CS 225 – Data Structures

ZJUI – Spring 2021

Lecture 9: Graph Algorithms I

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

9 Common Graph Algorithms

9.1 Minimum Spanning Trees

Let us look at connected undirected graphs $G = (V, E)$, where the edges $e \in E$ are labelled by costs $c(e) > 0$

We want to find a subset $E' \subseteq E$ of edges such that the subgraph (V, E') is connected and $\sum_{e \in E'} c(e)$ is minimal

It is clear that a solution $G' = (V, E')$ must be a tree: if not, there is a cycle in G' and we can remove any edge from this cycle preserving the property of G' being connected and further reducing the total costs

Applications: Suppose you want to connect n islands of some small country by ferry lines such that each island can be reached from any other island and the costs for the ferries are minimised

The same problem arises for cost-efficient support lines for water, electricity, cables, etc. in some city

Definitions & Terminology

Let $G = (V, E)$ be a connected undirected graph with a cost function $c : E \to \mathbb{R}_+$. A tree $T = (V, E')$ is called a ***minimum spanning tree*** for G iff $\sum_{e \in E'} c(e)$ is minimal.

We will now investigate two greedy algorithms for determining minimum spanning trees, for which the following terminology will be useful:

- A subset $E' \subseteq E$ of edges is **feasible** iff (V, E') does not contain cycles
- A subset $E' \subseteq E$ of edges is **promising** iff (V, E') can be extended to a minimum spanning tree—clearly, $\emptyset$ is promising and feasible
- An edge $e \in E$ **touches** a subset $V' \subseteq V$ of vertices iff exactly one of the end-points of e is in V'

A Helpful Lemma

Let G be as above and $V' \subseteq V$ a proper subset of the vertex set V. If $E' \subseteq E$ is promising such that no edge $e \in E'$ touches V' and $e \in E$ is an edge with minimal cost $c(e)$ and one end-point in V', then $E' \cup \{e\}$ is promising.

The lemma suggest a simple way for the construction of a minimum spanning tree: start with $E' = \emptyset$, then extend E' step-by-step adding one edge e

The added edge e must have an end-point in a set of vertices V' not touching the edges in E' and have minimum cost $c(e)$ among such edges

Both the algorithms by Kruskal and Prim will exploit this approach with different choices of V'

Proof

As E' is promising, there exists a minimum spanning tree (V, E'') with $E' \subseteq E''$

If we have $e \in E''$, there is nothing more to show: $E' \cup \{e\} \subseteq E''$ is promising

If we have $e \notin E''$, the graph $(V, E'' \cup \{e\})$ contains a cycle with edge e

As e is incident to V', there must also be another edge $e' \in E''$ that is incident to V'

If we replace e' by e, we obtain a spanning tree $(V, E'' - \{e'\} \cup \{e\})$

The edge e was selected with $c(e) \leq c(e')$, so this spanning tree is as well a minimum spanning tree, and its edge set contains $E' \cup \{e\}$

Kruskal’s Algorithm

We start with $E' = \emptyset$, so each vertex forms its own connected component $\{v\}$

Then V' is the set of all vertices that are not touched by any edge of E', so initially $V' = V$

Order the edges $e \in E$ with respect to increasing costs $c(e)$ and process the edges in this order

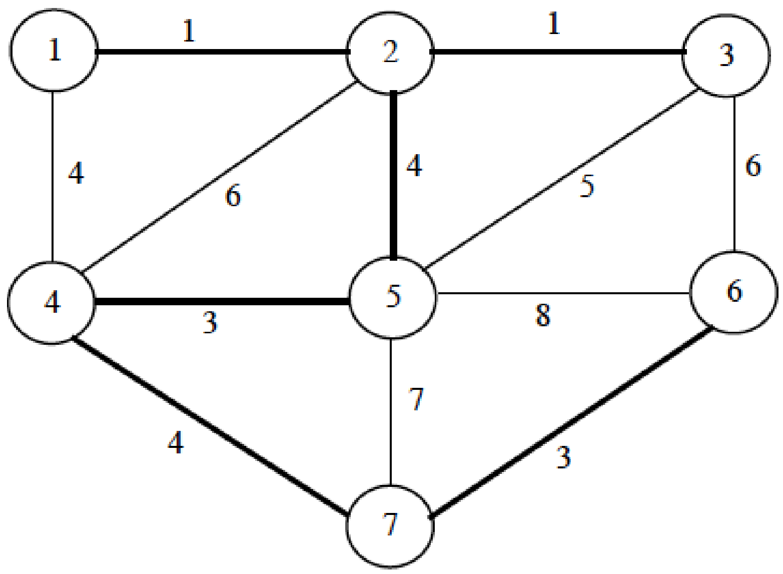
If e satisfies the conditions of the lemma, i.e. it has one end-point in V', then add e to E'

In doing so the connected components of the end-points of e are merged

When there is only one connected component, the algorithm terminates with (V, E') being the constructed minimum spanning tree

Example: Kruskal's Algorithm

Consider the following graph:



Take edges in the following order (with increasing costs):

$$E = \{\{1,2\},\{2,3\},\{4,5\},\{6,7\},\{2,5\},\{1,4\},\\ \{4,7\},\{3,5\},\{2,4\},\{3,6\},\{5,7\},\{5,6\}\}\ .$$

Example/ cont.

We obtain the following connected components:

Step	Edge	Connected Components
0 (initialisation)	—	$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
1	$\{1, 2\}$	$\{1, 2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
2	$\{2, 3\}$	$\{1, 2, 3\}, \{4\}, \{5\}, \{6\}, \{7\}$
3	$\{4, 5\}$	$\{1, 2, 3\}, \{4, 5\}, \{6\}, \{7\}$
4	$\{6, 7\}$	$\{1, 2, 3\}, \{4, 5\}, \{6, 7\}$
5	$\{2, 5\}$	$\{1, 2, 3, 4, 5\}, \{6, 7\}$
6	$\{1, 4\}$	rejected
7	$\{4, 7\}$	$\{1, 2, 3, 4, 5, 6, 7\}$

As there is only one connected component, we have obtained a minimum spanning tree, and there is no need to consider further edges

Complexity

Let $|V| = n$ and $|E| = m$, then we must have $n - 1 \leq m \leq \frac{n(n-1)}{2}$, as the graph is connected and we can ignore loops

We have to sort the edges, which requires time in $O(m \log m) = O(m \log n)$

Initially we build n disjoint sets of vertices, which requires time in $O(n)$

We have at most m steps, and in each step we have to search twice the connected components with together n elements

The search requires time in $O(\log n)$, so the effort in all steps together gives rise to a time complexity in $O(m \log n)$

For dense graphs we have $m \approx n^2$—so the complexity is in $O(n^2 \log n)$—while for sparse graphs we have $m \approx n$ and the complexity will be in $O(n \log n)$

Prim's Algorithm

In Kruskal's algorithm always the next promising edge is selected, regardless whether we obtain a connected subgraph or not

The main difference in Prim's algorithm is that after each step (V, E') is connected

That is, we start with $E' = \emptyset$ and $W = \{v_0\}$ with some arbitrary $v_0 \in V$

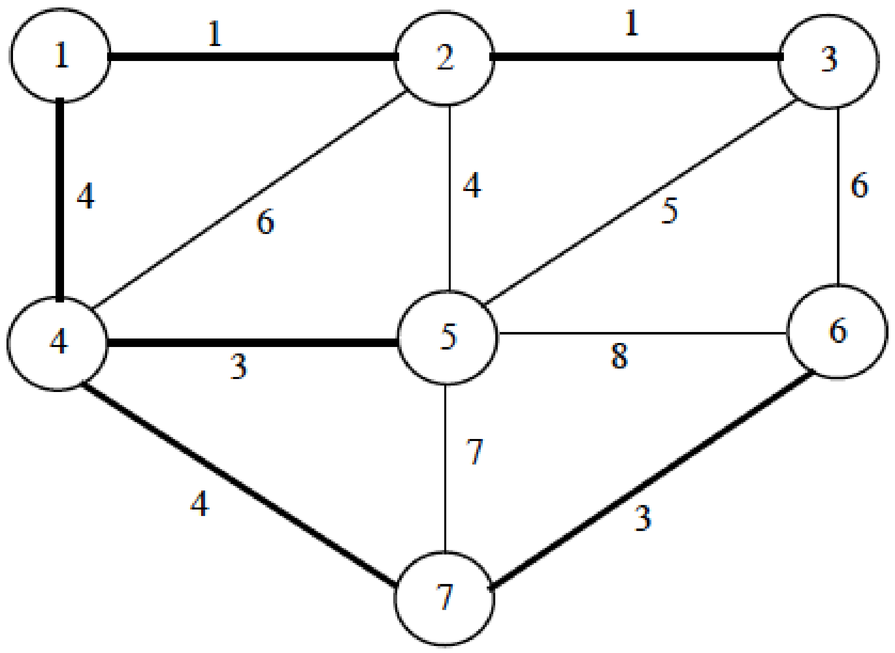
In every step choose $e \in E$ with minimal $c(e)$ such that e is incident to W, and add e to E'

If $e = (v, w)$ with $v \in W$, then add w to W

The algorithm terminates when we reach $W = V$

Example: Prim's Algorithm

Consider the following graph:



Choose 1 as the initial node

Example / cont.

