Datenstrukturen und Algorithmen

Übung 11

FS 2020

Programm von heute

1. Feedback letzte Übung
2. Wiederholung Theorie
3. MaxFlow
4. Zwei Quiz-Fragen

1. Feedback letzte Übung

Closeness Centrality

- Gegeben: Eine Adjazenzmatrix für einen *ungerichteten* Graphen auf n Knoten.
- Aufgabe: für jeden Knoten v die *Closeness Centrality* $C(v)$ von v.

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

- Intuitiv: Wenn viele verbundene Knoten nahe bei v liegen, dann ist $C(v)$ klein.
- „Wie zentral ist ein Knoten in seiner Zusammenhangskomponente?“

Alle kürzesten Pfade

```
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;
}
```

Aufgabe 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;
    }
};
```

Aufgabe 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_;
    }
};
```

Aufgabe 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. Wiederholung Theorie

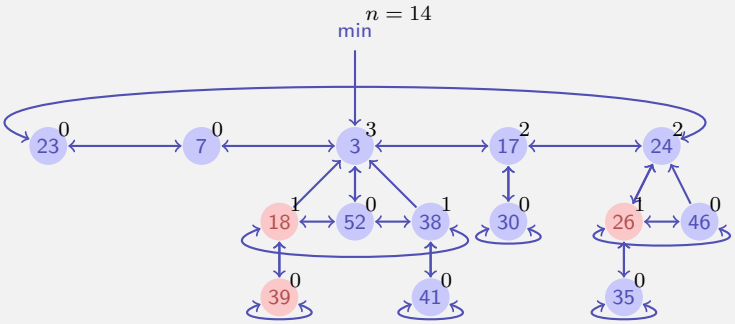
Fibonacci Heaps

Datenstruktur zur Verwaltung von Elementen mit Schlüsseln.
Operationen

- MakeHeap(): Liefere neuen Heap ohne Elemente
- Insert(H, x): Füge x zu H hinzu
- Minimum(H): Liefere Zeiger auf das Element m mit minimalem Schlüssel
- ExtractMin(H): Liefere und entferne (von H) Zeiger auf das Element m
- Union(H_1, H_2): Liefere Verschmelzung zweier Heaps H_1 und H_2
- DecreaseKey(H, x, k): Verkleinere Schlüssel von x in H zu k
- Delete (H, x): Entferne Element x von H

Implementation

Doppelt verkettete Listen von Knoten mit marked-Flag und Anzahl Kinder. Zeiger auf das minimale Element und Anzahl Knoten.



Einfache Operationen

- MakeHeap (trivial)
- Minimum (trivial)
- Insert(H, e)
  1. Füge neues Element in die Wurzelliste ein
  2. Wenn Schlüssel kleiner als Minimum, min-pointer neu setzen.
- Union (H_1, H_2)
  1. Wurzellisten von H_1 und H_2 aneinander hängen
  2. Min-Pointer neu setzen.
- Delete(H, e)
  1. DecreaseKey$(H, e, -\infty)$
  2. ExtractMin(H)

ExtractMin

1. Entferne Minimalknoten m aus der Wurzelliste
2. Hänge Liste der Kinder von m in die Wurzelliste
3. Verschmelze solange heapgeordnete Bäume gleichen Ranges, bis alle Bäume unterschiedlichen Rang haben:
   Rangarray $a[1, \ldots, n]$ von Elementen, zu Beginn leer. Für jedes Element e der Wurzelliste:
   a. Sei g der Grad von e.
   b. Wenn $a[g] = nil$: $a[g] \leftarrow e$.
   c. Wenn $e' := a[g] \neq nil$: Verschmelze e mit e' zu neuem e'' und setze $a[g] \leftarrow nil$. Setze e'' unmarkiert Iteriere erneut mit $e \leftarrow e''$ vom Grad $g + 1$.

DecreaseKey (H, e, k)

1. Entferne e von seinem Vaterknoten p (falls vorhanden) und erniedrige den Rang von p um eins.
2. Insert(H, e)
3. Vermeide zu dünne Bäume:
   a. Wenn $p = nil$, dann fertig
   b. Wenn p unmarkiert: markiere p, fertig.
   c. Wenn p markiert: unmarkiere p, trenne p von seinem Vater pp ab und Insert(H, p). Iteriere mit $p \leftarrow pp$.

Laufzeiten

	Binary Heap (worst-Case)	Fibonacci Heap (amortisiert)
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	$\Theta(\log n)$	$\Theta(\log n)$

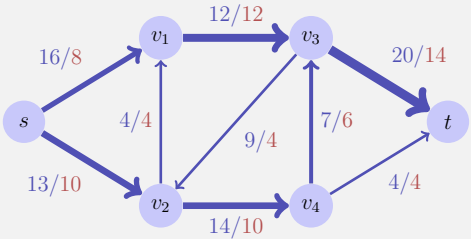
3. MaxFlow

Fluss

Ein *Fluss* $f : V \times V \to \mathbb{R}$ erfüllt folgende Bedingungen:

- *Kapazitätsbeschränkung*:
  Für alle $u, v \in V$: $f(u, v) \leq c(u, v)$.
- *Schiefsymmetrie*:
  Für alle $u, v \in V$: $f(u, v) = -f(v, u)$.
- *Flusserhaltung*:
  Für alle $u \in V \setminus \{s, t\}$:

