Datenstrukturen und Algorithmen

Exercise 11

FS 2019

Program of today

1. Feedback of last exercise
2. Repetition theory
   - Algorithm Jarnik, Prim, Dijkstra
3. Programming Task

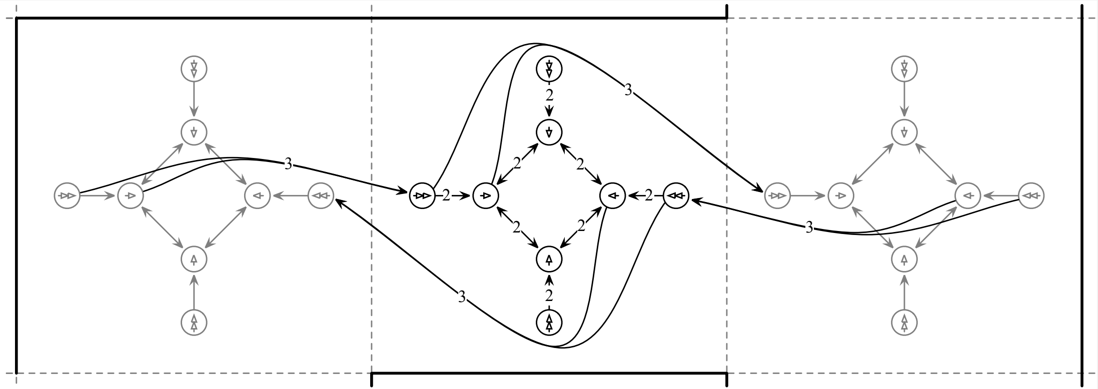
1. Feedback of last exercise

Exercise : Labyrinth

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

Exercise 9.1: Labyrinth

- position × direction × speed



- Runtime?

Exercise 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 $\mathcal{O}(n \log n)$

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

```cpp
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

Algorithm MST-Kruskal(G)

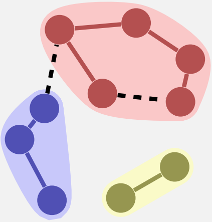
Input: Weighted Graph $G = (V, E, c)$
Output: Minimum spanning tree with edges A.

Sort edges by weight $c(e_1) \leq \ldots \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ **to** m **do**
    if $(V, A \cup \{e_k\})$ acyclic **then**
        $A \leftarrow E' \cup \{e_k\}$

return (V, A, c)

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)) // conceptual: $A \leftarrow A \cup e_k$
    else // conceptual: $R \leftarrow R \cup e_k$

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):

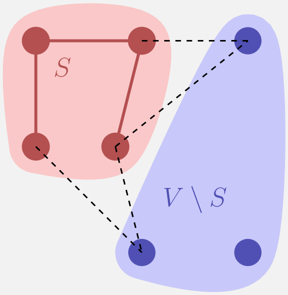
$j \leftarrow i$
while $(p[i] \neq i)$ **do** $i \leftarrow p[i]$
while $(j \neq i)$ **do**
    $t \leftarrow j$
    $j \leftarrow p[j]$
    $p[t] \leftarrow i$
return i

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

MST algorithm of Jarnik, Prim, Dijkstra

Idea: start with some $v \in V$ and grow the spanning tree from here by the acceptance rule.

$S \leftarrow \{v_0\}$
for $i \leftarrow 1$ **to** $|V|$ **do**
  Choose cheapest (u, v) mit $u \in S$, $v \notin S$
  // conceptual $A \leftarrow A \cup \{(u, v)\}$
  $S \leftarrow S \cup \{v\}$ // (Coloring)



Remark: a union-Find data structure is not required. It suffices to color nodes when they are added to S.

Running time

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

Improvements (like with Dijkstra's ShortestPath)

- Memorize cheapest edge to S: for each $v \in V \setminus S$. $\deg^+(v)$ many updates for each new $v \in S$. Costs: $|V|$ many minima and updates: $\mathcal{O}(|V|^2 + \sum_{v \in V} \deg^+(v)) = \mathcal{O}(|V|^2 + |E|)$
- With Minheap: costs $|V|$ many minima $= \mathcal{O}(|V| \log |V|)$, $|E|$ Updates: $\mathcal{O}(|E| \log |V|)$, Initialization $\mathcal{O}(|V|)$: $\mathcal{O}(|E| \cdot \log |V|.)$
- With a Fibonacci-Heap: $\mathcal{O}(|E| + |V| \cdot \log |V|)$.

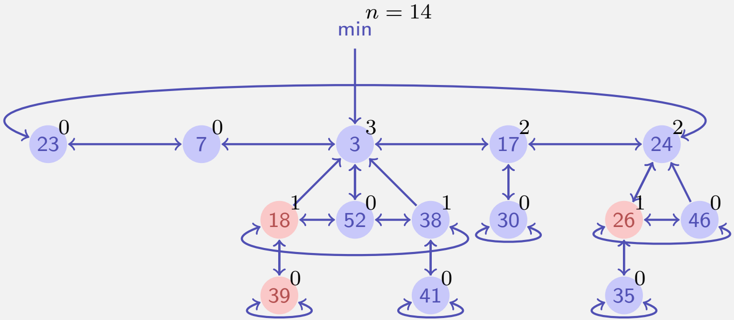
Fibonacci Heaps

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	$\Theta(\log n)$	$\Theta(\log n)$

3. Programming Task

Task 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 Kruskal's MST algorithm

- Edges have to be sorted.

Task Kruskal's MST algorithm

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

Questions?