Exercise Session 12

Data Structures and Algorithms, D-MATH, ETH Zurich

Program of today

Feedback of last exercise

MaxFlow

C++ Threads

Two Quizzes

1. Feedback of last exercise

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;
    }
};
```

Union-Find with Map (Exercise Kruskal)

```
class UnionFind {
  private:
    std::unordered_map<NodeP,NodeP> parent;
    std::unordered_map<NodeP,unsigned> depth;
  public:
    void MakeSet(NodeP n){
      parent[n] = n; depth[n] = 0;
    }
    NodeP Find(NodeP n){
      while (parent[n] != n){
        n = parent[n];
      }
      return n;
    }
```

Optimizing Union

```
bool Union(NodeP l, NodeP r){
  l = Find(l);
  r = Find(r);
  if (l == r){
    return false;
  } else {
    if (depth[l] < depth[r])
      std::swap(l,r);
    parent[r] = l;
    if (depth[l] == depth[r])
      depth[l]++;
    return true;
  }
}
```

Alternative: optimizing Find

```
NodeP Find(NodeP n){
  NodeP root = n;
  while (parent[root] != root){
    root = parent[root];
  }
  while (parent[n] != root){
    auto next = parent[n];
    parent[n] = root;
    n = next;
  }
  return root;
}
```

no **depth** required

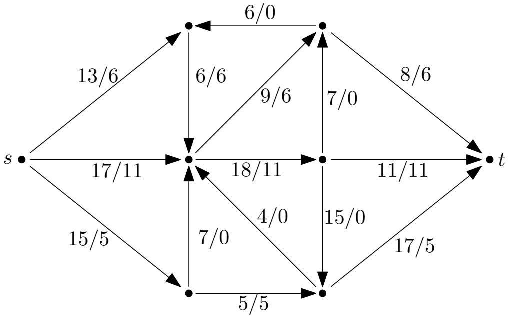
Kruskal

```
std::vector<Edge> result;
std::sort(edges.begin(),edges.end(),
  [](const Edge& l, const Edge& r) {return l.length < r.length;}
);

UnionFind uf;
for (auto n: nodes)
  uf.MakeSet(n);

