PREFACE

These course notes cover some of the most basic and fundamental graph networking algorithms, such as graph searching, flow networks, matching algorithms, spanning trees, shortest paths, etcetera. Furthermore, a number of advanced data structures useful when implementing these algorithms is treated. Many of these notes are inspired by the book *Introduction to Algorithms*, written by T.H. Cormen, C.E. Leiserson and R.L. Rivest and published by MIT Press, 2001. Another interesting, but far less accessible, book is *Data Structures and Network Algorithms* by R.E. Tarjan published by CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.

Benny Van Houdt
January 2021

Contents

GRAPH SEARCHING

1 Graph representations

This chapter focuses on a number of elementary graph searching algorithms such as the Breadth-first and Depth-first search. As we will see, a number of fundamental graph problems can be solved using these simple algorithms. They are also used by various other more advanced algorithms to solve specific subproblems.

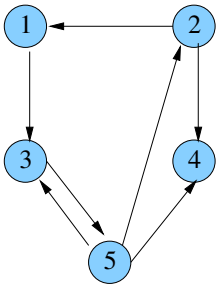
A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E. Each edge $e \in E$ represents an interconnection between two vertices u and v in V and is denoted as $e = (u, v)$. Edges can be **directed**, meaning going from one edge u to another v, or **undirected**, implying that (u, v) and (v, u) represent one and the the same edge. A graph's edges are either all directed or undirected and is termed directed in the first case and undirected in the latter.

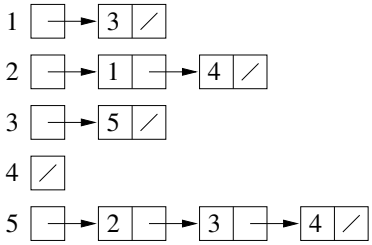
There are two important ways to store the info represented by (V, E). In the first, called the **adjacency matrix**, we use a $|V| \times |V|$ matrix, label the vertices 1 to $|V|$ and place a 1 on index (u, v) if $(u, v) \in E$ and a 0 otherwise. Notice, if G is undirected, its adjacency matrix is symmetric. Depending on whether we allow loops from a vertex u to itself, the main diagonal might contain nonzero entries. The advantage of this structure is that we can quickly check whether an edge (u, v) is part of G. However, the storage requirements are high, that is, $O(|V|^2)$, even though the number of nonzero entries might be low.

An alternative representation is given by the **adjacency list**. The adjacency list representation of a graph consists of $|V|$ lists, one for each vertex $u \in V$, which gives the vertices v to which u is adjacent (see Figure 1). For graphs with a fairly low number of edges, called sparse graphs, this results in a substantial gain in memory requirements, i.e., $O(|E|)$. However, checking whether an edge (u, v) is part of G takes more effort.

Exercises 1.1. *Graph representations:*

1. *Given an adjacency list representation of a directed graph $G = (V, E)$, give an $O(|V| + |E|)$ algorithm to obtain an adjacency list representation such that each of the $|V|$ lists are sorted.*





	1	2	3	4	5
1	0	0	1	0	0
2	1	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0
5	0	1	1	1	0

Fig. 1: Example of a directed graph, its adjacency list and matrix

2. *Let $G = (V, E)$ be a directed graph, give an $O(|V| + |E|)$ algorithm to obtain the graph $G' = (V, E')$ with $(u, v) \in E'$ if and only if $(v, u) \in E$ or $(u, v) \in E$.*

3. *Let A be the adjacency matrix representation of an* **undirected** *graph $G = (V, E)$. Let $(A^3)_{ij}$ be entry (i, j) of A^3. What does $\frac{1}{6}\sum_{i=1}^{|V|}(A^3)_{ii}$ represent?* ☆

4. *Consider the adjacency matrix representation A of a directed graph G. Define $v \in V$ as a universal sink if v has $|V| - 1$ incoming edges and no outgoing edges. Develop an $O(|V|)$ algorithm to determine whether G contains a universal sink.* ☆

5. *Give an $O(|V|^3)$ algorithm which determines for all pairs $u, v \in V$ the number of common neighbors of u and v in a directed graph $G = (V, E)$.* ☆

2 Breadth-first Search (BFS)

The Breadth-First Search (BFS) algorithm is one of the most fundamental and simple graph algorithms. Many famous algorithms, like the ones by Dijkstra and Prim, are very similar in nature. BFS algorithms are also used during the course while considering flow networks.

