Datenstrukturen und Algorithmen

Exercise 11

FS 2020

Program of today

1. Feedback of last exercise
2. Repetition theory
3. MaxFlow
4. Two Quizzes

1. Feedback of last exercise

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?"

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;
}
```

Exercise Union-Find

```
class UnionFind{
    std::vector<size_t> parents_;
public:
    UnionFind(size_t size) : parents_(size, size) { };

    size_t find(size_t index){
        while(parents_[index] != parents_.size())
            index = parents_[index];
        return index;
    }

    void unite(size_t a, size_t b){
        parents_[find(a)] = b;
    }
};
```

Exercise Kruskal

```
class Edge{
public:
    size_t u_, v_;
    int c_;
    Edge(size_t u, int v, int c) : u_(u), v_(v), c_(c) {}

    bool operator<(const Edge& other) const {
        return c_ < other.c_;
    }
};
```

Exercise Kruskal

```
std::vector<Edge> edges;

...

UnionFind uf(n_ + 1);
sort(edges.begin(), edges.end());
for(auto e : edges){
        size_t i=uf.find(e.u_);
        size_t j=uf.find(e.v_);
        if(i != j){
                out.addEdge(e);
                uf.unite(i, j);
        }
}
```

2. Repetition theory

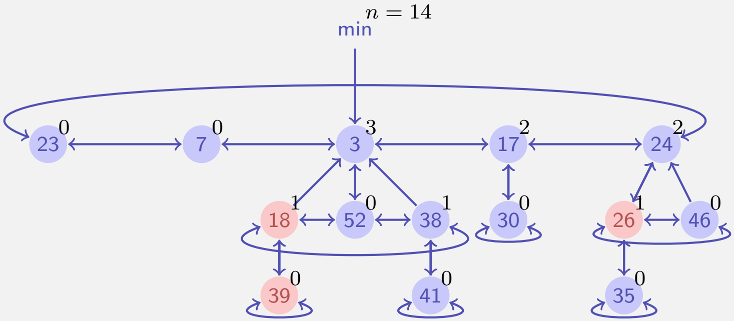
Fibonacci Heaps

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 $a[1, \dots, n]$ 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	$\Theta(\log n)$	$\Theta(\log n)$

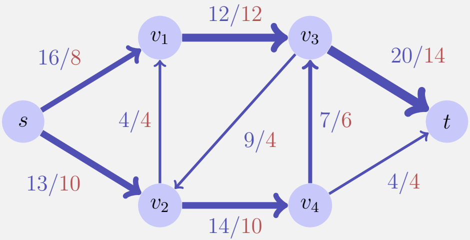
3. MaxFlow

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 $u \in V \setminus \{s,t\}$:

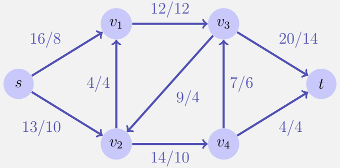
$$\sum_{v \in V} f(u,v) = 0.$$

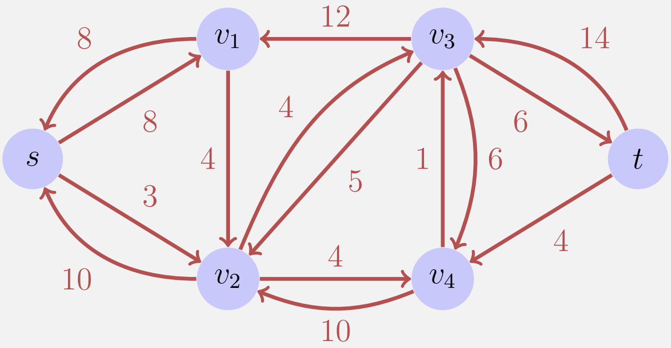


Value of the flow:
$|f| = \sum_{v \in V} f(s,v)$.
Here $|f| = 18$.

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

Augmenting Paths

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

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

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

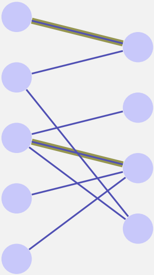
When the Edmonds-Karp algorithm is applied to some integer valued flow network $G = (V, E)$ with source s and sink t then the number of flow increases applied by the algorithm is in $\mathcal{O}(|V| \cdot |E|)$
$\Rightarrow$ Overal asymptotic runtime: $\mathcal{O}(|V| \cdot |E|^2)$

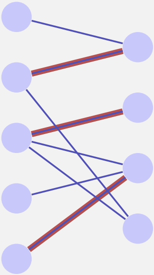
Application: maximal bipartite matching

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'.

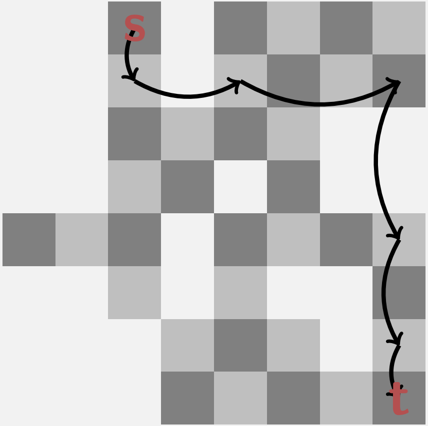


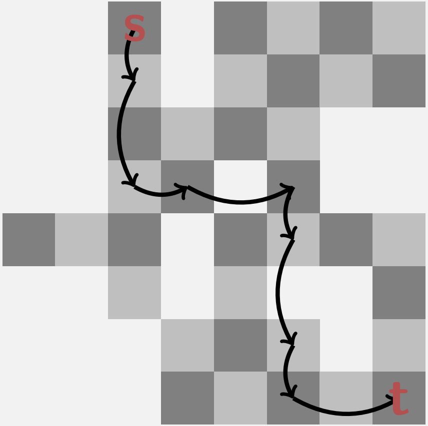


4. Two Quizzes

[Exam 2018.01], Tasks 4 and 5

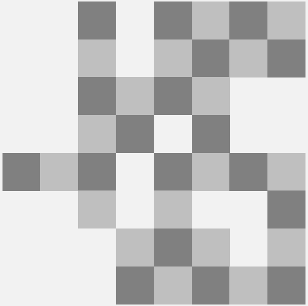
Shortest Path Question

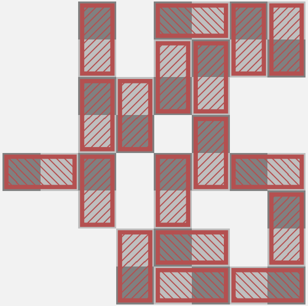




Most important question: What is the corresponding state space?

Max Flow Question





Most important question: How to map this to a max-flow (matching) setup?

Questions?