Datenstrukturen und Algorithmen

Exercise 10

FS 2018

Program of today

1. Feedback of last exercise
2. Repetition theory
3. Programming Task

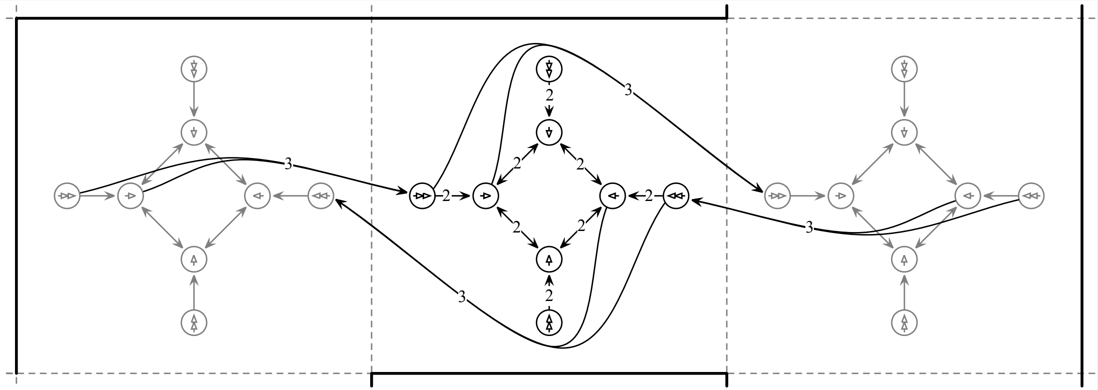
1. Feedback of last exercise

Exercise 9.1: Labyrinth

- Robot has to stop to change direction
- Interpret as shortest path problem

Exercise 9.1: Labyrinth

- position $\times$ direction $\times$ speed



- Runtime?

Exercise 9.1: Labyrinth

- Let n be the number of squares. Graph has $|V| = 8n$ nodes
- Graph has at $|E| \leq 20n$ edges
- Therefore, Dijkstra $\mathcal{O}(|E| + |V|log|V|)$ has runtime $\mathcal{O}(nlogn)$

Closeness Centrality

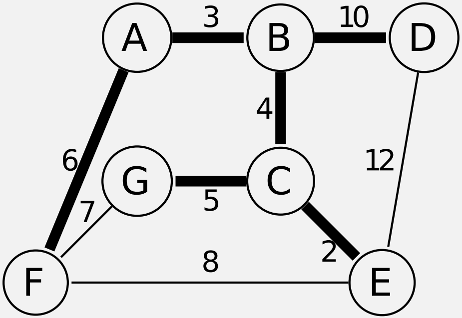
- Given: an adjacency matrix for an *undirected* graph on n vertices.
- Output: the *closeness centrality* $C(v)$ of every vertex v.

$$C(v) = \sum_{u \in V \setminus \{v\}} d(v, u)$$

- Intuition: If many connected vertices are close to v, then $C(v)$ is small.
- "How central is the vertex in its connected component?"

Minimum Spanning Tree

Kruskal computes the following MST:



Minimum Spanning Tree

Proof using induction over the number of vertices $|V|$:

- Hypothesis: undirected graph with $|V| - 1$ vertices has as most $|V| - 2$ edges.
- Induction base ($|V| = 1$): A graph with one node has no edges.
- Induction step ($|V| - 1 \rightarrow |V|$) Undirected cycle-free graph $G = (V, E)$. cycle-free $\Rightarrow$ there is at least one vertex w with degree 0 or 1. Let $V' = V \setminus \{w\}$ and $E' = \{\{u, v\} \in E | u, v \in V'\}$. Because there is at most one edge incident to w, $|E'| \geq |E| - 1$. Due to the induction hypothesis $|E'| \leq |V'| - 1$, we get

$$|E| \leq |E'| + 1 \leq |V'| - 1 + 1 = |V'| = |V| - 1$$

All Pairs Shortest Paths

```
template<typename Matrix>
void allPairsShortestPaths(unsigned n, Matrix& m){
  for(unsigned k = 0; k < n; ++k) {
    for(unsigned i = 0; i < n; ++i) {
      for(unsigned j = i + 1; j < n; ++j) {
        if(k == i || k == j)
          continue;
        if(m[i][k] == 0 || m[k][j] == 0)
          continue; // no connection via k
        if(m[i][j] == 0 || m[i][k] + m[k][j] < m[i][j])
          m[i][j] = m[j][i] = m[i][k] + m[k][j];
      }
    }
  }
}
```