for (const auto& e: edges){
  if (uf.Union(e.source, e.target)){
    result.push_back(e);
  }
}
return result;
```

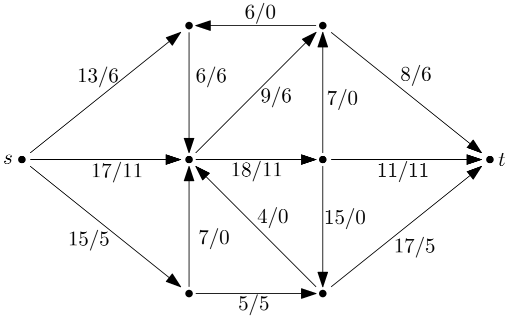
TSP

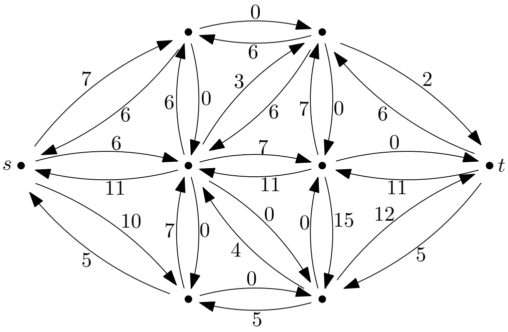
```
std::vector<Edge> Graph::TSP(){
  std::vector<Edge> mst = Kruskal();
  std::unordered_map<NodeP,std::vector<Edge>> adj;
  for (const auto& e: mst){
    adj[e.source].push_back(e);
    adj[e.target].emplace_back(e.target,e.source);
  }
  return DFS(adj,mst[0].source);
}
```

Exercise Manual Max-Flow



Exercise Manual Max-Flow





Travelling Salesperson Problem

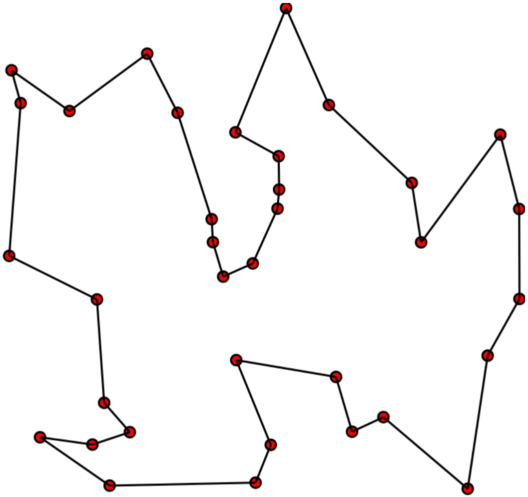
Problem

Given a map and list of cities, what is the shortest possible route that visits each city once and returns at the original city?

Mathematical model

On an undirected, weighted graph G, which cycle containing all of G's vertices has the lowest weight sum?

Travelling Salesperson Problem



Travelling Salesperson Problem

- The problem has no known polynomial-time solution.
- Many heuristic algorithms exists. They do not always return the optimal solution.

Travelling Salesperson Problem

- The heuristic algorithm that you are asked to implement on CodeExpert (*The Travelling Student*) on CodeExpert uses an MST:
    1. Compute the minimum spanning tree M
    2. Make a depth first search on M
- The algorithm is 2-approximate, meaning that the solution it generates has at most twice the cost of the optimal solution.
- The algorithm assumes a complete graph $G = (V, E, c)$that satisfies the triangle inequality: $c(v, w) \leq c(v, x) + c(x, w) \; \forall \; v, w, x \in V$

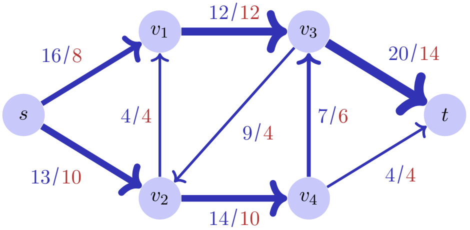
2. MaxFlow

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

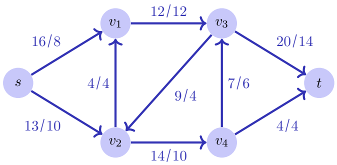
$$\sum_{v \in V} f(u, v) = 0.$$

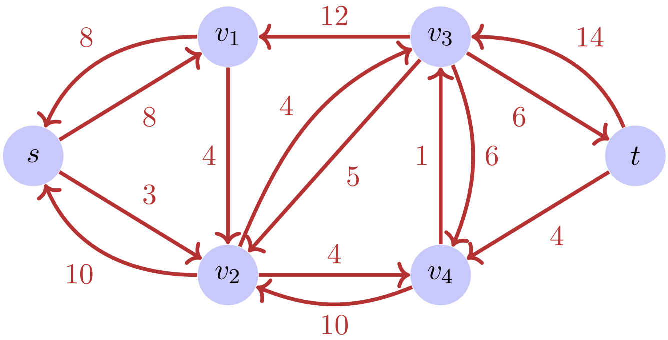


Value of the flow:
$|f| = \sum_{v \in V} f(s, v)$.
Here $|f| = 18$.

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 1

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) ∈ E do
    f(u,v) ← 0
while Exists path p : s ⇝ t in rest network G_f do
    c_f(p) ← min{c_f(u,v) : (u,v) ∈ p}
    foreach (u,v) ∈ p do
        if (u,v) ∈ E then
            f(u,v) ← f(u,v) + c_f(p)
        else
            f(v,u) ← 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 2

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 $\mathcal{O}(|V| \cdot |E|)$

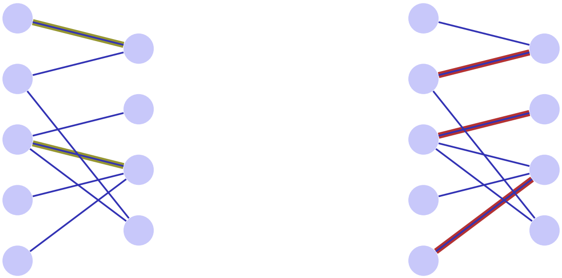
$\Rightarrow$ *Overal asymptotic runtime:* $\mathcal{O}(|V| \cdot |E|^2)$

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 $v \in V$.

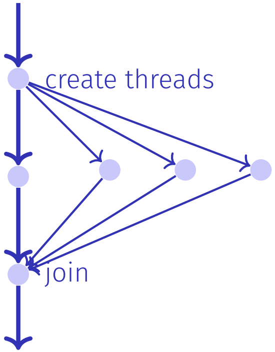
Maximal Matching M: Matching M, such that $|M| \geq |M'|$ for each matching M'.



3. C++ Threads

C++11 Threads