Here we can select edges in the order

$$\{1,2\}, \{2,3\}, \{1,4\}, \{4,5\}, \{4,7\}, \{6,7\} ,$$

which produces the minimum spanning tree

Step	**Selected Edge**	W
0 (initialisation)	—	$\{ 1 \}$
1	$\{1,2\}$	$\{1,2\}$
2	$\{2,3\}$	$\{1,2,3\}$
3	$\{1,4\}$	$\{1,2,3,4\}$
4	$\{4,5\}$	$\{1,2,3,4,5\}$
5	$\{4,7\}$	$\{1,2,3,4,5,7\}$
6	$\{6,7\}$	$\{1,2,3,4,5,6,7\}$

Complexity

Assume again $|V| = n$ and $|E| = m$

In each step one vertex is added to W, so we have exactly $n - 1$ steps

If in every step we use an array, which for each vertex $w \notin W$ contains the vertex $v \in V$ such that $c(v, w)$ is minimal, the time complexity of a single step is in $O(n)$

So the time complexity of Prim's algorithm is always in $O(n^2)$

For dense graphs Prim's algorithm performs better than Kruskal's one; for sparse graphs Kruskal's algorithm is the better one

For sparse graphs there are more sophisticated minimum spanning tree algorithms that outperform Kruskal's algorithm

Greedy Algorithms

We remarked that the algorithms by Kruskal and Prim are **greedy algorithms**

In general, greedy algorithms form a simple class of algorithms connected to optimisation problems, which chooses the most promising candidate from a set of candidates

Greedy algorithms do not always guarantee that an optimum is really achieved

Another important class of algorithms are **divide & conquer algorithms**, which use a top-down approach dividing a problem into smaller ones that are solved recursively

Examples for divide & conquer are mergesort, the towers of Hanoi, the Euclidean algorithm, the median/selection algorithm, etc.

A third important class of algorithms are called **dynamic programming algorithms**, which exploit an optimality principle, according to which in a sequence of optimal decisions subsequences must also be optimal

We will see an example for dynamic programming in our treatment of shortest path algorithms

A Semi-External MST Algorithm

What about finding a minimum spanning tree for a graph that is too large to be stored in main memory

If the list of vertices is still small enough such that the union-find data structure can be kept in main memory, we can define a semi-external version of Kruskal's algorithm

Keep the graph in external storage and also use an in-place sorting algorithm (such as mergesort) that keeps the edge list in external storage during sorting

Use main memory for the union-find data structure representing the connected components of the partly constructed MST

Whenever an edge connects two previously separated connected components, this edge can be printed on the output; otherwise the edge is simply discarded

Edge Contraction

Edge contraction is a technique that can be applied, when already the number of vertices is so large that the union-find data structure cannot be kept in main memory

In this case, whenever (u, v) is an edge in the minimum spanning tree, edge contraction identifies the two nodes, i.e. every edge (v, w) incident to v wll be replaced by an edge (u, w)

In addition, preserve the information that the original edge was (v, w) in an attribute added to the edge (u, w), which is kept only in external storage

Computing a minimum spanning tree with edge contraction then allows us to easily transform the result into an minimum spanning tree of the original graph

This gives the idea of an external MST algorithm, which contracts edges in the minimum spanning tree as long as the union-find data structure is too large, and switches to a semi-external MST algorithm once this becomes feasible

One such algorithm is **Sibeyn's algorithm** (see the Mehlhorn/2008 textbook, pp. 226ff.), which we do not investigate here

9.2 The Union-Find Data Structure

We leave the (similar) implementations of the algorithms of Kruskal and Prim as lab exercises (see lab/discussion 8)

Let us stress the importance of using appropriate data structures in these algorithms

For instance, in Kruskal's algorithm we need to store connected components given by a partition of the vertex set V, find the connected component of a given vertex and build the union of connected components

This can be done by using an array *parent* of length $|V|$, where initially $\mathit{parent}[i] = i$ holds

We may see *parent* as a representation of a forest, where each tree corresponds to a connected component

Then two vertices i and j are in the same connected component iff the roots of their trees coincide—we can **find** these roots by following the *parent* links until we reach some r with $\mathit{parent}[r] = r$

The Union-Find Data Structure / Optimisation

In case these roots are different, the **union** of the connected components results from **linking** the two roots making one the parent of the other

However, if *find* and *link* are implemented naively, a poor performance may result

Therefore, use another array *rank* of length $|V|$, where initially $\mathit{rank}[i] = 0$ holds

When linking connected components, where i and j are the root nodes of the representing trees, make j the parent of i, if $\mathit{rank}[i] < \mathit{rank}[j]$ holds

In case $\mathit{rank}[i] = \mathit{rank}[j]$ choose i as the parent and increment its rank

In addition, optimise *find* by updating $\mathit{parent}[i]$ to r, if we discover that r is the root of the representing tree

We leave implementation details to the lab exercises

Find and Merge

Rules for MERGE and FIND can be specified as follows:

```
MERGE(i,j) =
    if      rank(i) = rank(j)
    then    rank(i) := rank(i) + 1
            parent(j) := i
    else    if      rank(i) > rank(j)
            then    parent(j) := i
            else    parent(i) := j
            endif
    endif
```

```
$r \leftarrow$ FIND($k$) =
    $r := k$
    while parent($r$) $\neq r$
          do
                $r :=$ parent($r$)
          enddo
    $i := k$
    while $i \neq r$
          do  par
                   parent($i$) := $r$
                   $i :=$ parent($i$)
              endpar
          enddo
```

We omit the obvious definition of the signatures

The Union-Find Data Structure / Rank

Union by rank ensures that the height of any tree representing a connected component never exceeds $\log_2 |V|$.

Proof. Show by induction that a tree of rank k contains at least 2^k elements—this is obvious for $k = 0$

The rank increases from $k - 1$ to k, when a child of rank $k - 1$ is added

By the induction hypothesis the children have at least 2^{k-1} elements

As there are at least two children, the tree of rank k has at least 2^k elements, which completes the induction proof

Complexity Analysis

In order to analyse the worst case time complexity for a random sequence of n MERGE and FIND operations we apply a **dirty trick**

We extend the FIND rule by additional updates that are not needed for the computation, but allow us to estimate the complexity

We use additional counters defined by a function symbol global of arity 0 and a function symbol cost of arity 1—the location (cost, i) will only be defined for $1 \leq i \leq N = |V|$

Initially, we will have $val(\text{global}) = 0$ and $val(\text{cost}, i) = 0$ for all $1 \leq i \leq N$

We assume a static unary function $F : \mathbb{N} \to \mathbb{N}$ that is strictly monotone increasing with $F(0) = 0$—we will determine F later

Define a derived unary function $G : \mathbb{N} \to \mathbb{N}$ by $G(n) = \min\{m \in \mathbb{N} \mid F(m) \geq n\}$

Extend G to a "*group*" function G defined on V by letting $G(v) = G(\text{rank}(v))$

Extended FIND Rule

Then the extended FIND rule is defined as follows:

```
$r \leftarrow \textsc{find}(k) =$
   $r := k$
   **while** $\text{parent}(r) \neq r$
         **do** $r := \text{parent}(r)$ **enddo**
   $i := k$
   **while** $i \neq r$
         **do** **par**
               **if** $G(\text{rank}(i)) < G(\text{rank}(\text{parent}(i)) \vee r = \text{parent}(i)$
               **then** global := global $+1$
               **else** $\text{cost}(i) := \text{cost}(i) + 1$
               **endif**
               $\text{parent}(i) := r$
               $i := \text{parent}(i)$
            **endpar**
         **enddo**
```

Time Estimation

For a call of the FIND rule the number of times we run through both loops is the same, and it is given by the increase of global $+ \sum_{i=1}^{N} \text{cost}(i)$

All other operations require some constant time, so the number of elementary steps of FIND is in $O(1 + inc)$, where inc is the increase of global $+ \sum_{i=1}^{N} \text{cost}(i)$

The number of steps required by a call to the MERGE rule is bounded by a constant

Hence, the total time required for a random sequence of calls to MERGE and FIND including also the initialisation is in

$$O\Big(N + n + val_{S_{\text{fin}}}(\text{global}) + \sum_{i=1}^{N} val_{S_{\text{fin}}}(\text{cost}, i)\Big) \enspace ,$$

where S_{fin} is the final state reached after completion of the n calls

So $val_{S_{\text{fin}}}(\text{global})$ and $val_{S_{\text{fin}}}(\text{cost}, i)$ refer to the values of the counters after the execution of the sequence of n operations

Observations

We make the following relevant observations:

- If v is not a root, then $\text{rank}(v) < \text{rank}(\text{parent}(v))$—this is obvious from the definition of the rank
- Once v ceases to be the root of a tree, it will never become a root again, and its rank does not change anymore

  This is an obvious consequence of the definition of the MERGE rule
- For any vertex v we always have $\text{rank}(v) \leq \log_2 \text{size}(t_v)$, where $\text{size}(t_v)$ is the number of vertices in the tree t_v, in which v occurs

  This follows by induction (see slide 443)

Observations / cont.