The idea is to select a specific vertex s, called the **source**, as the starting point of the search. Next, we discover each of the neighbors of s, say s_1 to s_k. Subsequently, the undiscovered neighbors of s_1 are discovered, followed by those of s_2 and so on. Each time we discover a vertex v we will draw an edge between v and u if u was the vertex from which we (first) discovered v. The vertex u will be termed the parent $\pi(v)$ of v. As we shall see, the

```
BFS(G,s):

for each vertex u ∈ V − {s}
    do color(u) = WHITE; d(u) = ∞; π(u) = NIL
color(s) = GRAY; d(s) = 0; π(s) = NIL
Q = {s}

while Q ≠ ∅
    do u = HEAD(Q)
       for each v ∈ Γ(u)
          do if color(v) = WHITE
             then color(v) = GRAY; d(v) = d(u) + 1; π(v) = u
                ENQUEUE(Q, v)
       DEQUEUE(Q)
       color(u) = BLACK
```

Fig. 2: The BFS algorithm

end result of this search is a single tree holding all vertices reachable from s, where each vertex v is connected to its parent $\pi(v)$.

The BFS algorithm is typically implemented using an adjacency list. In order to distinguish between discovered and undiscovered vertices, each vertex u gets a color $color(u) \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$. WHITE vertices are considered undiscovered, once discovered a vertex turns GRAY. Further all vertices u get a pointer $\pi(u)$ to identify its parent. For the source s, this pointer is set to NIL. Finally, a queue Q is also used to determine the next vertex v from which we try to discover new vertices, given that all of the neighbors of the current vertex u have been discovered.

The piece of pseudo code in Figure 2 gives a detailed description of the inner workings of the BFS. Notice, the variable $d(u)$ keeps track of the number of vertices needed to get from u to the source vertex s by following its line of ancestors. The operation HEAD returns the element at the head of the queue, while ENQUEUE and DEQUEUE will add and remove a vertex to the back and from the front of the queue, respectively. Finally, $\Gamma(u)$ represents the set of neighbors of u (these are the vertices part of the adjacency list of u). An example of a breath-first search on a graph G with 9 nodes is given in Figure 3.

The runtime of this algorithm is easy to assess. Clearly, the initialization phase requires $O(|V|)$ time. Afterwards, for each u that is at the head of

