiple edges (several edges with the same end points).

Note:

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

$$0 \leq m \leq \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$$

$$\log m = O(\log n)$$

Representation of graphs

Consider graph $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_n\}$.

- **Adjacency matrix** represents the graph as an $n \times n$ matrix $A = (a_{i,j})$, where

  $$a_{i,j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

  The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

  We may consider various modifications. For example for weighted graphs, we may have

  $$a_{i,j} = \begin{cases} w(v_i, v_j), & \text{if } (v_i, v_j) \in E, \\ \textit{default}, & \text{otherwise,} \end{cases}$$

  where *default* is some sensible value based on the meaning of the weight function (for example, if weight function represents length, then *default* can be ∞, meaning value larger than any other value).

- **Adjacency lists** represent the graph by listing for each vertex v_i its outgoing vertices in a list $out(v_i)$. (Representation can be linked list, or another appropriate structure.)

  If the graph is directed, it makes sense to build for each vertex v_i also list of its incoming vertices $in(v_i)$.

Comparison of graph representations

	Adjacency matrix	Adjacency list
Is $(u, v) \in E$?	$\Theta(1)$	$\Theta(outdeg(u))$
List edges outgoing from u	$\Theta(n)$	$\Theta(outdeg(u))$
Memory	$\Theta(n^2)$	$\Theta(m + n)$

We will be using adjacency lists to represent graphs, unless stated otherwise.

Paths and cycles

A path is a sequence of vertices $(v_1, v_2, \dots, v_k)$, where for all i, $(v_i, v_{i+1}) \in E$. A **path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices $(v_1, v_2, \dots, v_k, v_{k+1} = v_1)$, where for all i, $(v_i, v_{i+1}) \in E$ and all vertices in the cycle are distinct except pair v_1, v_{k+1}.

Subgraphs and spanning trees

Subgraphs: A graph $G' = (V', E')$ is a subgraph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

Induced subgraphs: A subgraph $G' = (V', E')$ of graph $G = (V, E)$ is an **induced subgraph**, iff $E' = \{(u, v) \in E \,:\, u, v \in V'\}$.

The undirected graph G is connected, if for every pair of vertices u, v there exists a path from u to v.

If a graph is not connected, the vertices of the graph can be divided into **connected components**. Two vertices are in the same connected component iff they are connected by a path.

Tree is a connected acyclic graph. A **spanning tree** of a graph $G = (V, E)$ is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

Lemma 1. *Let T be a spanning tree of a graph G. Then*

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T, then T becomes disconnected.*
3. *If we add any edge into T, then the new graph will contain a cycle.*
4. *Number of edges in T is $n - 1$.*

(For proof, see [CLRS2, Theorem B.2])

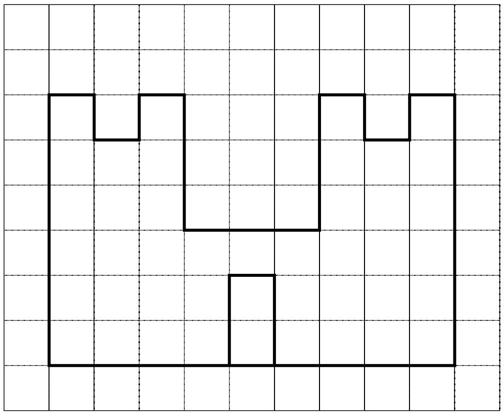
6.2 Exploring undirected graphs

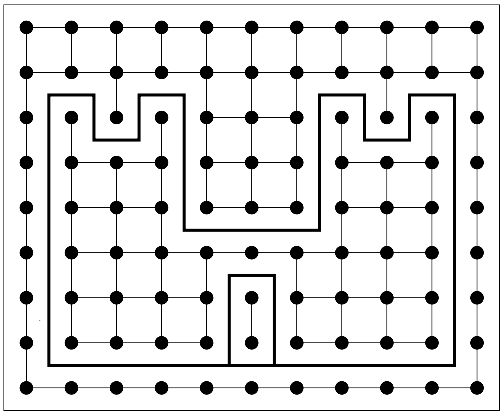
[BB 9.3-9.5 or CLRS2 22.2-22.3]

Problem: Given an undirected graph $G = (V, E)$, separate vertices into connected components. In particular, assign each vertex a number so that they have the same number iff they are in the same connected component.

Application: We have a picture drawn on a grid. The picture separates the grid into several areas. We want to fill every area by a different color.

Figure on the left presents example of the problem. Note, that the problem can be formulated as finding connected components in an undirected graph (see figure on the right).