- At any time for any k there are no more than $N/2^k$ vertices of rank k

Proof. Define $sub_k(v) = \emptyset$, if vertex v never reaches rank$(v) = k$, otherwise $sub_k(v)$ is the set of vertices in the subtree rooted at v, when v reaches rank$(v) = k$

Due to our second observation, when v reaches rank$(v) = k$, it must have been a root

$sub_k(v) \neq \emptyset$ implies $|sub_k(v)| \geq 2^k$ due to our third observation above

Due to our first observation we have $sub_k(v) \cap sub_k(w) = \emptyset$ for $v \neq w$

If the number of vertices of rank k were $> N/2^k$, the total number of vertices would be $> N$, which is a contradiction

Observations / cont.

- The function G is monotone inceasing

Proof. If we have $n_1 \leq n_2$, then for all m with $F(m) \geq n_2$ we also have $F(m) \geq n_1$, and $\{m \in \mathbb{N} \mid F(m) \geq n_2\} \subseteq \{m \in \mathbb{N} \mid F(m) \geq n_1\}$

This implies $G(n_1) = \min\{m \in \mathbb{N} \mid F(m) \geq n_1\} \leq \min\{m \in \mathbb{N} \mid F(m) \geq n_2\} = G(n_2)$

- At any time for any vertex v we have $\text{rank}(v) \leq \lfloor \log_2 N \rfloor$, and consequently $G(v) \leq G(\lfloor \log_2 N \rfloor)$

Proof. If we have $\text{rank}(v) = k > \lfloor \log_2 N \rfloor$, we get $2^k > N$ and $N/2^k < 1$

As there are at most $N/2^k$ vertices of rank k, there cannot be such a vertex v

Estimation of Complexity / 1

Now return to our estimation of the time complexity; first consider the increase to global by a call to the FIND rule

For the $i \neq r$ considered in the second while loop we always have $G(\text{rank}(i)) \leq G(\text{rank}(\text{parent}(i))$ due to our first observation

All these group values are bounded by $G(\lfloor \log_2 N \rfloor)$ due to our last observation

So the possible group values range from 0 to $G(\lfloor \log_2 N \rfloor)$, and hence the increase of the value of global is at most $1 + G(\lfloor \log_2 N \rfloor)$

Consequently, after at most n FIND operations we have that $val_{S_{\text{fin}}}(\text{global})$ is bounded by $c \cdot \Big(1 + nG(\lfloor \log_2 N \rfloor)\Big)$ for some constant $c > 0$

Estimation of Complexity / 2

Now consider the increase to cost(i) for all $1 \leq i \leq N$ by a call to the FIND rule

As long as i is a root, the value of cost(i) remains 0

So let r be the rank of i, when i ceases to be a root; according to our second observation above this rank will never change anymore

cost(i) can only be incremented, when its parent is not a root and satisfies $G(\text{rank}(\text{parent}(i$
$G(r)$

According to the definition of G we have $G(n) = G(r)$ iff $F(G(r) - 1) < n < F(G(r))$ holds—the second $<$ holds, because n is the rank of a child, so $n < r$ holds, whereas $F(G(r)) \geq r$

When cost(i) is incremented, a path containing i is compressed, i.e. the root is made the parent of all vertices in the path

Hence, the increment of cost(i) can be done at most $F(G(r)) - F(G(r) - 1) - 1$ times

Estimation of Complexity / 3

Then the final value of $\text{cost}(i)$ is $\leq F(G(r))$ for every vertex $i \in \text{final}(r)$, where $\text{final}(r)$ is the set of vertices that cease to be a root, when they have rank $r > 0$

If i never ceases to be a root, and has rank 0 when becoming a child of another vertex, we get $val_{S_{\text{fin}}}(\text{cost}, i) = 0$

With $K = G(\lfloor \log_2 N \rfloor) - 1$ we then get

$$\begin{aligned}\sum_{i=1}^{N} val_{S_{\text{fin}}}(\text{cost}, i) &= \sum_{g=0}^{K} \sum_{r=F(g)+1}^{F(g+1)} \sum_{i \in \text{final}(r)} val_{S_{\text{fin}}}(\text{cost}, i) \\ &\leq \sum_{g=0}^{K} \sum_{r=F(g)+1}^{F(g+1)} \sum_{i \in \text{final}(r)} F(G(r)) \\ &\leq \sum_{g=0}^{K} \sum_{r=F(g)+1}^{F(g+1)} \frac{N}{2^r} F(g+1) \leq N \sum_{g=0}^{K} \frac{F(g+1)}{2^{F(g)}}\end{aligned}$$

Estimation of Complexity / 4

Finally, let $F(g+1) = 2^{F(g)}$ for all $g \leq 0$, so we get

$$\sum_{i=1}^{N} val_{S_{\text{fin}}}(\text{cost}, i) \leq N \cdot (K+1) = N \cdot G(\lfloor \log_2 N \rfloor)$$

Taking our estimates together, the time needed for a sequence of n calls to MERGE and FIND (including initialisation) is in

$$\begin{aligned} O\Big(N + n + val_{S_{\text{fin}}}(\text{global}) + \sum_{i=1}^{N} val_{S_{\text{fin}}}(\text{cost}, i)\Big) \\ \subseteq O(N + n + n \cdot G(\lfloor \log_2 N \rfloor) + N \cdot G(\lfloor \log_2 N \rfloor)) \\ = O(\max(n, N) \cdot (1 + G(\lfloor \log_2 N \rfloor)) \end{aligned}$$

As we set $F(g+1) = 2^{F(g)}$ with $F(0) = 0$ we get

$$G(n) \;=\; \log_2^* n = \min\{k \mid \underbrace{\log_2 \ldots \log_2}_{k \text{ times}} N \leq 0\}$$

with $G(\lfloor \log_2 N \rfloor) = G(N) - 1$

Final Complexity Result

For $n \geq N$ we have obtained the following result:

The time complexity of a random sequence of n calls of MERGE and FIND for the union-find data structure with N elements is in $O(n \cdot \log_2^* N)$, provided $n \geq N$ holds

Here $\log_2^* N$ is called the **iterated logarithm** of N

Note that $G(N) = \log_2^* N$ increases very slowly: we have $\log_2^* N \leq 5$ for all $N \leq 65536$ and $\log_2^* N \leq 6$ for all $N \leq 2^{65536}$

Thus the complexity of a random sequence of n calls of MERGE and FIND for the union-find data structure with N elements is almost linear, i.e. in $O(n)$

The complexity estimate can be further improved using the Ackermann function—this is even more tricky

9.3 Shortest Path Algorithms

Let us now consider a directed graph $G = (V, E)$ with a cost function $c : E \to \mathbb{R}_+$

The shortest path problem consists in determining for each pair (v, w) of vertices a path $v = v_0, \dots, v_k = w$ such that $\sum_{i=1}^{k} c(v_{i-1}, v_i)$ is minimal

Clearly such shortest paths with minimum edge costs are simple, and for $v = w$ the cost of a shortest path is 0

We will look mainly into two common shortest path algorithms: the greedy algorithm by Dijkstra and Floyd's algorithm based in dynamic programming

For the latter one we exploit that if $v_0, \dots, v_k$ is a shortest path from v_0 to v_k, then a subpath $v_i, \dots, v_j$ $(i \leq j)$ is the shortest path from v_i to v_j

Dijkstra's Algorithm

Let us consider that we use one start node v_0—the general case is handled by considering in parallel all nodes as start nodes

We can use a matrix to represent the edge costs using $c(v, w) = \infty$ when there is no edge from v to w

Use a subset V' of V to indicate, for which nodes v we have not yet determined the shortest path length from v_0 to v—initially, this subset is V, and finally it shall be $\emptyset$

In each step select one node $v \in V'$ and compute the length of paths from v_0 to all other nodes w, where the last edge is (v, w)