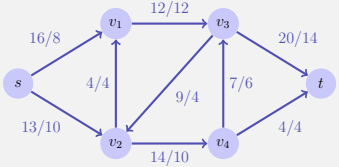
$$\sum_{v \in V} f(u, v) = 0.$$

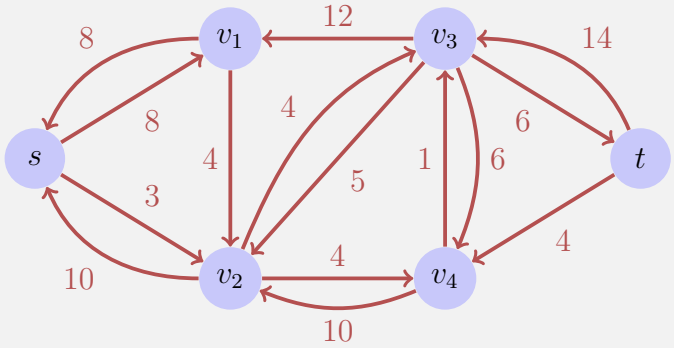


Wert w des Flusses:
$|f| = \sum_{v \in V} f(s, v)$.
Hier $|f| = 18$.

Restnetzwerk

Restnetzwerk G_f gegeben durch alle Kanten mit Restkapazität:





Restnetzwerke haben dieselben Eigenschaften wie Flussnetzwerke, ausser dass antiparallele Kanten zugelassen sind.

Erweiterungspfade

Erweiterungspfad p: einfacher Pfad von s nach t im Restnetzwerk G_f.

Restkapazität $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ Kante in } p\}$

Max-Flow Min-Cut Theorem

Theorem

Wenn f ein Fluss in einem Flussnetzwerk $G = (V, E, c)$ mit Quelle s und Senke t is, dann sind folgende Aussagen äquivalent:

1. *f ist ein maximaler Fluss in G*
2. *Das Restnetzwerk G_f enthält keine Erweiterungspfade*
3. *Es gilt $|f| = c(S, T)$ für einen Schnitt (S, T) von G.*

Algorithmus Ford-Fulkerson(G, s, t)

Input: Flussnetzwerk $G = (V, E, c)$
Output: Maximaler Fluss f.

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

while Existiert Pfad $p : s \rightsquigarrow t$ im Restnetzwerk 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 Algorithmus

Wähle in der Ford-Fulkerson-Methode zum Finden eines Pfades in G_f jeweils einen Erweiterungspfad kürzester Länge (z.B. durch Breitensuche).

Theorem

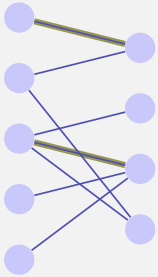
Wenn der Edmonds-Karp Algorithmus auf ein ganzzahliges Flussnetzwerk $G = (V, E)$ mit Quelle s und Senke t angewendet wird, dann ist die Gesamtanzahl der durch den Algorithmus angewendete Flusserhöhungen in $\mathcal{O}(|V| \cdot |E|)$
$\Rightarrow$ *Gesamte asymptotische Laufzeit: $\mathcal{O}(|V| \cdot |E|^2)$*

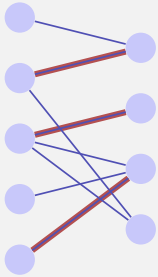
Anwendung: Maximales bipartites Matching

Gegeben: bipartiter ungerichteter Graph $G = (V, E)$.

Matching M: $M \subseteq E$ so dass $|\{m \in M : v \in m\}| \leq 1$ für alle $v \in V$.

Maximales Matching M: Matching M, so dass $|M| \geq |M'|$ für jedes Matching M'.

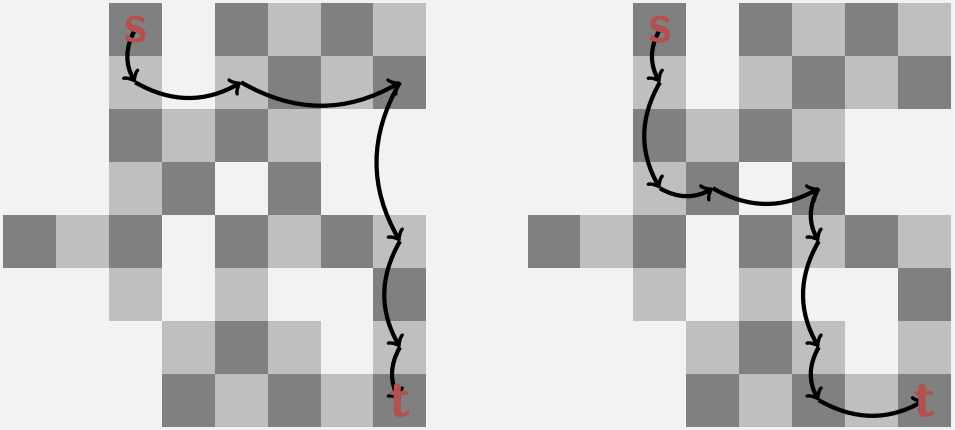




4. Zwei Quiz-Fragen

[Exam 2018.01], Aufgaben 4 und 5

Kürzeste Wege Frage



Wichtigste Frage: Was ist der dazugehörige Zustandsraum?

Max Flow Question

Wichtigste Frage: Wie bildet man das auf ein Max-Flow (Matching) Problem ab?

Fragen?