Datenstrukturen und Algorithmen

Exercise 12

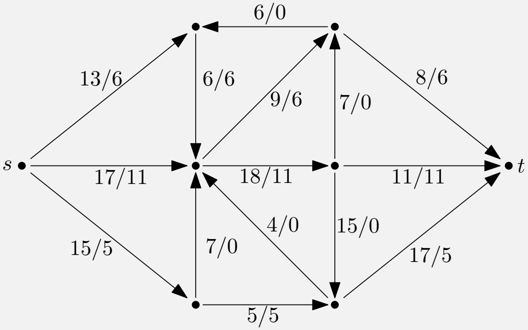
FS 2019

Program of today

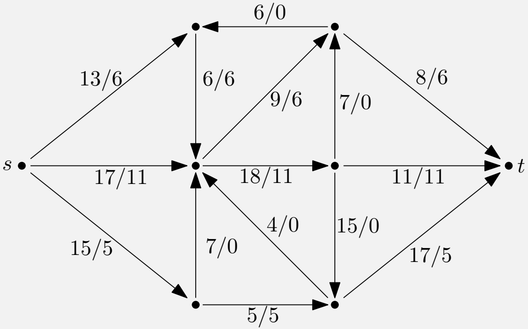
1. Feedback of last exercise
2. MaxFlow
3. Two Quizzes
4. Parallel Programming
5. Programming Tasks

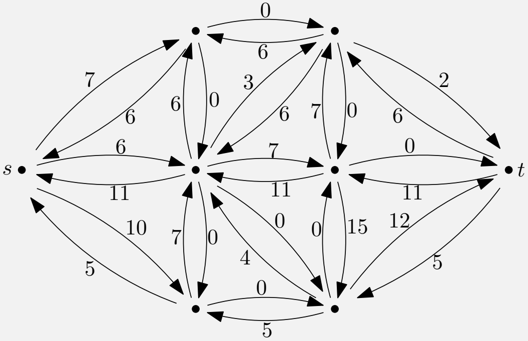
1. Feedback of last exercise

Exercise Manual Max-Flow



Exercise Manual Max-Flow





Exercise Applying Maximum Flow

- Vertex capacity: replace vertex with an in-vertex and and out-vertex.
- Connect these vertices by an edge with this capacity.

Exercise Union-Find

```cpp
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);
        }
}
```

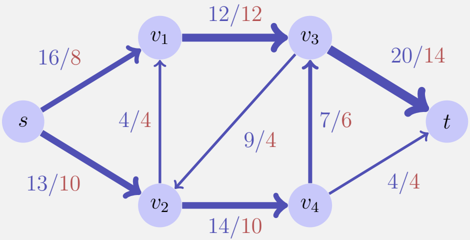
2. MaxFlow

Flow

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

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

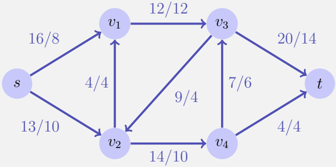
$$\sum_{v \in V} f(u,v) = 0.$$

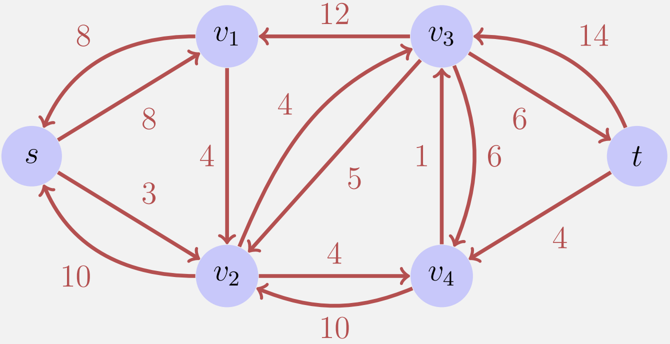


Value of the flow:
$|f| = \sum_{v \in V} f(s,v)$.
Here $|f| = 18$.