If this is shorter than the currently known shortest path length, update this path length accordingly

Dijkstra's Algorithm: Formalisation

For the formalisation and initialisation of Dijkstra's algorithm we proceed as follows:

- Let Nodes be a nullary function symbol that represents all the nodes in our graph that still have to be explored

  Nodes is initialised by the value $\{1, \dots, k\}$, i.e. the set of all nodes except a dedicated start node 0

- Let Edge be a binary function symbol such that $\mathit{Edge}(i,j)$ takes a value in $\mathbb{R}$, the costs of moving directly from i to j—if there is no such edge, let it be ∞

  Edge is initialised by some values

- Let Cost be a unary function symbol such that $\mathit{Cost}(j)$ takes a value in $\mathbb{R}$, the costs of moving from 0 to j

  Intially we have $\mathit{Cost}(j) = \mathit{Edge}(0,j)$

Dijkstra's Algorithm: Specification

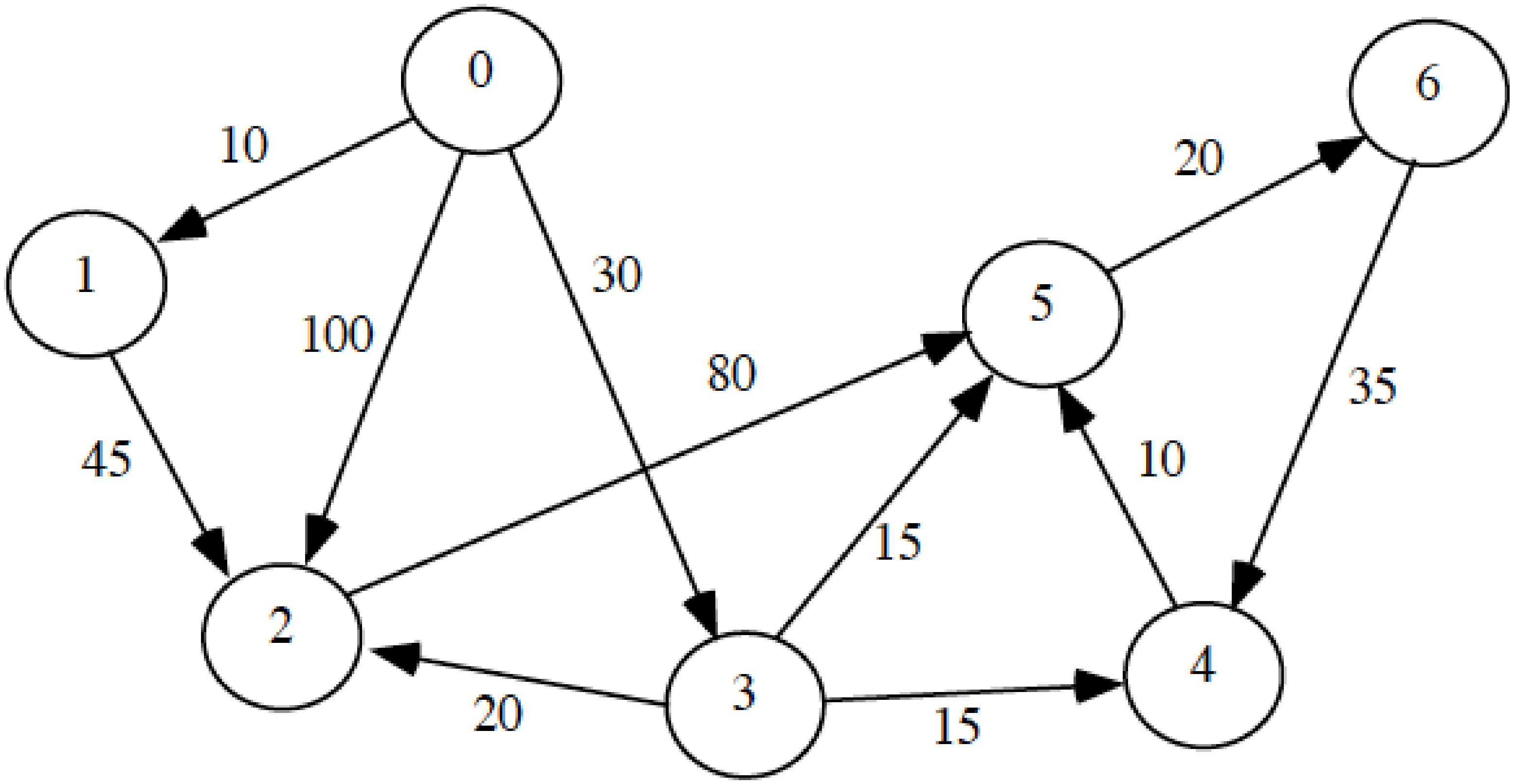
Then we can specify Dijkstra's algorithm using the following rule:

```
IF $Nodes \neq \emptyset$
THEN CHOOSE $i$ WITH $i \in Nodes \wedge \forall j . j \in Nodes \rightarrow Cost(i) \leq Cost(j)$
        DO PAR
              FORALL $j \in \{1, \ldots, k\}$
                  DO
                      $Cost(j) := \min(Cost(j), Cost(i) + Edge(i, j))$
                  ENDDO
              $Nodes := Nodes - \{i\}$
              ENDPAR
        ENDDO
ENDIF
```

Note that here we used a **FORALL** rule, which in this case is just a shortcut for a parallel rule for all nodes j

Example: Dijkstra's Algorithm

Let us consider the following directed graph:



Example / cont.

The matrix on the left represents the edge costs, for which we used *Edge*

The rows of the matrix on the right show the development of the cost vector for start node 0, for which we used *Cost*—we underline the cost of the selected node

$$
\begin{bmatrix}
\infty & 10 & 100 & 30 & \infty & \infty & \infty \\
\infty & \infty & 45 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & 80 & \infty \\
\infty & \infty & 20 & 15 & 15 & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & 10 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & 20 \\
\infty & \infty & \infty & \infty & 35 & \infty & \infty
\end{bmatrix}
\qquad
\begin{matrix}
\underline{10} & 100 & 30 & \infty & \infty & \infty \\
10 & 55 & \underline{30} & \infty & \infty & \infty \\
10 & 50 & 30 & \underline{45} & 45 & \infty \\
10 & 50 & 30 & 45 & \underline{45} & \infty \\
10 & \underline{50} & 30 & 45 & 45 & 65 \\
10 & 50 & 30 & 45 & 45 & \underline{65} \\
10 & 50 & 30 & 45 & 45 & 65
\end{matrix}
$$

The last row shows the lengths of shortest paths from the start node 0

Correctness

Without loss of generality let $V = \{0, \dots, k\}$ and $v_0 = 0$

The correctness of Dijkstra's algorithms follows, if we can show that in every state of the computation the following properties hold:

- If $i \in \mathit{Nodes}$, then $\mathit{Cost}(i)$ is the length of the shortest path from 0 to i containing only intermediate nodes in $\{1, \dots, k\} - \mathit{Nodes}$
- If $i \notin \mathit{Nodes}$, then $\mathit{Cost}(i)$ is the length of the shortest path from 0 to i

As initially $\mathit{Cost}(i) = \mathit{Edge}(0, i)$, these conditions hold in the initial state

Correctness / cont.

Assume that for a state transition we choose $i_0 \in \mathit{Nodes}$, so $\mathit{Cost}(i_0)$ is minimal among nodes in Nodes

Then we have two cases:

- If $i \in \mathit{Nodes} - \{i_0\}$, then $\mathit{Cost}(i)$ and $\mathit{Cost}(i_0)$ were the lengths of the shortest paths from 0 to i and i_0 containing only intermediate nodes in $\{1, \dots, k\} - \mathit{Nodes}$

  So $\mathit{Cost}(i_0) + \mathit{Edge}(i_0, i)$ is the length of a path from 0 to i with intermediate nodes in $\{1, \dots, k\} - (\mathit{Nodes} - \{i_0\})$, and the minimum length of such a path is $\min(\mathit{Cost}(i), \mathit{Cost}(i_0) + \mathit{Edge}(i_0, i))$

- If $i \notin \mathit{Nodes}$, then $\mathit{Cost}(i)$ was already the length of the shortest path from 0 to i, so the assignment does not change this

  The same applies to $i = i_0$, as i_0 was selected to have minimum cost $\mathit{Cost}(i_0)$

Worst Case Complexity

We have to iterate the rule exactly k times, each time choosing a vertex i and eliminating it from $Nodes$

For the choice of the node i we have to make $|Nodes| - 1 \leq k$ comparisons

For the parallel update of $Cost(j)$ we require constant time for each $j \in \{1, \dots, k\}$

Therefore, the time complexity for each execution of the rule is in $O(k)$ and consequently the worst case complexity for Dijkstra's algorithm is in $O(k^2)$