Solution 1: Depth-first search Let us keep the following color scheme to record status of the vertices:

- "white" – vertices which we have not seen so far in our search
- "gray" – vertices which we have seen, but we are not finished dealing with them yet
- "black" – vertices which we have seen and we will not need to access their information any more.

```
function dfs-visit(v,cnum)
  // pre-condition: v is WHITE vertex
  // find all vertices that are reachable from v
  // by path going through white vertices only
  status[v]:=gray;
  num[v]:=cnum;
  for each w in out(v)
    if status[w]=white
       dfs-visit(w,cnum)
  status[v]:=black;

// --- main program ---
status of all vertices is white
cnum=0; // component number
for all vertices v in V
  if status[v]=white
    // all vertices in v's component are white;
    // explore v's component
    dfs-visit(v,cnum);
    cnum:=cnum+1;
```

Example: Simulate the algorithm on the "castle" pictures.

Running time:

- We call `dfs-visit` for each vertex $v \in V$ exactly once.
- If we ignore recursive calls, then `dfs-visit` for vertex v takes $\Theta(1) + \Theta(deg(v))$ time.
- Thus the total running time is $\Theta(n) + \Theta(\sum_{v \in V} deg(v)) = \boxed{\Theta(n+m)}$.

Other properties of DFS

Notation:

- **Discovery time.** We assign to every vertex a "timestamp" $d(v)$ when it changes color from white (unexplored) to gray (discovered).
- **Finishing time.** We give every vertex a "timestamp" $f(v)$ when it changed its color from gray (discovered) to black (finished).
- **Tree edges.** When we discover vertex w by calling `dfs-visit` from vertex v, we mark edge (v, w) a **tree edge**. We will call v the **parent** of w. All other edges are **back edges**.

```
function dfs-visit(v,cnum)
  status[v]:=gray;
* time:=time+1; d[u]:=time;
  num[v]:=cnum;
  for each w in out(v)
    if status[w]=white
*      edge (v,w) is a tree edge;
       dfs-visit(w,cnum)
  status[v]:=black;
* time:=time+1; f[u]:=time;
```

Example: Example of a DFS tree edges and discovery and finishing times on a simple graph.

Note: If graph G is connected, then tree edges form a spanning tree of G (called **DFS tree**). Otherwise the tree edges form a spanning tree for every component of G.

Lemma 2. *Consider a DFS tree T. Let $e = (u, v)$ be a* **back edge***; without loss of generality assume $d(u) < d(v)$. Then u is ancestor of v in T.*

Proof. At time $d(u)$, the vertex u becomes gray, while v is still white. At time $f(u)$, the vertex u becomes black. At this point, vertex v must be either gray or black. This is because edge e connects u and v and if v was still white, we could not have turned u black. Thus $d(u) < d(v) < f(u)$.

However, all vertices which we discover between times $d(u)$ and $f(u)$ will become descendants of u in DFS tree T. Thus vertex v must be descendant of u. □

Corollary 1. *Consider a DFS tree T. Let u and v are two vertices which are not descendants of each other in T. Then there is no edge between descendants of u and descendants of v.*

Proof. Let u' be a descendant of u and v' be a descendant of v. Without loss of generality, let $d(u') < d(v')$.

Assume there exists an edge $e = (u', v') \in E$. If e is a tree edge, then v' is a child of u'. If e is a back edge, then according to Lemma 2, v' must be a descendant of u'.

Now consider the tree path from root of T through u and u' to vertex v' and the tree path from root of T through v to v'. These cannot be the same path, because u and v cannot lie on the same path originating in root. Thus there exist two tree paths from root to v' which is a contradiction with T being a tree. □

Finding articulations

[BB 9.3.1 or CLRS2 Problem 22-2]

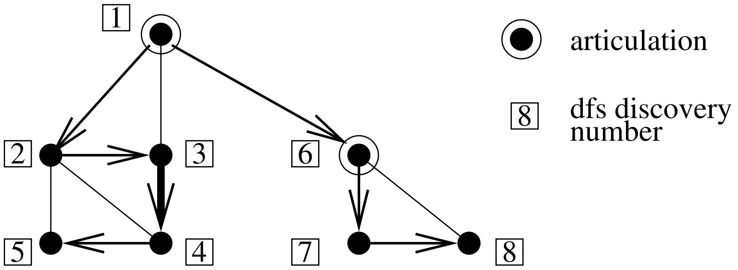
Definition 1. *A node v of a connected graph G is an* articulation point *iff by removal of v (and all its edges) G becomes disconnected.*

Motivation:

- important nodes in the network
- traffic points which, if blocked, can stop traffic between parts of a city

Problem: Given graph $G = (V, E)$, find the articulation points.

Example:



Trivial approach: for all vertices $v \in V$:

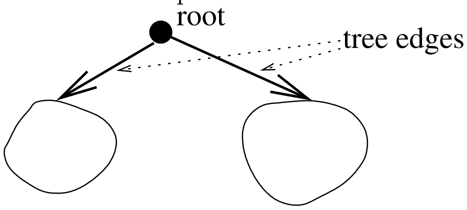
- remove v
- test connectivity with DFS

Running time: $\Theta(mn)$

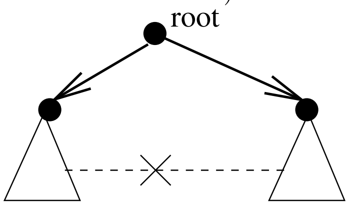
Can we do better?