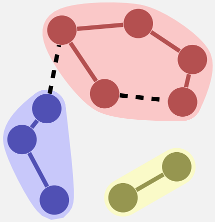
Closeness Centrality

```
vector<vector<unsigned> > adjacencies(n,vector<unsigned>(n, 0));
vector<string> names(n);
// ...
allPairsShortestPaths(n, adjacencies);
for(unsigned i = 0; i < n; ++i) {
  cout << names[i] << ": "; unsigned centrality = 0;
  for(unsigned j = 0; j < n; ++j) {
    if(j == i) continue;
    centrality += adjacencies[i][j];
  }
  cout << centrality << endl;
}
```

2. Repetition theory

Implementation Issues

Consider a set of sets $i \equiv A_i \subset V$. To identify cuts and circles: membership of the both ends of an edge to sets?



Union-Find Algorithm MST-Kruskal(G)

Input : Weighted Graph $G = (V, E, c)$
Output : Minimum spanning tree with edges A.

Sort edges by weight $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ **to** m **do**
    MakeSet(k)

for $k = 1$ **to** m **do**
    $(u, v) \leftarrow e_k$
    if Find(u) $\neq$ Find(v) **then**
        Union(Find(u), Find(v))
        $A \leftarrow A \cup e_k$

return (V, A, c)

Implementation Union-Find

Index	1	2	3	4	5	6	7	8	9	10
Parent	**1**	1	1	6	**5**	**6**	5	5	3	**10**

Operations:

- Make-Set(i): $p[i] \leftarrow i$; **return** i
- Find(i): **while** $(p[i] \neq i)$ **do** $i \leftarrow p[i]$; **return** i
- Union(i, j): $p[j] \leftarrow i$; **return** i

Optimization of the runtime for Find

Tree may degenerate. Example: Union$(1,2)$, Union$(2,3)$, Union$(3,4)$, ...
Idea: always append smaller tree to larger tree. Additionally required: size information g

Operations:

- Make-Set(i): $p[i] \leftarrow i$; $g[i] \leftarrow 1$; **return** i
- Union(i,j):
  if $g[j] > g[i]$ **then** swap(i,j)
  $p[j] \leftarrow i$
  $g[i] \leftarrow g[i] + g[j]$
  return i

Further improvement

Link all nodes to the root when Find is called.

Find(i):

$j \leftarrow i$
while $(p[i] \neq i)$ **do** $i \leftarrow p[i]$
while $(j \neq i)$ **do**
    $t \leftarrow j$
    $j \leftarrow p[j]$
    $p[t] \leftarrow i$
return i

Amortised cost: amortised *nearly* constant (inverse of the Ackermann-function).

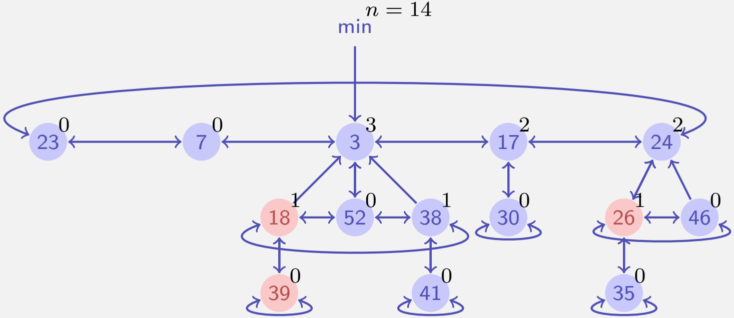
Fibonacci Heaps

Data structure for elements with key with operations

- MakeHeap(): Return new heap without elements
- Insert(H, x): Add x to H
- Minimum(H): return a pointer to element m with minimal key
- ExtractMin(H): return and remove (from H) pointer to the element m
- Union(H_1, H_2): return a heap merged from H_1 and H_2
- DecreaseKey(H, x, k): decrease the key of x in H to k
- Delete (H, x): remove element x from H