For the average case complexity we refer to a starred Section 10.4 in the textbook by Mehlhorn (2008)

Floyd's Algorithm

Floyd's algorithm is an example for dynamic programming: we use a matrix D with rows and columns corresponding to the vertices

$D(i, j)$ is to contain the length of the shortest path from i to j, which we initialise by the costs of the edges

Also use a set of vertices *Nodes*, which initially contains all vertices

Then we simply iterate the following rule:

```
CHOOSE $i' \in Nodes$ IN
    FORALL $i, j \in \{0, \dots, k\}$
        DO
            $D(i, j) := \min(D(i, j), D(i, i') + D(i', j))$
        ENDDO
    $Nodes := Nodes - \{i'\}$
END
```

Example: Floyd's Algorithm

We apply Floyd's algorithm to the same graph we used as an example for Dijkstra's algorithm

We obtain the following sequences of matrices:

$$\begin{bmatrix} 0 & 10 & 100 & 30 & \infty & \infty & \infty \\ \infty & 0 & 45 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty & 80 & \infty \\ \infty & \infty & 20 & 0 & 15 & 15 & \infty \\ \infty & \infty & \infty & \infty & 0 & 10 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 20 \\ \infty & \infty & \infty & \infty & 35 & \infty & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 10 & 55 & 30 & \infty & \infty & \infty \\ \infty & 0 & 45 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty & 80 & \infty \\ \infty & \infty & 20 & 0 & 15 & 15 & \infty \\ \infty & \infty & \infty & \infty & 0 & 10 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 20 \\ \infty & \infty & \infty & \infty & 35 & \infty & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 10 & 55 & 30 & \infty & 135 & \infty \\ \infty & 0 & 45 & \infty & \infty & 125 & \infty \\ \infty & \infty & 0 & \infty & \infty & 80 & \infty \\ \infty & \infty & 20 & 0 & 15 & 15 & \infty \\ \infty & \infty & \infty & \infty & 0 & 10 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 20 \\ \infty & \infty & \infty & \infty & 35 & \infty & 0 \end{bmatrix}$$

Example / cont.

$$\begin{bmatrix} 0 & 10 & 50 & 30 & 45 & 45 & \infty \\ \infty & 0 & 45 & \infty & \infty & 125 & \infty \\ \infty & \infty & 0 & \infty & \infty & 80 & \infty \\ \infty & \infty & 20 & 0 & 15 & 15 & \infty \\ \infty & \infty & \infty & \infty & 0 & 10 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 20 \\ \infty & \infty & \infty & \infty & 35 & 45 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 10 & 50 & 30 & 45 & 45 & 65 \\ \infty & 0 & 45 & \infty & \infty & 125 & 145 \\ \infty & \infty & 0 & \infty & \infty & 80 & 100 \\ \infty & \infty & 20 & 0 & 15 & 15 & 35 \\ \infty & \infty & \infty & \infty & 0 & 10 & 30 \\ \infty & \infty & \infty & \infty & \infty & 0 & 20 \\ \infty & \infty & \infty & \infty & 35 & 45 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 10 & 50 & 30 & 45 & 45 & 65 \\ \infty & 0 & 45 & \infty & 180 & 125 & 145 \\ \infty & \infty & 0 & \infty & 135 & 80 & 100 \\ \infty & \infty & 20 & 0 & 15 & 15 & 35 \\ \infty & \infty & \infty & \infty & 0 & 10 & 30 \\ \infty & \infty & \infty & \infty & 55 & 0 & 20 \\ \infty & \infty & \infty & \infty & 35 & 45 & 0 \end{bmatrix}$$

Clearly, we obtain a time complexity in $O(k^3)$ for $k = |V|$, as we execute the rule exactly once for each vertex, each time updating k^2 matrix entries

This is the same worst case complexity as for Dijkstra's algorithm, when applied with all different start nodes

9.4 Greedy Algorithms

We briefly discussed two classes of algorithms: greedy algorithms (Kruskal, Prim, Dijkstra) and dynamic programming (Floyd)

We now look into more examples for these classes of algorithms

Greedy algorithms are typically used for optimisation problems aiming at sub-optimal solutions, in particular, when it is infeasible to compute a true optimum

The common greedy approach is as follows:

- Assume an ***objective function*** that is to be optimised, e.g. the path length in the shortest path problem (Dijkstra)

- Assume a set of ***candidates*** and a subset of those candidates that have already been used, e.g. the set of vertices and those that have already been explored (Dijkstra)

Greedy Algorithms / cont.

- Assume a function checking whether a particular set of candidates ***provides a solution*** ignoring any optimility criterion; e.g., in Dijkstra's algorithm we need to process all vertices
- Assume a function that checks if a set of candidates is ***feasible***, i.e. whether it is possible to extend the set to obtain a (not necessarily optimal) solution; e.g., in Dijkstra's algorithm the cost function gives the shortest path using just the already processed vertices
- Assume that there is at least one solution from an initial set of candidates
- Assume a ***selection function*** that indicates the most promising not yet used candidate, e.g. the cost function in Dijkstra's algorithm

A greedy algorithm proceeds step-by-step, each time using the selection function to select a new candidate to be processed and then added to the subset of already explored candidates

Enlarging the set of already processed candidates must preserve feasibility

Scheduling with Processing Time

Assume we have a single **server** p serving n **customers** $C_1, \dots, c_n$, each with a fixed **service time** t_i

The problem is to minimise the **total system time** $\sum_{i=1}^{n}(t_i + w_i)$, where w_i is the **waiting time** of customer c_i

If p has already served customers $c_{i_1}, \dots, c_{i_k}$ with total system time T, then ading next customer c_m increases the total system time by

$$\sum_{i=1}^{k} t_{i_j} + t_m$$

This gives rise to the following **greedy strategy**:

> At each stage add the customer c_m from the set of not yet served customers for which t_m is minimal

Optimality

The greedy strategy for scheduling with processing time leads to minimal total system time.

Proof. Let $\sigma \in S_n$ be a permutation

If customers are selected in the order $\sigma(1), \dots, \sigma(n)$, the total system time is

$$T(\sigma) = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{\sigma(j)} = \sum_{j=1}^{n} (n - j + 1) t_{\sigma(j)}$$

Assume $k < \ell$ with $t_{\sigma(k)} > t_{\sigma(\ell)}$, then take $\tau = \sigma \circ (k, \ell)$, i.e. $\tau(k) = \sigma(\ell)$, $\tau(\ell) = \sigma(k)$, and $\tau(j) = \sigma(j)$ for all other j

So we get

$$\begin{aligned} T(\sigma) - T(\tau) &= \sum_{j=1}^{n} (n - j + 1)(t_{\sigma(j)} - t_{\tau(j)}) \\ &= (n - k + 1)(t_{\sigma(k)} - t_{\sigma(\ell)}) + (n - \ell + 1)(t_{\sigma(\ell)} - t_{\sigma(k)}) \\ &= (\ell - k)(t_{\sigma(k)} - t_{\sigma(\ell)}) > 0 \ , \end{aligned}$$

which proves the claim

Scheduling with Deadlines

Now consider a single **server** p and n **jobs** requiring unit time such that only one jib can be executed at a time by p

The i'th job has a **deadline** d_i and produces a **profit** g_i $(1 \leq i \leq n)$

The problem is to maximise the total profit executing jobs on p before their deadline

Call a set J of k jobs **feasible** iff there is a permutation $\sigma \in S_k$ such that $i \leq d_{\sigma(i)}$ holds for all $1 \leq i \leq k$—read: the i'th job is finished at time point $\sigma(i)$

In this case also the permutsation σ is called **feasible**

An obvious **greedy strategy** is the following:

> Create a set of jobs step-by-step adding a job $\sigma(i+1)$ such that $g_{\sigma(i+1)}$ is maximal among the jobs not yet in the set and the set of jobs is feasible

We will show that it is not necessary to check all $k!$ permutations when checking feasibility

Scheduling / cont.

Lemma. Let J be a set of k jobs and let $\sigma \in S_k$ be a permutation with $d_{\sigma(1)} \leq \cdots \leq d_{\sigma(k)}$. Then J is feasible iff σ is feasible.