Lemma 3. *Root of the DFS tree is an articulation point iff it has at least two children.*

Proof. (⇒) Assume that the root is an articulation. Therefore if we remove the root, we get at least two connected components. The only way how the DFS tree can span all the vertices is for root to have a child in each of the components.



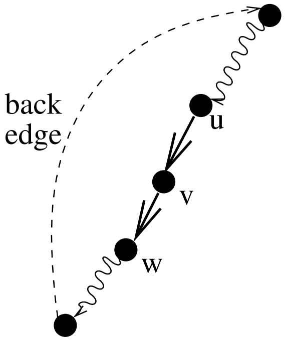
(⇐) If we remove the root, its children become disconnected (there are no edges between their descendants because of DFS lemma). Therefore if the root has at least two children then the root must be an articulation.



□

Observation: What can help us to connect descendants of v to ancestors of v if we remove v? **"Detour"**

Definition 2. *Let w be a child of v.* Detour *is a path starting in w, following several (possibly 0) tree edges "down the tree" followed by a back edge to an ancestor of v.* Detour number *of vertex w, $detour(w)$, is the smallest discovery number of a vertex to which we can get from w by a detour.*

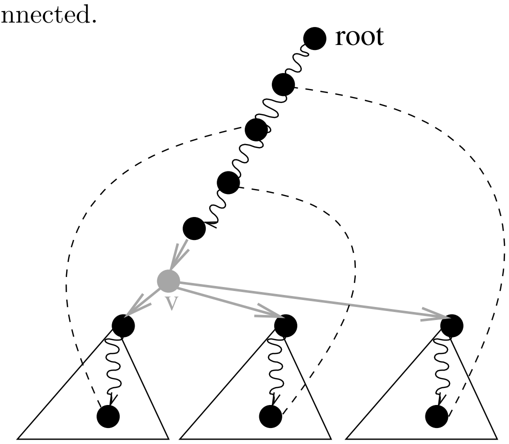


Lemma 4. *A non-root vertex v is NOT an articulation iff for every child w of v*

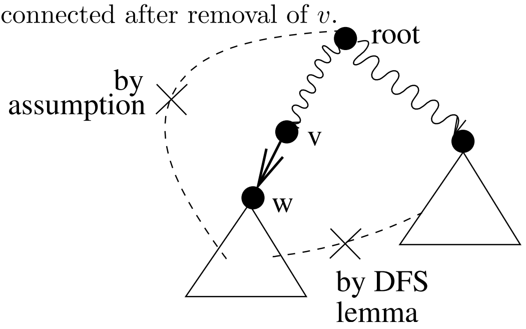
$$detour(w) < d(v).$$

Here $d(v)$ means DFS discovery time. If this inequality is satisfied for a child w we say that w has a detour.

Proof. (⇐) Assume every child of v has a detour. Remove vertex v. All vertices in all subtrees of v stay connected.



(⇒) Assume there is a child w without a detour. It means no vertex in its subtree has a back edge going "above" vertex v. By the properties of DFS we cannot get from w to the root. Therefore graph will become disconnected after removal of v.



□

Algorithm:

- Use DFS,
- add computation of detour number,
- add test for the condition from Lemmas 4 and 3

The condition for the root vertex needs to be handled separately in the code (not shown).

```
// explore vertex v
// compute detour[v]
// return number of children of v in DFS tree
function dfs-visit(v,cnum)
  status[v]:=gray;
  time:=time+1; d[v]:=time;
* detour[v]:=d[v]; number_of_children:=0;
  for each w in out(v)
    if status[w]=white
       number_of_children:=number_of_children+1;
       //--- (v,w) is a TREE edge
       parent[w]:=v;
       dfs-visit(w,cnum); // detour[w] is now computed!
*      if detour[w]>=d[v] and v<>root then
*        vertex v is an articulation!
*      if detour[w]<detour[v] then detour[v]:=detour[w];
*   else if w<>parent[v] then
*      //--- (v,w) is a BACK edge
*      if d[w]<detour[v] then detour[v]:=d[w];

  status[v]:=black;
  return number_of_children;

// --- main program ---
// Assumption: graph G=(V,E) is connected
status of all vertices is white
root:=any vertex from V
parent[root]:=undefined;
if (dfs-visit(root)>1) then
  vertex root is an articulation!
```

Running time: $\Theta(n+m)$

Solution 2: Breadth-first search

Recall: Problem of graph exploration.

Given an undirected graph $G = (V, E)$, separate vertices into connected components. In particular, assign each vertex a number so that they have the same number iff they are in the same connected component.

Depth-first search: When DFS arrives at some node, it tries to visit a neighbour, then a neighbour of the neighbour, ...

Different approach – **Breadth-first search:** Explore vertices in order of increasing distance from initial vertex.

- first all vertices in distance 1
- then all vertices in distance 2
- ...