Implementation

Doubly linked lists of nodes with a marked-flag and number of children. Pointer to minimal Element and number nodes.



Simple Operations

- MakeHeap (trivial)
- Minimum (trivial)
- Insert(H, e)
  1. Insert new element into root-list
  2. If key is smaller than minimum, reset min-pointer.
- Union (H_1, H_2)
  1. Concatenate root-lists of H_1 and H_2
  2. Reset min-pointer.
- Delete(H, e)
  1. DecreaseKey$(H, e, -\infty)$
  2. ExtractMin(H)

ExtractMin

1. Remove minimal node m from the root list
2. Insert children of m into the root list
3. Merge heap-ordered trees with the same degrees until all trees have a different degree:
   Array of degrees $a[1, \ldots, n]$ of elements, empty at beginning. For each element e of the root list:
   a. Let g be the degree of e
   b. If $a[g] = nil$: $a[g] \leftarrow e$.
   c. If $e' := a[g] \neq nil$: Merge e with e' resutling in e'' and set $a[g] \leftarrow nil$. Set e'' unmarked. Re-iterate with $e \leftarrow e''$ having degree $g + 1$.

DecreaseKey (H, e, k)

1. Remove e from its parent node p (if existing) and decrease the degree of p by one.
2. Insert(H, e)
3. Avoid too thin trees:
   a. If $p = nil$ then done.
   b. If p is unmarked: mark p and done.
   c. If p marked: unmark p and cut p from its parent pp. Insert (H, p). Iterate with $p \leftarrow pp$.

Runtimes

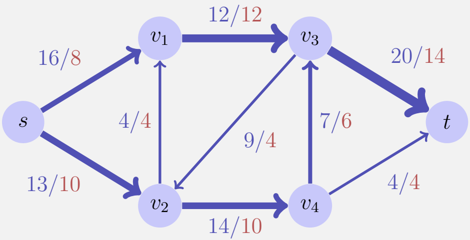
	Binary Heap (worst-Case)	Fibonacci Heap (amortized)
MakeHeap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
ExtractMin	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(1)$
DecreaseKey	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$

Flow

A *Flow* $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- *Bounded Capacity*:
  For all $u, v \in V$: $f(u,v) \leq c(u,v)$.
- *Skew Symmetry*:
  For all $u, v \in V$: $f(u,v) = -f(v,u)$.
- *Conservation of flow*:
  For all $u \in V \setminus \{s,t\}$:

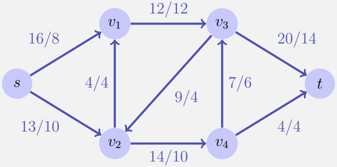
$$\sum_{v \in V} f(u,v) = 0.$$

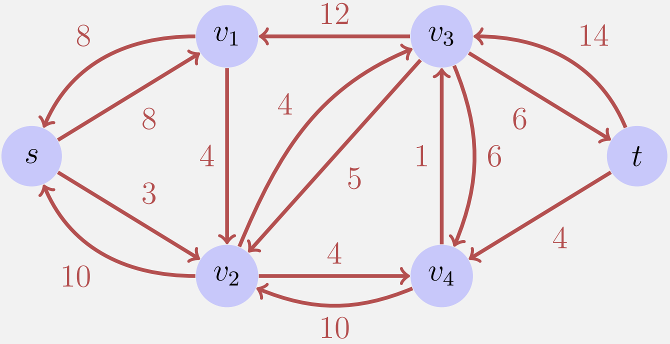


Value of the flow:
$|f| = \sum_{v \in V} f(s,v)$.
Here $|f| = 18$.