Proof. The "if" part is obvious

For the "only if" part assume that J is feasible with some $\tau \in S_k$ satisfying $i \leq d_{\tau(i)}$ for all $1 \leq i \leq k$

For $\tau \neq \sigma$ take ℓ minimal with $\tau(\ell) \neq \sigma(\ell)$ and assume $\tau(\ell) = \sigma(m)$ for some $m > \ell$, hence $d_{\tau(m)} = d_{\sigma(\ell)} \leq d_{\tau(\ell)}$

With $\rho = \tau \circ (\ell, m)$ we get $\rho(\ell) = \tau(m)$, $\rho(m) = \tau(\ell)$ and $\rho(i) = \tau(i)$ for other i

This implies $\ell < m \leq d_{\tau(m)} \leq d_{\rho(\ell)}$ and $m \leq d_{\tau(m)} \leq d_{\tau(\ell)} = d_{\rho(m)}$, i.e. ρ is feasible, and the smallest i for which $\rho(i) \neq \sigma(i)$ is $> \ell$

Thus, we can repeat this construction for ρ instead of τ until we obtain equality with σ, hence σ is feasible

Optimality

The lemma implies that in our greedy strategy at each stage we have to check only a single sequence in order of increasing deadlines to detect, if the extended set of jobs is still feasible

The greedy strategy for scheduling with deadlines leads to an optimal solution, i.e. maximum profit.

Proof. Assume that J is a feasible set of jobs providing a optimal solution, i.e. maximum profit

Assume that the algorithm produces a feasible set I of jobs with $I \neq J$

Let σ_I and σ_J be the corresponding feasible permutations

If I and J have different cardinalities, extend the shorter of the two sequences of jobs by $\perp$, which stands for an idle job with deadline $d_\perp = \infty$ and $g_\perp = 0$

By scanning $\sigma_I(1), \dots, \sigma_I(k)$ and $\sigma_J(1), \dots, \sigma_J(k)$ downward from k to 1 we can change σ_I to τ_I and σ_J to τ_J that $\tau_I(i) = \tau_J(i)$ holds, whenever $\tau_I(i)$ or $\tau_J(i)$ is in $I \cap J$

Optimality Proof / cont.

If we find $\tau_I(i) \in I \cap J$, but $\tau_I(i) \neq \tau_J(i)$, then swap $\tau_J(i)$ with $\tau_J(j) = \tau_I(i)$ for some $j < i$ (analogously the other way round)

Now consider an arbitrary ℓ with $\tau_I(\ell) \neq \tau_J(\ell)$.

If $\tau_J(\ell) = \perp$, we can extend τ_J by $\tau_J(\ell) = \tau_I(\ell)$, which gives a larger profit contradicting the assumption that J is optimal; hence $\tau_J(\ell) \neq \perp$

If $\tau_I(\ell) = \perp$, the extended set $I \cup \{\tau_J(\ell)\}$ would be feasible, so the greedy algorithm would have added $\tau_I(\ell)$; hence $\tau_I(\ell) \neq \perp$

If $g_{\tau_I(\ell)} > g_{\tau_J(\ell)}$, we can replace $\tau_J(\ell)$ by $\tau_I(\ell)$ and increase the profit, which contradicts the optimality of J; hence $g_{\tau_I(\ell)} \leq g_{\tau_J(\ell)}$

If $g_{\tau_I(\ell)} < g_{\tau_J(\ell)}$, the greedy algorithm would have preferred $\tau_J(\ell)$ over $\tau_I(\ell)$, as $(I - \{\tau_I(\ell)\}) \cup \{\tau_J(\ell)\}$ is also feasible; hence $g_{\tau_I(\ell)} = g_{\tau_J(\ell)}$

This shows that I is already optimal

Greedy Scheduling Algorithm

The greedy strategy together with the optimality lemma give rise to an obvious greedy scheduling algorithm

We tacitly assume that $g_1 \geq g_2 \geq \cdots \geq g_n$ holds, so we only need the deadlines as input

We use a signature with two unary function symbols $\textsc{Deadline}$ and $\textsc{Sigma}$ plus two variables K and I

Initially we have $\textsc{Deadline}(0) = 0$, $\textsc{Deadline}(i) = \mathit{undef}$ for $i < 0$ or $i > n$, $\textsc{Sigma}(0) = 0$, $\textsc{Sigma}(1) = 1$, $\textsc{Sigma}(i) = \mathit{undef}$ for $i < 0$ or $i > 1$, and $I = 2$, $K = 1$

Then the following rule is iterated, and the final result is $\textsc{Sigma}$

Greedy Scheduling Rule

```
IF $I \leq n$
THEN LET $r_I \in \{0, \dots, K\}$ WITH
          $r_I = \max\{r \mid \textsc{Deadline}(\textsc{Sigma}(r)) \leq \textsc{Deadline}(I)\}$
    IN PAR
        IF $\textsc{Deadline}(I) > r_I$
        THEN PAR
            FORALL $\ell \in \{r_I + 1, \dots, K\}$
                DO $\textsc{Sigma}(\ell + 1) := \textsc{Sigma}(\ell)$ ENDDO
            $\textsc{Sigma}(r_I + 1) := I$
            $K := K + 1$
            ENDPAR
        ENDIF
        $I := I + 1$
    ENDPAR
ENDIF
```

Worst Case Complexity

Initially we have $I = 2$, finally $I = n + 1$, so the rule will be iterated n times

In each iteration step we scan Sigma (arguments from 1 to K) to determine r_I

Then we insert I in the correct position in Sigma and increment K, if this is possible for the deadline of I

This requires at most $2K$ steps, and in the worst case K runs from 1 to n

Putting these considerations together, the worst case time complexity of the above greedy scheduling algorithm is in $O(n^2)$

An Improved Algorithm

A set J with k jobs is feasible iff jobs $i \in J$ can be arranged one-by-one into a sequence $\sigma(1), \dots, \sigma(k)$ such that $\sigma(t) = i$ holds, where t is maximal with $0 \le t \le \min(n, d_i)$, when the job for t has not yet been decided.

Proof. Consider the "**if**" part, i.e. assume that a procedere with the given properties exists

If $i \in J$ is placed into position t_i, then we have $t_i \le \min(n, d_i)$, in particular $t_i \le d_i$ and $\sigma(t_i) = i$

Hence $t_i \le d_{\sigma(t_i)}$ holds for all $i \in J$, i.e. J is feasible by definition

For the "**only-if**" part assume that J is feasible

Assume $J = \{s_1, \dots, s_k\}$ with $d_{s_1} \le \cdots \le d_{s_k}$, then σ with $\sigma(i) = s_i$ is feasible

Proof / cont.

Proceed downwards from k to 1 and place s_j into the largest free position $\leq \min(n, s_{d_j})$

As σ is feasible, we have $j \leq d_{s_j}$ for all j, so the “worst case” occurs with s_k placed in position k and then each s_j placed in position j

Based on this lemma the idea of an **improved greedy scheduling algorithm** is as follows:

- Fill one-by-one the positions in a sequence of length $\ell = \min(n, \max_{1 \leq i \leq n} d_i)$
- For each position t define $n_t = \max\{k \leq t \mid \text{position } k \text{ is free}\}$
- Define an additional fictious position 0 that is always free

Greedy Scheduling Algorithm / cont.

- Keep **candidate sets** K of positions into which a job in J could be placed
  - Two positions $i, j \leq \ell$ are in the same set iff $n_i = n_j$ holds
  - So initially all candidate sets must be different
  - Let $F(K) = \min K$, so initially $F(\{i\}) = i$
- Use a union-find data structure to represent the candidate sets
- Process the jobs according to decreasing profit, i.e. assume $g_1 \geq g_2 \geq \cdots \geq g_n$

Greedy Scheduling Algorithm / cont.

- A job with deadline d is processed as follows:
  - Find the set K with $\min(n, d) \in K$
  - If $F(K) = 0$, the job is rejected, as it cannot be added to the sequence anymore
  - If $F(K) > 0$,
    - assign the job to position $F(K)$
    - let L be the set containing $F(K) - 1$; clearly we have $L \neq K$
    - merge L with K, then $F(L \cup K) = F(L)$
- If the final sequence contains gaps, compress it

We dispense with further formal details—see pg.99f. in the “Algorithmics” textbook

Correctness

The correctness of the algorithm follows from the fact that *in a candidate set K all positions except $F(K)$ have been filled*

Clearly, this holds initially, as $K = \{F(K)\}$

When K is merged with L, all positions of L are smaller than all positions in K

- As L contains $F(K) - 1$, all positions in L that are $\leq F(K) - 1$ are smaller than $F(K) = \min K$
- If L contained a position $> F(K) - 1$, the candidate sets would not be disjoint

Then the only free position $F(K)$ in K is filled, so $F(L)$ will be the only free position in $L \cup K$

Complexity

The algorithm essentially processes every of the n jobs in a loop

In each iteration we have to find candidate sets K and L and merge them—all other operations require constant time

That is, we need at most $n + \ell$ *find* operations and ℓ *merge* operations

As $n \geq \ell$ and we use a union-find data structure to represent the candidate sets, the complexity of the sequence of *find* and *merge* is in $O(n \log_2^* \ell)$ using the **iterated logarithm**

$$\log_2^* \ell = \min\{k \mid \underbrace{\log_2 \ldots \log_2}_{k \text{ times}} \ell \leq 0\}$$