Let $dist[u]$ be distance of u from the initial vertex (i.e., the length of the shortest path measured in the number of edges by which we can get from the initial vertex to u).

```
function bfs-visit(v,cnum)
  create empty queue Q;
  status[v]:=gray;
  dist[v]:=0;
  enqueue(Q,v);

  while Q is not empty
    u:=dequeue(Q);
    num[u]:=cnum;
    for each w in out(u)
      if status[w]=white then
        status[w]:=gray;
        dist[w]:=dist[u]+1;
        enqueue(Q,w);
    status[u]:=black;

Main program the same as for DFS
```

Note: Vertices in Q are always stored and processed in order of increasing distance from vertex v.

Example: Simulate the algorithm on the “castle” picture and compare with the DFS picture.

Running time: For each vertex u in the graph we dequeue it and check all its neighbours in $\Theta(1 + deg(u))$ time. Therefore the running time is

$$\Theta(\sum_{v \in V} 1 + deg(v)) = \Theta(n + \sum_{v \in V} deg(v)) = \boxed{\Theta(n+m)}.$$

Useful application: Compute the shortest path from vertex u to vertex v in unweighted graph.

6.3 Shortest paths

[BB 8.5 or CLRS2 25.2,24.3]

BFS could compute the length of the shortest path between two vertices if the length was measured in the number of edges. This is a special case of the shortest path problem.

Problem: Given a weighted directed or undirected graph $G = (V, E)$, all weights are non-negative. Compute the shortest path from u to v.

Lemma 5. *Let P be the shortest path from u to v. Then any part of P must also be the shortest path between its endpoints.*

Proof. If there was a subpath of P from u' to v' which was not the shortest path from u' to v', then we could replace this subpath by the shortest path from u' to v', obtaining a shorter path overall. □

This property suggests possibility of using dynamic programming.

Floyd-Warshall algorithm

Assumptions:

- Vertex set of the graph is $V = \{1, 2, \ldots, n\}$ (i.e., all vertices are numbered from 1 to n).
- The graph is represented by an adjacency matrix w, where $w(i, j) = \infty$, if $(i, j) \notin E$.

Subproblem: Let $dist[i, j, k]$ be the length of the shortest path from vertex i to vertex j which is allowed to pass only through vertices $1, 2, \ldots, k-1$ (or ∞ if such path does not exists). Solution is found in $dist[u, v, n]$.

Recurrence: For the shortest path $dist[i, j, k]$, we have two options:

- The path passes through vertex k. Then its length must be $dist[i, k, k-1] + dist[k, j, k-1]$.
- The path does not pass through vertex k. Then its length is $dist[i, j, k-1]$.

$$dist[i, j, k] = \min\{dist[i, j, k-1], dist[i, k, k-1] + dist[k, j, k-1]\}$$

Base cases: $dist[i, j, 0] = w(i, j)$.

Order of computation: from smallest k to largest.

```
// initialization
for i:=1 to n do
  for j:=1 to n do
    dist[i,j,0]:=w[i,j];

// main computation
for k:=1 to n do
  for i:=1 to n do
    for j:=1 to n do
      dist[i,j,k]:=min{dist[i,j,k-1],
                       dist[i,k,k-1]+dist[k,j,k-1]}
```

Simplification: Note that $dist[i,k,k-1] = dist[i,k,k]$ and $dist[k,j,k-1] = dist[k,j,k]$.

We do not need the third parameter in the matrix (reducing needed memory from $\Theta(n^3)$ to $\Theta(n^2)$).

```
// initialization
for i:=1 to n do
  for j:=1 to n do
    dist[i,j]:=w[i,j];

// main computation
for k:=1 to n do
  for i:=1 to n do
    for j:=1 to n do
      if dist[i,k]+dist[k,j]<dist[i,j] then
        dist[i,j]:=dist[i,k]+dist[k,j];
```

This program is very simple, but notice that the order of the loops is important.

How to recover the shortest path. So far we have only the length of the shortest path. We could use the typical "solution recovery" approach from dynamic programming. However, there is an easier way. Always remember the **second** vertex of the shortest path found so far. Due to Lemma 5, the path from second vertex to the last must also be the shortest path.

```
// initialization
for i:=1 to n do
  for j:=1 to n do
*   dist[i,j]:=w[i,j]; next[i,j]:=j;
// main computation
for k:=1 to n do
  for i:=1 to n do
    for j:=1 to n do
      if dist[i,k]+dist[k,j]<dist[i,j] then
        dist[i,j]:=dist[i,k]+dist[k,j];
*       next[i,j]:=next[i,k];
// find the shortest path from u to v
w:=u; write w;
while w<>v do
  w:=next[w,v]; write w;
```

Running time: $\Theta(n^3)$.

What if we need distances between several pairs of vertices? No additional computation is required. Floyd-Warshall algorithm computes the length of the shortest path between **all** pairs of vertices simultaneously (ALL-ALL shortest paths).

Dijkstra's algorithm Another algorithm for solving the shortest path problem, similar to BFS.