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

[Without proof]

Application: maximal bipartite matching

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

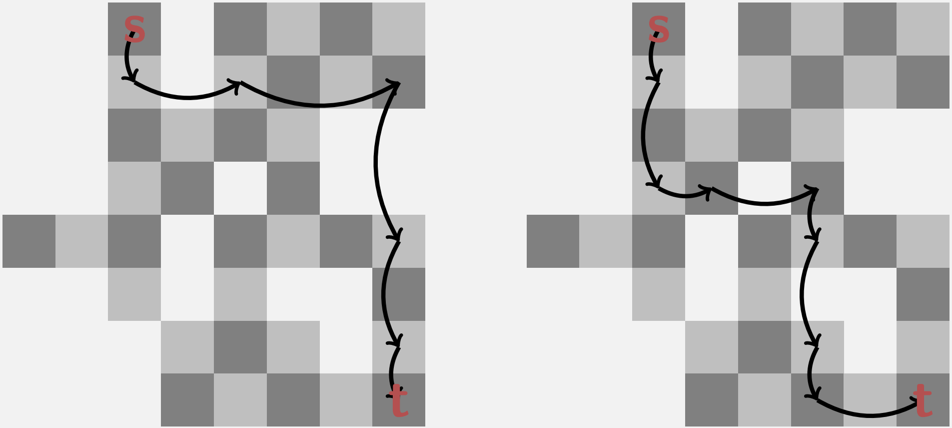
Matching M: $M \subseteq E$ such that $|\{m \in M : v \in m\}| \leq 1$ for all $v \in V$.

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

3. Two Quizzes

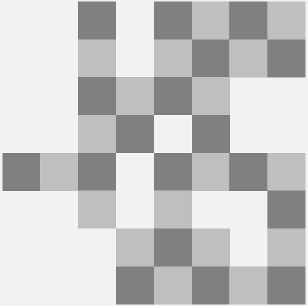
[Exam 2018.01], Tasks 4 and 5

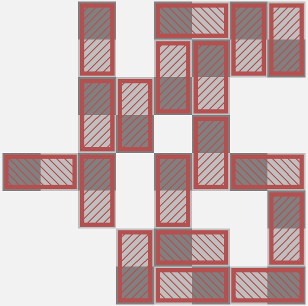
Shortest Path Question



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?

4. Parallel Programming

Parallel Performance

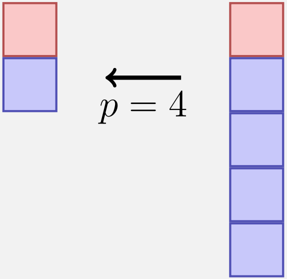
Given

- fixed amount of computing work W (number computing steps)
- Sequential execution time T_1
- Parallel execution time on p CPUs

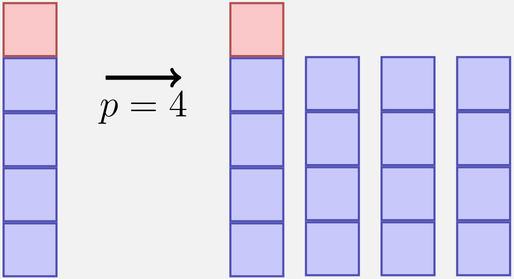
	runtime	speedup	efficiency
perfection (linear)	$T_p = T_1/p$	$S_p = p$	$E_p = 1$
loss (sublinear)	$T_p > T_1/p$	$S_p < p$	$E_p < 1$
sorcery (superlinear)	$T_p < T_1/p$	$S_p > p$	$E_p > 1$

Amdahl vs. Gustafson

Amdahl



Gustafson



Amdahl vs. Gustafson, or why do we care?

Amdahl	**Gustafson**
pessimist	optimist
strong scaling	weak scaling

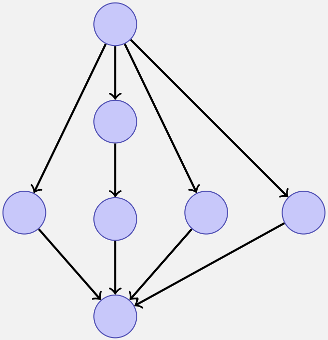
Amdahl vs. Gustafson, or why do we care?