Thus, if the set J of jobs is provided already in order by decreasing profit, the complexity of our improved scheduling algorithm is in $O(n \log_2^* \ell)$, which is almost linear

If the set J of jobs is not ordered, we obtain complexity in $O(n \log n)$, as J first has to be sorted

Huffman Coding

Suppose we have an alphabet A and we are looking for an encoding by bitstrings $cd(a) \in \{0,1\}^*$

We are particularly interest in codes satisfying the following conditions:

- No code $cd(a)$ is a prefix of another code $cd(b)$ (for $a, b \in A$)
- The bitstring $\mu 0$ is a code iff the bitstring $\mu 1$ is a code; in other words: either $\mu 0$ and $\mu 1$ are both codes or none of them

Such **prefix codes** can be represented by a tree:

- The leaves are labelled by the symbols $a \in A$
- The edges are labelled by 0 and 1; precisely: the edge to the left successor of any non-leaf node is labelled by 0, and the edge to the right successor by 1
- The labels of the path from the root to $a \in A$ define $cd(a)$

Huffman's Greedy Algorithm

Given any word $w \in A^*$ we want to find a prefix code for which the length of $cd(w)$ is minimal, where $cd(w) = cd(a_1) \dots cd(a_n)$ for $w = a_1 \dots a_n$

Equivalently, given frequencies $f_a \in [0, 1]$ for all $a \in A$—actually f_a is the probability for a random character to be a—we want to minimise the expected number of bits for $a \in A$

Huffman's algorithm works on forests of trees t as defined above, in each step merging two trees until only a single tree remains—the code defined by this tree is called a **Huffman code**

Each tree t has a weight $w(t)$, which is $\sum_{a \in t} f_a$ with the sum ranging over those $a \in A$ appearing as leaf in the tree

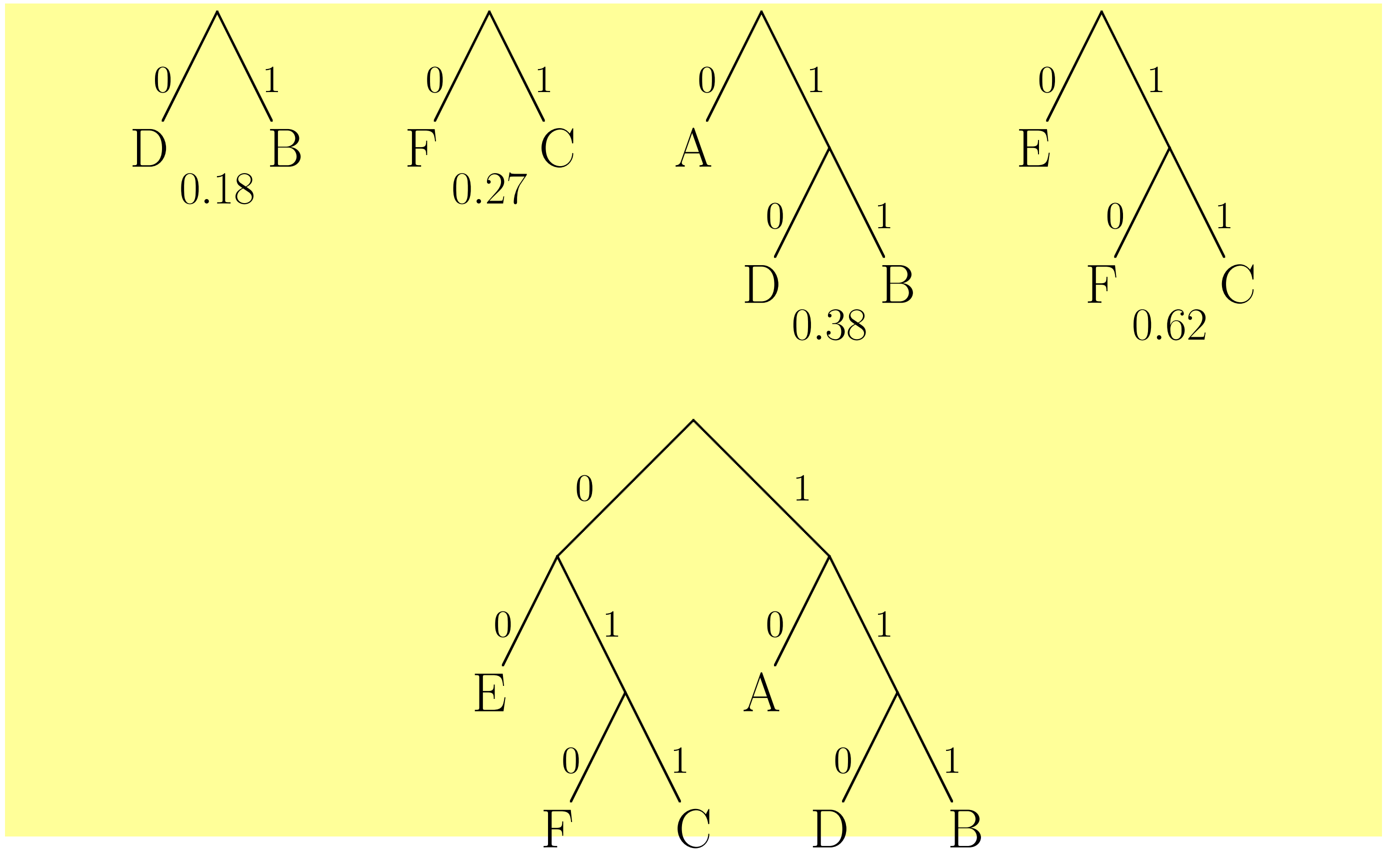
Initially, we have $n = |A|$ trivial trees with a single node labelled by $a \in A$, one for each symbol in A

The **greedy strategy** is to select in each step two trees t_1 and t_0 with smallest weights $w(t_1)$ and $w(t_0)$ and merge them into a new tree, in which t_0 and t_1 are the left and right successor trees of the new root, where $w(t_0) \geq w(t_1)$

Clearly, the weight of such a merged tree is $w(t_0) + w(t_1)$

Example

For the alphabet $\{A, B, C, D, E, F\}$ with frequencies $f_A = 0.2, f_B = 0.08, f_C = 0.12, f_D = 0.1, f_E = 0.35, f_F = 0.15$ the following trees will be constructed:



So the codewords are $cd(A) = 10, cd(B) = 111, cd(C) = 011, cd(D) = 110, cd(E) = 00, cd(F) = 010$

Optimality of Huffman Codes

Certainly, the length $\ell(x)$ of $cd(x)$ is a random variable with expectation $E(\ell) = \sum_{a \in A} f_a \ell(a)$

Let t_{huff} be the tree resulting from Huffman's greedy algorithm, and let t be any other tree representing a prefix code

We can create t out of t_{huff} by a sequence of operations of the following form:

Cut. Let $e \in A$ and take the subtree t' rooted at the parent of e, replace t' by the subtree rooted at the sibling of e

Insert. Replace any subtree t' by a tree having $e \in A$ and t' as successor trees, provided e does not appear in the tree

Without loss of generality we can assume that we always alternate cut and insert operations (with the same $e \in A$), i.e. we successively move elements $e \in A$ to a different position in the tree until we finally reach t

Upward and Downward Moves

Refer to a sequence of a cut and an insert with the same $e \in A$ as a *move* of e

- If the subtree t_1 in the cut is a subtree of the subtree t_2 in the insert, we have an **upward move**
- If the subtree t_1 in the cut contains the subtree t_2 of the insert, we have a **downward move**

Upward Moves. For an upward move consider the sequence of nodes from the parent of e before the cut to the new parent of e after the insert, let its length be $k+2$

Then the length of $cd(e)$ is decreased by k

All k nodes in the sequence without the first and last one have a successor tree with nodes not in the sequence; for all $a \in A$ appearing in one of these subtrees the operation increments $cd(a)$ by 1

Hence the expectation $E(\ell)$ changes by $\sum_{i=1}^{k} \sum_{a \in A_i} f_a - k f_e$, where A_i is the set of all nodes in the i'th of these subtrees

Upward and Downward Moves / cont.

For all $1 \leq i \leq k$ we must have $\sum_{a \in A_i} f_a \geq f_e$, otherwise e would have been chosen later by the algorithm

So an upward move increases the expected length of bitstrings

Downward Moves. The argument is analogous

For a downward move consider the sequence of nodes from the parent of e before the cut to the new parent of e after the insert, let its length be $k+2$

Then the length of $cd(e)$ is increased by k

All k nodes in the sequence without the first and last one have a successor tree with nodes not in the sequence; for all $a \in A$ appearing in one of these subtrees the operation decrements $cd(a)$ by 1

Hence the expectation $E(\ell)$ changes by $k f_e - \sum_{i=1}^{k} \sum_{a \in A_i} f_a$, where A_i is the set of all nodes in the i'th of these subtrees

Optimality / cont.