Maintain two sets:

- S – the set of "finished" vertices (for them we already know the shortest path from vertex u)
- T – the set of "unfinished" vertices

We will gradually "grow" set S. We start from a single vertex u and in each step we add one more vertex s for which we can be sure that the path found so far is the shortest path from u to s.

Notation: Let $dist[s]$ be the length of the shortest path from u to s going only through vertices in S (or ∞ if such path does not exists).

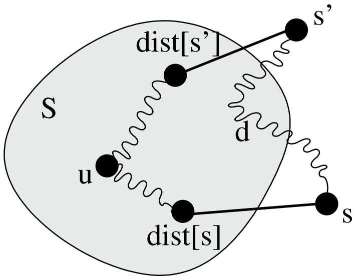
```
function shortest_paths(u)
// initialize dist, S, T
S:=0; T:=V;
for all w in V do dist[w]:=infinity;
dist[u]:=0;

// add one vertex at a time to S
while T is non-empty do
**s:=vertex for which dist[s] represents the length
**   of the shortest path from u to s;
  add s to S; remove s from T;
  // update dist to account for enlarged set S
  for all t in out(s) do
    // try to shorten current path to t through s
    if (dist[s]+w[s,t]<dist[t]) then
      dist[t]:=dist[s]+w[s,t];
```

How to choose vertex s in **?

Lemma 6. *Let vertex $s \in T$ has the smallest $dist[s]$ among all vertices from T. Then $dist[s]$ is the length of the shortest path u to s.*

Proof. Assume that the shortest path P from u to s has length less than $dist[s]$. There must be a vertex on path P which is not in S. Let s' be the first such vertex. By Lemma 5 the part of P from u to s' is the shortest path to s' and by the choice of s' it passes only through vertices in S. Therefore the length of this part of P is already computed in $dist[s']$.



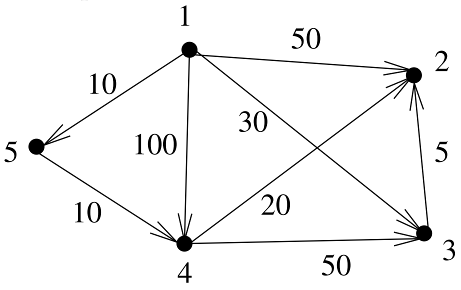
From our assumption $length(P) = dist[s'] + d < dist[s]$. Since $d \geq 0$, we have $dist[s'] < dist[s]$. But then our algorithm would have chosen s' instead of s, which is contradiction. □

```
// initialize dist, S, T
S:=0; T:=V;
for all w in V do dist[w]:=infinity;
dist[u]:=0;

// add one vertex at a time to S
while T is non-empty do
  s:=vertex with the smallest dist[s];
  add s to S; remove s from T;
  // update dist to account for enlarged set S
  for all t in out(s) do
    // try to shorten current path to t through s
    if (dist[s]+w[s,t]<dist[t]) then
      dist[t]:=dist[s]+w[s,t];
```

Note: Since we choose s with the smallest value of $dist[s]$ in each step, some people consider this to be a greedy algorithm (although it is not a typical one).

Example:



1	2	3	4	5
$\boxed{0}$	∞	∞	∞	∞
	50	30	100	$\boxed{10}$
	50	30	$\boxed{20}$	
	40	$\boxed{30}$		
	$\boxed{35}$			

How do we reconstruct the shortest path? Similar trick as in Floyd-Warshall algorithm: keep last but one vertex in the shortest path.

```
// initialize dist, S, T
S:=0; T:=V;
for all w in V do
* dist[w]:=infinity; last[w]:=undefined;
dist[u]:=0;

// add one vertex at a time to S
while T is non-empty do
  s:=vertex with the smallest dist[s];
  add s to S; remove s from T;
  // update dist to account for enlarged set S
  for all t in out(s) do
    // try to shorten current path to t through s
    if (dist[s]+w[s,t]<dist[t]) then
*     dist[t]:=dist[s]+w[s,t]; last[t]:=s;

// path reconstruction from u to v
w:=v; create an empty path;
while last[w]<>undefined do
  add w to the beginning of the path;
  w:=last[w];
```

Running time: Depends on implementation of data structure for $dist$.

• Build a structure with n elements	A
• at most m times decrease the value of an item	mB
• n times select the smallest value	nC

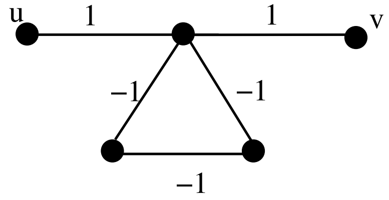
- For $\boxed{\text{array}}$ $A = O(n), B = O(1), C = O(n)$ which gives $\boxed{O(n^2)}$ total.
- For $\boxed{\text{heap}}$ $A = O(n), B = O(\log n), C = O(\log n)$ which gives $\boxed{O(n + m \log n)}$ total. This is better for sparse graphs ($m < n^2/\log n$).