```
void hello(int id){
  std::cout << "hello from " << id << "\n";
}

int main(){
  std::vector<std::thread> tv(3);
  int id = 0;
  for (auto & t:tv)
    t = std::thread(hello, ++id);
  std::cout << "hello from main \n";
  for (auto & t:tv)
    t.join();
  return 0;
}
```



Nondeterministic Execution!

One execution:

hello from main
hello from 2
hello from 1
hello from 0

Other execution:

hello from 1
hello from main
hello from 0
hello from 2

Other execution:

hello from main
hello from 0
hello from hello from 1
2

Technical Details I

- With allocating a thread, reference parameters are copied, except explicitly std::ref is provided at the construction.

Technical Details I

- With allocating a thread, reference parameters are copied, except explicitly std::ref is provided at the construction.

```
void calc( std::vector<int>& very_long_vector ){
  // doing funky stuff with very_long_vector
}
int main(){
  std::vector<int> v( 1000000000 );
  std::thread t1( calc, v );          // bad idea, v is copied
  // here v is unchanged
  std::thread t2( calc, std::ref(v) ); // good idea, v is not copied
  // here v is modified
  std::thread t2( [&v]{calc(v)}; } ); // also good idea
  // here v is modified
  // ...
```

Technical Details II

- Threads cannot be copied.

Technical Details II

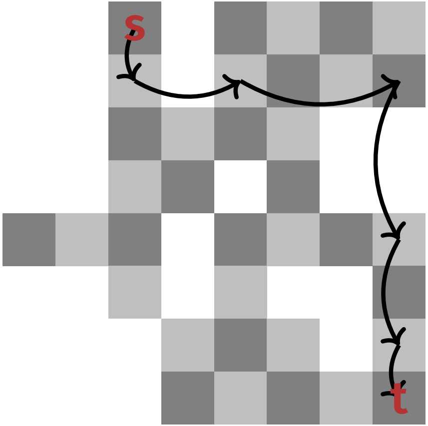
- Threads cannot be copied.

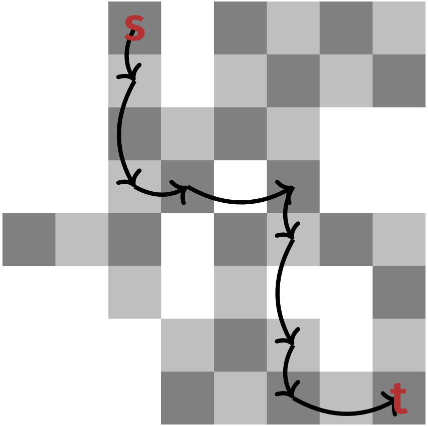
```
{
  std::thread t1(hello);
  std::thread t2;
  t2 = t1; // compiler error
  t1.join();
}
{
  std::thread t1(hello);
  std::thread t2;
  t2 = std::move(t1); // ok
  t2.join();
}
```

4. Two Quizzes

[Exam 2018.01], Tasks 4 and 5

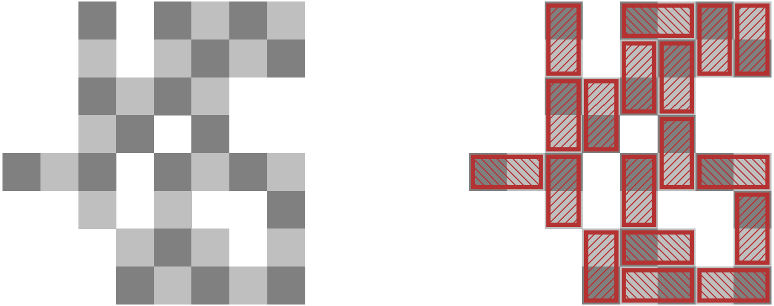
Shortest Path Question





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?