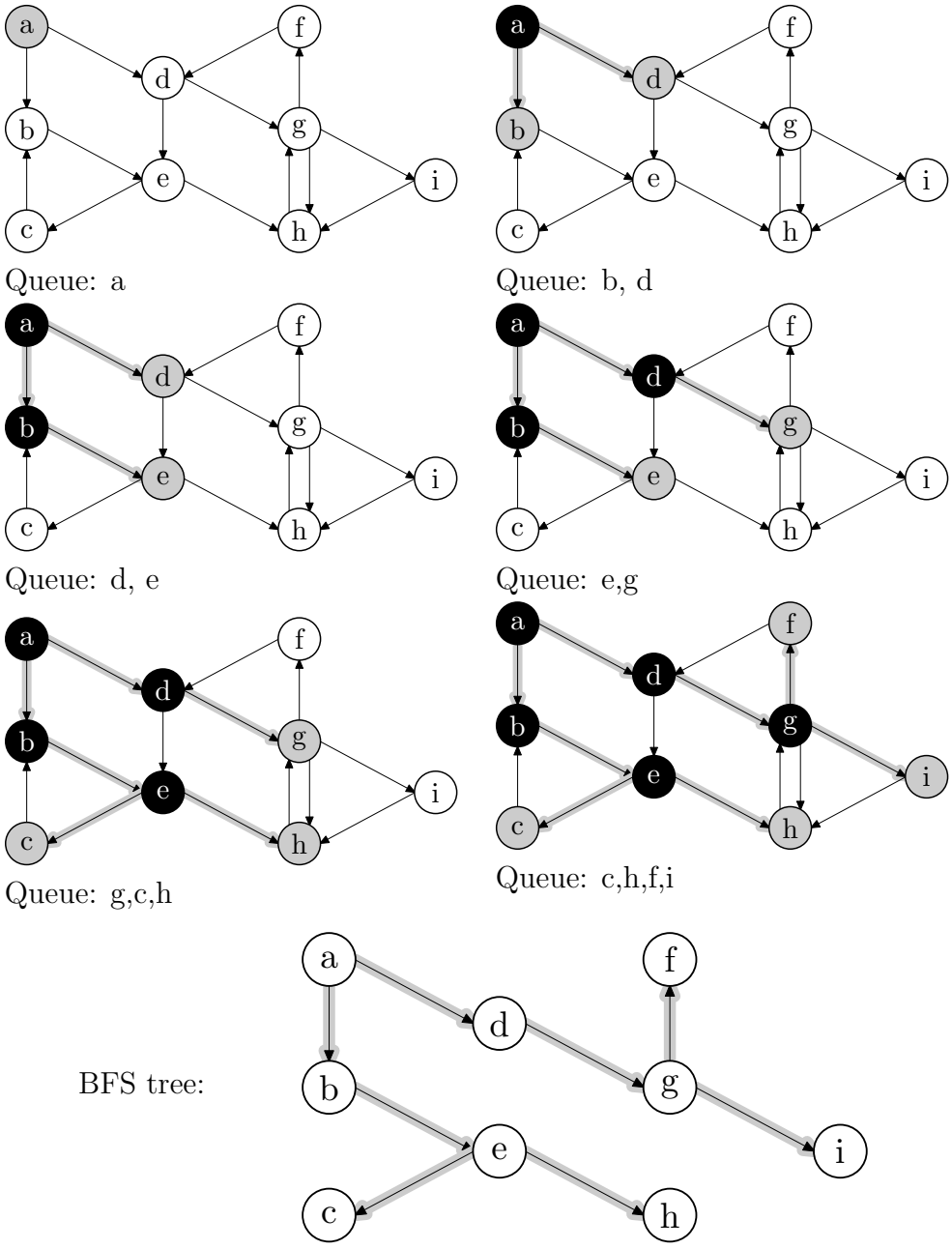


Fig. 3: Example of Breath-first Search (BFS).

the queue, we scan its adjacency list and check for each neighbor v whether it is still undiscovered (i.e., WHITE). If so, we set a few of the parameters of v and add it to the back of the queue Q. A vertex is only queued when being

discovered, hence, each vertex is added to the back of the queue Q just once. Meaning we only need to scan the neighbors of each vertex once. Therefore, the total runtime is $O(|V| + |E|)$.

Theorem 2.1. *Given $G = (V, E)$ a graph with $s \in V$ the source vertex, the BFS algorithm returns a tree containing all vertices reachable from s and $d(u) = \delta(s, u)$ upon termination, where $\delta(s, u)$ gives the length of the shortest simple path from s to u in G (that is, the number of edges needed to get from s to u).*

Proof. To prove this theorem we start by arguing that the following three statements hold:

1. If $d(u) = k$, vertex u is at level k of the BFS tree and $\delta(s, u) \leq k = d(u)$.
2. If $Q = v_1 v_2 \dots v_r$ during the algorithm, we have $d(v_1) \leq d(v_2) \leq \dots \leq d(v_r) \leq d(v_1) + 1$.
3. If v is dequeued before u, we have $d(v) \leq d(u)$.

The first statement is trivial. The second holds by noting that $Q = s$ at the start and whenever we add a vertex v to Q when u is at the head of the queue, we have $d(v) = d(u) + 1$. The last statement follows immediately from statement 2.

The proof completes as follows. Assume the theorem does not hold and let v be a vertex for which $d(v) > \delta(s, v)$ with $\delta(s, v)$ as small as possible. Note, by statement 1 it is impossible to have $d(v) < \delta(s, v)$. Let u be the last vertex on a shortest path from s to v. As $\delta(s, v)$ was chosen as small as possible we have $d(u) = \delta(s, u) = \delta(s, v) - 1$.