- For $\boxed{\text{Fibonacci heap}}$ (amortized analysis - see cs466) $A = O(n)$, $B = O(1)$, $C = O(\log n)$ which gives $\boxed{O(m + n \log n)}$ total.

Note: Dijkstra's algorithm computes all shortest paths from a single vertex at once (single source shortest paths).

Discussion on negative edges So far we assumed that all edges have non-negative weights. Let us try to drop this assumption.

What is the shortest path from u to v in the following graph?



We can talk about shortest **simple** paths, but that is a different (in general much harder) problem.

Notes on algorithms we have encountered:

- Dijkstra's algorithm does not work for negative edges at all. The assumption was essential part of the proof.
- Floyd-Warshall algorithm works for graphs with negative edges if there is no cycle of negative length (in such case, the shortest path = the shortest simple path).

6.4 Minimum spanning trees

[BB 6.3 or CLRS2 23.2]

Weight of a spanning tree $w(T)$ is the sum of weights of all edges in T.

Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

Problem: Given is a connected undirected weighted graph $G = (V, E)$. Find a minimum spanning tree of the graph G.

Motivation: We want to connect several points by a computer network. We want to minimize the total cost.

Assumption: $V = \{1, 2, \ldots, n\}$, $E = \{f_1, f_2, \ldots, f_m\}$

Kruskal's algorithm

```
Sort edges in order of increasing weight
so that w[f[1]] <= w[f[2]] <= ... <=w[f[m]]

T:=empty set
for i:=1 to m do
  let u,v be the endpoints of edge f[i]
  if there is no path between u and v in T then (**)
```

```
    add f[i] to T
return T
```

Proof of correctness

Lemma 7. *Let greedy solution T_G computed by the Kruskal algorithm contains edges $e_1, e_2, \ldots, e_{n-1}$ (numbered in order of increasing weight). Then for any $0 \leq k \leq n-1$ there exists a minimum spanning tree that contains edges $e_1, e_2, \ldots, e_k$.*

Proof. By induction on k.

Base case: For $k = 0$ the lemma trivially holds.

Induction step: Suppose there exists a minimum spanning tree T^* containing edges $e_1, e_2, \ldots, e_{k-1}$.

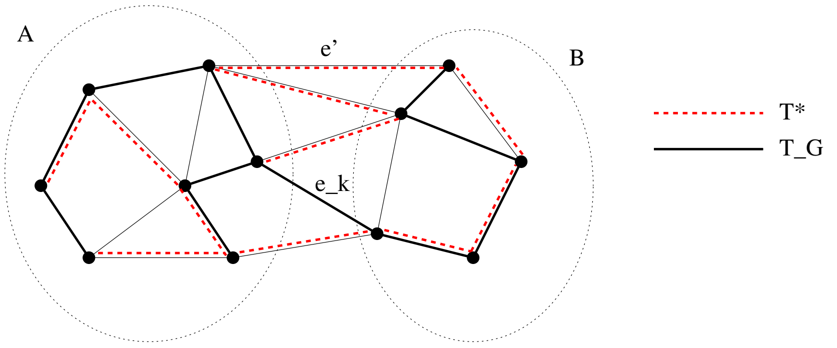
- **Case 1:** $e_k \in T^*$. Then T^* contains all of the edges $e_1, e_2, \ldots, e_k$ and the statement is true.
- **Case 2:** $e_k \notin T^*$.
  - If we remove edge e_k from T_G, T_G becomes disconnected and will have 2 components A and B.
  - Now let us add e_k into T^*. This will create a cycle in T^*. The cycle involves vertices in both A and B, therefore the cycle must contain an edge $e' \neq e_k$ that has one endpoint in A and one in B. Remove edge e' and denote the new graph T' (i.e. $T' = T^* \cup \{e_k\} \setminus \{e'\}$). Obviously, T' is a spanning tree.
  - Note that $w(e') \geq w(e_k)$. Otherwise e' would have been chosen by the Kruskal's algorithm instead of e_k.
  - The cost of T' can be written as

$$\begin{aligned} w(T') &= w(T^*) + \underbrace{w(e_k) - w(e')}_{\leq 0} \\ w(T') &\leq w(T^*) \end{aligned}$$

  Since T^* is a minimum spanning tree, $w(T') = w(T^*)$ and T' is also a MST. Moreover, T' contains all edges $e_1, e_2, \ldots, e_k$ which is what we wanted to prove.

Thus we have proved by induction that for every k there exists a MST that contains each of the edges $e_1, \ldots, e_k$.

□



Running time: Sorting takes $\Theta(m \log m) = \Theta(m \log n)$. The rest depends on the implementation of query (**).

- $\boxed{\text{Run DFS}}$ on the edges of T selected so far. There are less then n of them, so it will take $O(n)$ per query. This means the running time is $\boxed{O(mn)}$.
- $\boxed{\text{Union/find-set data structure}}$ (introduced in cs466) takes $O(\log n)$ (or even better) per query. This means the running time is $\boxed{O(m \log n)}$.