Rest Network

Rest network G_f provided by the edges with positive rest capacity:





Rest networks provide the same kind of properties as flow networks with the exception of permitting antiparallel edges

Observation

Theorem

Let $G = (V, E, c)$ be a flow network with source s and sink t and f a flow in G. Let G_f be the corresponding rest networks and let f' be a flow in G_f. Then $f \oplus f'$ with

$$(f \oplus f')(u, v) = f(u, v) + f'(u, v)$$

defines a flow in G with value $|f| + |f'|$.

Augmenting Paths

expansion path p: simple path from s to t in the rest network G_f.

Rest capacity $c_f(p) = \min\{c_f(u,v) : (u,v) \text{ edge in } p\}$

Flow in G_f

Theorem

The mapping $f_p : V \times V \to \mathbb{R}$,

$$f_p(u,v) = \begin{cases} c_f(p) & \textit{if } (u,v) \textit{ edge in } p \\ -c_f(p) & \textit{if } (v,u) \textit{ edge in } p \\ 0 & \textit{otherwise} \end{cases}$$

provides a flow in G_f *with value* $|f_p| = c_f(p) > 0$.

f_p is a flow (easy to show). there is one and only one $u \in V$ with $(s,u) \in p$. Thus $|f_p| = \sum_{v \in V} f_p(s,v) = f_p(s,u) = c_f(p)$.

Max-Flow Min-Cut Theorem

Theorem

Let f be a flow in a flow network $G = (V, E, c)$ with source s and sink t. The following statementsa are equivalent:

1. *f is a maximal flow in G*
2. *The rest network G_f does not provide any expansion paths*
3. *It holds that $|f| = c(S, T)$ for a cut (S, T) of G.*

Algorithm Ford-Fulkerson(G, s, t)

Input : Flow network $G = (V, E, c)$
Output : Maximal flow f.

for $(u, v) \in E$ **do**
    $f(u, v) \leftarrow 0$

while Exists path $p : s \rightsquigarrow t$ in rest network G_f **do**
    $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$
    foreach $(u, v) \in p$ **do**
        if $(u, v) \in E$ **then**
            $f(u, v) \leftarrow f(u, v) + c_f(p)$
        else
            $f(v, u) \leftarrow f(u, v) - c_f(p)$

Edmonds-Karp Algorithm

Choose in the Ford-Fulkerson-Method for finding a path in G_f the expansion path of shortest possible length (e.g. with BFS)

Theorem

When the Edmonds-Karp algorithm is applied to some integer valued flow network $G = (V, E)$ with source s and sink t then the number of flow increases applied by the algorithm is in $\mathcal{O}(|V| \cdot |E|)$
$\Rightarrow$ *Overal asymptotic runtime:* $\mathcal{O}(|V| \cdot |E|^2)$

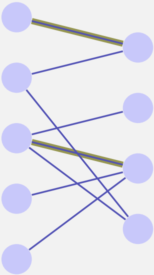
[Without proof]

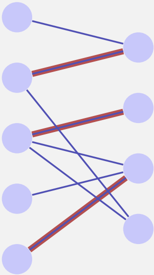
Application: maximal bipartite matching

Given: bipartite undirected graph $G = (V, E)$.

Matching M: $M \subseteq E$ such that $|\{m \in M : v \in m\}| \leq 1$ for all $v \in V$.

Maximal Matching M: Matching M, such that $|M| \geq |M'|$ for each matching M'.





3. Programming Task

Task 10.3: Union Find

- Input: *union* operations to be performed, followed by queries if they are located in the same set.
- Output: For each query, answer if they are in the same set.
- Make sure you can re-use your code in the next task.

Task 10.4: Kruskal's MST algorithm

- Edges have to be sorted.

Task 10.4: Kruskal's MST algorithm

- Edges have to be sorted.
- Create an *Edge* class that implements the comparison operator.
- Then use *std::sort*.

Questions?