For all $1 \leq i \leq k$ we must have $\sum_{a \in A_i} f_a \leq f_e$, otherwise e would have been chosen earlier by the algorithm

So a downward move also increases the expected length of bitstrings

Arbitrary Moves. A move that is neither an upward nor a downward move moves e into a different subtree

So there is a node v such that e appeared in one successor tree before the move, and in the other after the move

In this case the effect of the move is the same as first moving e up beyond the node v followed by moving e down

Combining our findings for upward and downward moves we conclude that any move increases the expected length of bitstrings

Consequently, $E(\ell)$ is minimal for the tree t_{huff}

9.5 The Travelling Salesman Problem

Let us return once more to the travelling salesman problem (TSP)

Assuming a complete undirected graph $G = (V, E)$ with edges labelled by distances $d : E \to \mathbb{N}$ a straightforward greedy strategy works as follows:

Start with $E' = \emptyset$ and always select the shortest remaining edge e, i.e. $d(e)$ is minimal, such that

- $E' \cup \{e\}$ does not contain a cycle
- $(V, E' \cup \{e\})$ does not contain a vertex with degree 3

If these conditions are satisfied, add e to E'

We will obtain a Hamilton cycle, but the sum of edge labels may not necessarily be minimal

Example

Take a graph $G = (V, E)$ with the following distance matrix

$$\begin{pmatrix} 0 & 3 & 10 & 11 & 7 & 25 \\ 3 & 0 & 6 & 12 & 8 & 26 \\ 10 & 6 & 0 & 9 & 4 & 20 \\ 11 & 12 & 9 & 0 & 5 & 15 \\ 7 & 8 & 4 & 5 & 0 & 18 \\ 25 & 26 & 20 & 15 & 18 & 0 \end{pmatrix}$$

Then a greedy algorithm with the strategy above selects the edges in the order

$$\{1,2\}, \{3,5\}, \{4,5\}, \{2,3\}, \{4,6\}, \{1,6\}$$

which gives a complete tour $1 - 2 - 3 - 5 - 4 - 6 - 1$ with total costs 58

However, a truly optimal tour would be $1 - 2 - 3 - 6 - 4 - 5 - 1$ with total costs 56

Dynamic Programming Approach

As an optimal tour will contain some edge from node 1 to some node j, the remaining path from j to 1 must also be optimal, which motivates a dynamic programming solution

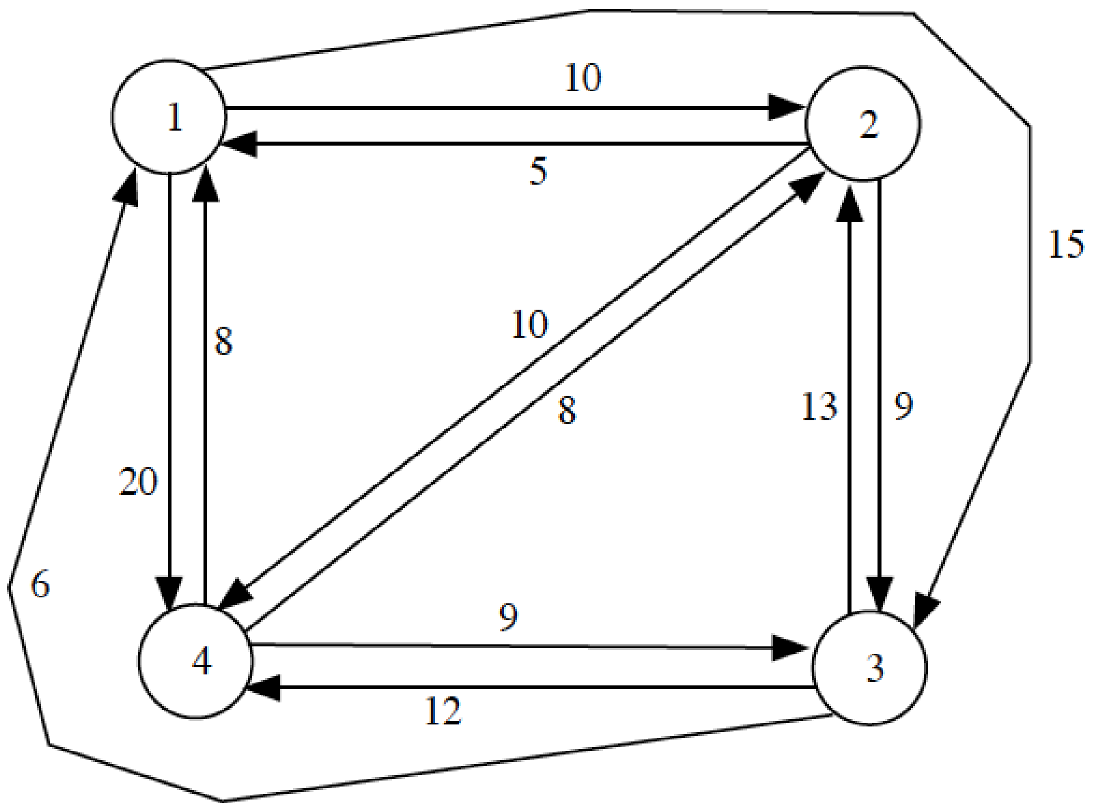
For this define $g(i, S)$ for a node i and a subset $S \subseteq V - \{i, 1\}$ as the length of a shortest path from i to 1, in which each node from S occurs exacly once—then $g(1, V - \{1\})$ is the length of the sought optimal tour

We have $g(i, S) \quad = \quad \begin{cases} \min_{j \in S}(c(i, j) + g(j, S - \{j\})) & \text{if } S \neq \emptyset \\ c(i, 1) & \text{if } S = \emptyset \end{cases}$

If we keep the index $j = ind(i, S)$ when building the minimum, then with $j_0 = j_n = 1$, $S_0 = V - \{1\}$, $ind(j_i, S_i) = j_{i+1}$ and $S_{i+1} = S_i - \{j_i\}$ the path $\{(j_0, j_1), (j_1, j_2), \dots, (j_{n-1}, j_n)\}$ is an optimal solution of the TSP

Example: TSP

Let us consider the following graph for the TSP:



Example / cont.

Then we can successively compute the following values for $g(i, S)$ and $ind(i, S)$:

$$g(2, \emptyset) = 5 \qquad g(3, \emptyset) = 6 \qquad g(4, \emptyset) = 8$$

$$\begin{array}{ll}
g(2, \{3\}) = 9 + 6 = 15 & ind(2, \{3\}) = 3 \\
g(2, \{4\}) = 10 + 8 = 18 & ind(2, \{4\}) = 4 \\
g(3, \{2\}) = 13 + 5 = 18 & ind(3, \{2\}) = 2 \\
g(3, \{4\}) = 12 + 8 = 20 & ind(3, \{4\}) = 4 \\
g(4, \{2\}) = 8 + 5 = 13 & ind(4, \{2\}) = 2 \\
g(4, \{3\}) = 9 + 6 = 15 & ind(4, \{3\}) = 3
\end{array}$$

Example / cont.

$$\begin{aligned} g(2,\{3,4\}) &= \min(9+20, 10+15) = 25 & \qquad ind(2,\{3,4\}) &= 4 \\ g(3,\{2,4\}) &= \min(13+18, 12+13) = 25 & ind(3,\{2,4\}) &= 4 \\ g(4,\{2,3\}) &= \min(8+15, 9+18) = 23 & ind(4,\{2,3\}) &= 2 \end{aligned}$$

and

$$g(1,\{2,3,4\}) = \min(10+25, 15+25, 20+23) = 35 \qquad ind(1,\{2,3,4\}) = 2$$

Then $E' = \{(1,2),(2,4),(4,3),(3,1)\}$ is an optimal solution of the TSP

Memory Functions

In an implementation of the above dynamic programming algorithm for TSP we need to compute values of the function g

$g(i, S)$ is recursively defined with cases for $S = \emptyset$ and $S \neq \emptyset$, so a recursive algorithm computing $g(i, S)$ is straightforward

This algorithm would compute intermediate results $distviaj(i, S, j) = c(i, j) + g(j, S - \{j\})$

However, a naive recursion like this will compute many values $g(j, S')$ repeatedly

A better strategy is to use a table $gtab(i, S)$ storing values of the function g, once they have been computed—such a function will be called a **memory function**

The recursive call in then modified to first check the stored memory function (initialised to -1 values to indicate undefinedness) and only call g, when $gtab(j, S') < 0$