$\boxed{\textbf{Prim's algorithm}}$ The main idea is to start from an arbitrary single vertex s and gradually "grow" a tree. We maintain a set of "connected" vertices S.

```
S := {s};
T := empty set;
while S<>V do
  e := (u,v) such that u is in S, v is not in S and     (*)
       w(e) is smallest possible;
  add v to S;
  add e to T;
return T;
```

Proof of Prim's algorithm

Lemma 8. *Let the greedy Prim's algorithm give solution T_G containing edges $e_1, e_2, \ldots, e_{n-1}$ (in order as they were added by the algorithm). Then for any $0 \leq k \leq n-1$ there exists a MST containing edges $e_1, e_2, \ldots, e_k$.*

Proof. By induction on k.

Base case. For $k = 0$ lemma trivially holds.

Induction step. Assume that there exists a MST T^* containing all edges $e_1, e_2, \ldots, e_{k-1}$.

- **Case 1:** $e_k \in T^*$. Then T^* trivially satisfies the lemma for k as well.
- **Case 2:** $e_k \notin T^*$. Let S be the set of finished vertices after $k-1$ steps of the algorithm.
  - Add e_k to T^*. This will create a cycle in T^*. The cycle must contain an edge $e' \neq e_k$ with one endpoint in S and one not in S. Remove edge e' and denote the new graph T' (i.e. $T' = T^* \cup \{e_k\} \setminus \{e'\}$).
  - Obviously, T' is a spanning tree.
  - Note that $w(e') \geq w(e_k)$. Otherwise e' would have been chosen by the Prim's algorithm instead of e_k.
  - The cost of T' is
    $$w(T') = w(T^*) + w(e_k) - w(e') \leq w(T^*)$$
    Therefore T' is a MST containing all edges $e_1, e_2, \ldots, e_k$ which is what we wanted to prove.

Thus we have proved by induction that for every k there exists a MST that contains each of the edges $e_1, \ldots, e_k$.

□

Running time We make algorithm more efficient by keeping for each vertex not in S its closest neighbour in S. The distance to this neighbour will be stored in $dist[v]$ and the neightbour itself in $other[v]$.

```
S := {s};
T := empty set;
// initialize data structure
for each u not in S
  dist[u] := w(s,u);
  other[u] := s;
// main computation
while S<>V do
  v := vertex which is not in S and has the smallest dist[v];
  e := (v, other[v]);
  add v to S;
  add e to T;
  // update data structure
  for each x not in S
    if w(v,x)<dist[x] then
      dist[x] := w(v,x);
      other[x] := v;
return T;
```

We do the same set of operations with *dist* as in Dijkstra's algorithm (initialize structure, m times decrease value, $n-1$ times select minimum). Therefore we get $O(n^2)$ time when we implement *dist* with array, $O(n + m\log n)$ when we implement it with a heap and $O(m + n\log n)$ when we implement it with Fibonacci heaps.

6.5 Formulating problems as graph problems

Reliable network routing

Problem: There is a computer network with many links. Every link has assigned reliability (probability between 0 and 1 that the link will operate correctly).

We want to choose a route between nodes u and v with highest reliability. Reliability of a route is a product of reliabilities of all its links.

Solution:

- The route obviously corresponds to a path in the graph
- The higher the probability p, the smaller $-\log p$. Also $-\log p$ is non-negative for $p \leq 1$.
- $-\log(p_1 \cdot p_2) = (-\log p_1) + (-\log p_2)$
- Take the network in which each edge is weighted by $-\log p$ where p is its reliability
- Find the shortest part from u to v (e.g. by Dijkstra's algorithm)

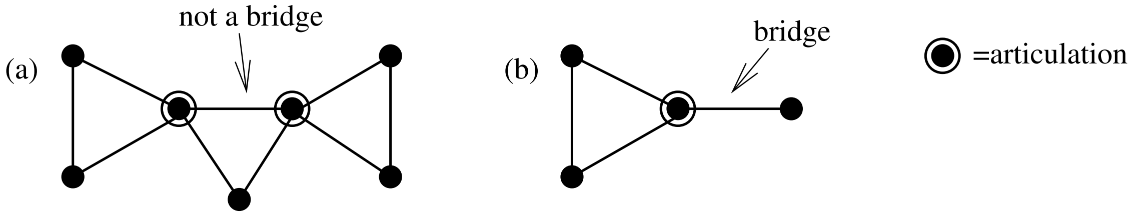
Bridges in a graph

Problem: A network consists of n nodes connected by links. We want to identify critical network links, which are links whose malfunction would make communication between some pairs of nodes impossible.

Straightforward graph formulation: Represent nodes as vertices, links as edges. The goal is to identify edges whose removal disconnects the graph. In graph terminology edges whose removal disconnectes graphs are called *bridges.*

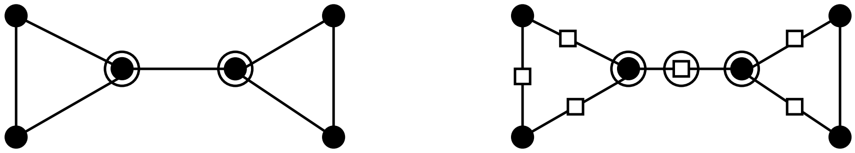
This tasks seems to be similar to finding articulations, but how do we proceed?