Amdahl	**Gustafson**
pessimist	optimist
strong scaling	weak scaling

$\Rightarrow$ need to develop methods with small sequential protion as possible.

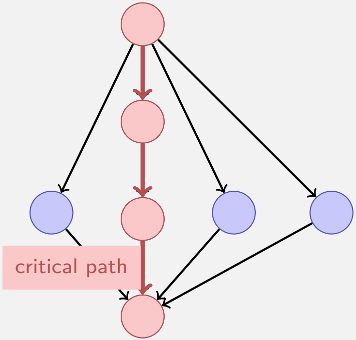
Question

- Each Node (task) takes 1 time unit.
- Arrows depict dependencies.
- Minimal execution time when number of processors $= \infty$?



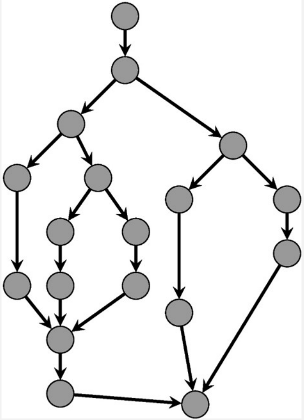
Question

- Each Node (task) takes 1 time unit.
- Arrows depict dependencies.
- Minimal execution time when number of processors $= \infty$?



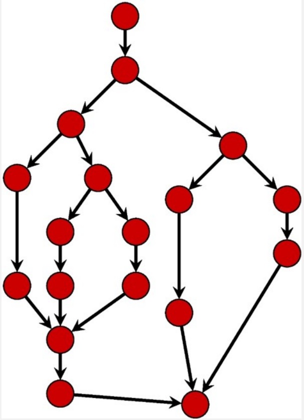
Performance Model

- p processors
- Dynamic scheduling
- T_p: Execution time on p processors



Performance Model

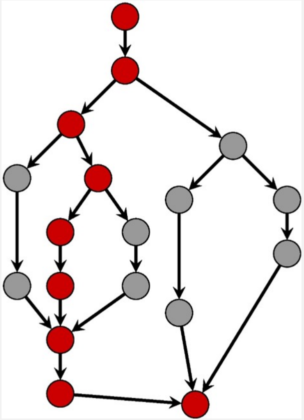
- T_p: Execution time on p processors
- T_1: *work*: time for executing total work on one processor
- T_1/T_p: Speedup



Performance Model

- T_∞: *span*: critical path, execution time on ∞ processors. Longest path from root to sink.
- T_1/T_∞: *Parallelism:* wider is better
- Lower bounds:

$$T_p \geq T_1/p \quad \text{Work law}$$
$$T_p \geq T_\infty \quad \text{Span law}$$



Greedy Scheduler

Greedy scheduler: at each time it schedules as many as availbale tasks.

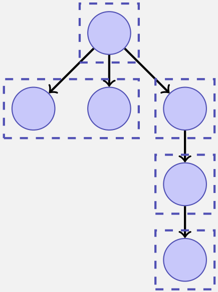
Theorem

On an ideal parallel computer with p processors, a greedy scheduler executes a multi-threaded computation with work T_1 and span T_∞ in time

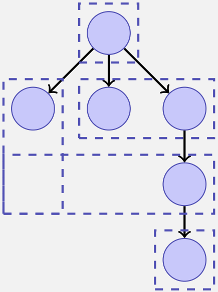
$$T_p \leq T_1/p + T_\infty$$

Beispiel

Assume $p = 2$.



$T_p = 5$



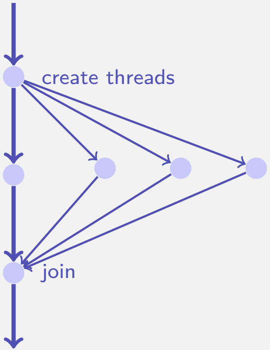
$T_p = 4$

5. Programming Tasks

C++11 Threads

```cpp
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();
}
```

Questions?