Naive approach: bridge is an edge with articulations on both sides. Does not work – see the following examples:



Different graph formulation:

- Create a vertex for every node AND every link of the network.
- Connect a link-vertex to a node-vertex if the link has an endpoint in the node.
- A link is critical (or a bridge) iff corresponding link-vertex is an articulation.



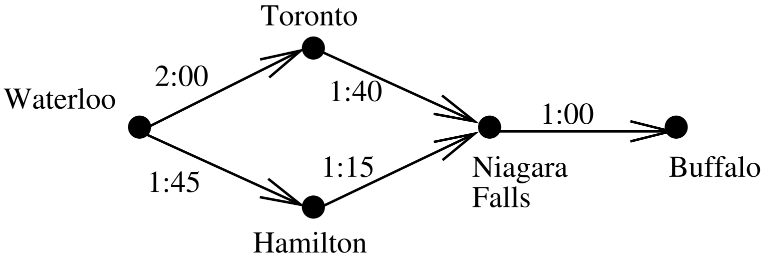
Bus schedules

Question: What is the fastest way to get by bus from Waterloo to Buffalo?

Example:

Waterloo	15:40	Waterloo	9:00	17:00	
Hamilton	17:25	Toronto	11:00	19:00	
Hamilton	17:30	Toronto	12:30	20:30	
Niagara Falls	18:45	Niagara Falls	14:05	22:10	
Niagara Falls	14:10	18:40	22:55		
Buffalo	15:25	19:40	23:59		

Straightforward graph formulation:



This looks like shortest path problem. This graph suggests the best route is through Hamilton – 4 hours. But we do not take waiting times into account. Also, different busses on the same route take different times.

Different graph formulation: Each bus is characterized by four elements (from, to, departure, arrival).

- Create a vertex for each bus
- Edge between busses a and b iff

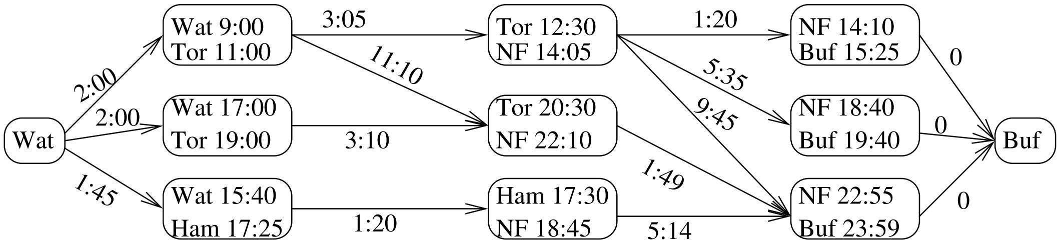
$$a.to = b.from \text{ and } b.departure \geq a.arrival$$

- Length of the edge (a, b):

$$\underbrace{(b.departure - a.arrival)}_{\text{waiting time}} + \underbrace{(b.arrival - b.departure)}_{\text{travel on bus } b} = b.arrival - a.arrival$$

- Create two special vertices for "origin" and "destination"
- Edge $(origin, a)$ iff $a.from = origin$. Length of the edge is $a.arrival - a.departure$.
- Edge $(b, destination)$ iff $b.to = destination$. Length of the edge is 0.

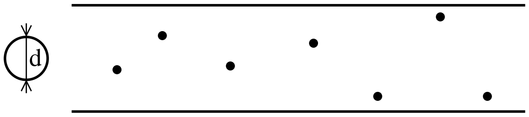
Now it is the shortest path problem!



We get that the shortest way is through Toronto 6:25 hours, which is depressing.

Cylinder in a forest

Problem: We have a forest with northern and southern fence. For simplicity, trees will be represented by points and fences by straight lines. A cylindrical object (the "fat man") with diameter d stands at the western boundary of the forest. Can the object pass through the forest to its east boundary?



Graph formulation: This does not sound like a typical graph problem.

- Create a vertex for each tree, and for both fences.
- Two trees are connect by an edge iff "fat man" cannot pass between them (their distance is smaller than d).
- Similarly for edges between a fence and a tree.

If the two fences are connected by a path in this graph of this graph, the "fat man" cannot pass (there is a "wall" from trees going all the way from the northern fence to the southern fence).

Otherwise boundary of the connected component containing the northern fence defines a viable path.

Conclusion:

- Graphs are a very important formalism in the computer science.
- Efficient algorithms are available for many important problems:
  - exploration
  - shortest paths
  - minimum spanning trees
  - and others
- If we formulate a problem as a graph problem, chances are that an efficient non-trivial algorithm for solving the problem is known.
- Some problems have natural graph formulation. For others we need to choose a less intuitive graph formulation. Some problems that do not seem to be graph problems at all can be formulated as such.