When u is dequeued there are two options. Either v was dequeued earlier and we have $d(v) \leq d(u)$ (by statement 3). This yields $d(v) \leq \delta(s, u) \leq \delta(s, v)$. Otherwise as $(u, v) \in E$, the vertex v must be in the queue when u is dequeued. Hence, by statement 2 we have $d(v) \leq d(u) + 1 = \delta(s, v)$. Thus, in both cases we have $d(v) \leq \delta(s, v)$ which contradicts the assumption that the theorem does not hold. □

The **predecessor** tree $T \subseteq G = (V, E)$ is defined as $T = (V_\pi \cup \{s\}, E_\pi)$ with $v \in V_\pi$ if $\pi(v) \neq \text{Nil}$ and $(u, v) \in E_\pi \subseteq E$ if $u = \pi(v)$. The edges $e \in E_\pi$ are called tree edges.

Exercises 2.1. *BFS searches:*

1. *Determine the BFS tree of the graph depicted in Figure 3 if we reverse the order of the vertices in the adjacency lists of a and d.*

2. *Give an example of a graph G and source vertex s such that $E' \subseteq E$ forms a tree on V such that the unique path from s to v, for all $v \in V$, has length $\delta(s, v)$, but that cannot be produced as the predecessor tree by the BFS algorithm, no matter how the vertices were sorted in the adjacency lists.*

3. *A graph G is bipartite if V can be partitioned into V_1 and V_2 such that if $(u, v) \in E$, then $u \in V_1$ and $v \in V_2$ or vice versa. Give an efficient algorithm to test whether G is bipartite.*

4. *Let the diameter of a graph $G = (V, E)$ be $\max_{u,v \in V} \delta(u, v)$ (for those u, v couples with $\delta(u, v)$ finite). Give an algorithm to determine the diameter, what is the time complexity of your algorithm?*

5. *Give an $O(|V|)$ algorithm to find the diameter in an undirected acyclic connected graph?* ☆

6. *We have three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container. Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered. What algorithm should be applied to solve the problem?* ☆

7. *Consider a graph $G = (V, E)$ and two sets $V_1, V_2 \subset V$ with $V_1 \cap V_2 = \emptyset$. Give an $O(|V| + |E|)$ algorithm to find $\delta(V_1, V_2) = \min_{u \in V_1, v \in V_2} \delta(u, v)$.* ☆

8. *Adapt the BFS algorithm such that it not only determines the shortest path length $d(v)$ for all the nodes v reachable from s, but also the number of paths $M(v)$ of length $d(v)$ between s and v.* ☆

3 The Depth-first search (DFS)

The depth-first search (DFS) strategy differs from the BFS, in the sense that DFS tries to search *deeper* into the graph before exploring the breadth. As with the BFS algorithm, vertices are initially undiscovered and thus colored WHITE (except for the source node s in BFS). Furthermore, each vertex u is

```
DFS(G):

for each vertex u ∈ V
    do color(u) = WHITE; π(u) = NIL
time = 0
for each vertex u ∈ V
    do if color(u) = WHITE
       then DFS-VISIT(u)

DFS-VISIT(u):

color(u) = GRAY; time = time + 1; d(u) = time;
for each v ∈ Γ(u)
    do if color(v) = WHITE
       then π(v) = u; DFS-VISIT(v)
color(u) = BLACK; time = time + 1; f(u) = time
```

Fig. 4: The DFS algorithm

also linked to its parent upon discovery, where the parent node $\pi(u)$ is the node from which u is discovered. An important distinction with the BFS is that the DFS will produce a set of trees as its output (instead of a single BFS tree holding the shortest paths from the source s to all the vertices reachable from s).

Once a vertex u has been discovered it turns GRAY and once all of its neighbors have been discovered it will become BLACK. For each vertex we will keep track of two important parameters, that is, timestamps: $d(u)$ and $f(u)$. Initially, the variable *time* is initialized to 0 and each time a node turns GRAY (that is, is discovered) or turns BLACK (that is, is finished), we increase *time* by one. Upon discovery of a vertex u, we set $d(u) = time$ and similarly, when it is finished, we set $f(u) = time$. Thus, $d(u)$ and $f(u)$ indicate when a vertex u was discovered and finished with respect to the discovery and finishing times of other vertices. Note, $1 \leq d(u) < f(u) \leq 2|V|$ for all $u \in V$. These two parameters are often exploited by other more advanced algorithms.

A pseudo code version of the DFS algorithm is given in Figure 4, while Figure 5 gives an example of the DFS algorithm on a graph with 9 vertices. As can be seen from this Figure, the DFS algorithm has a recursive nature. As with the BFS algorithm, the **predecessor** or DFS tree/forest $T \subset G$ is

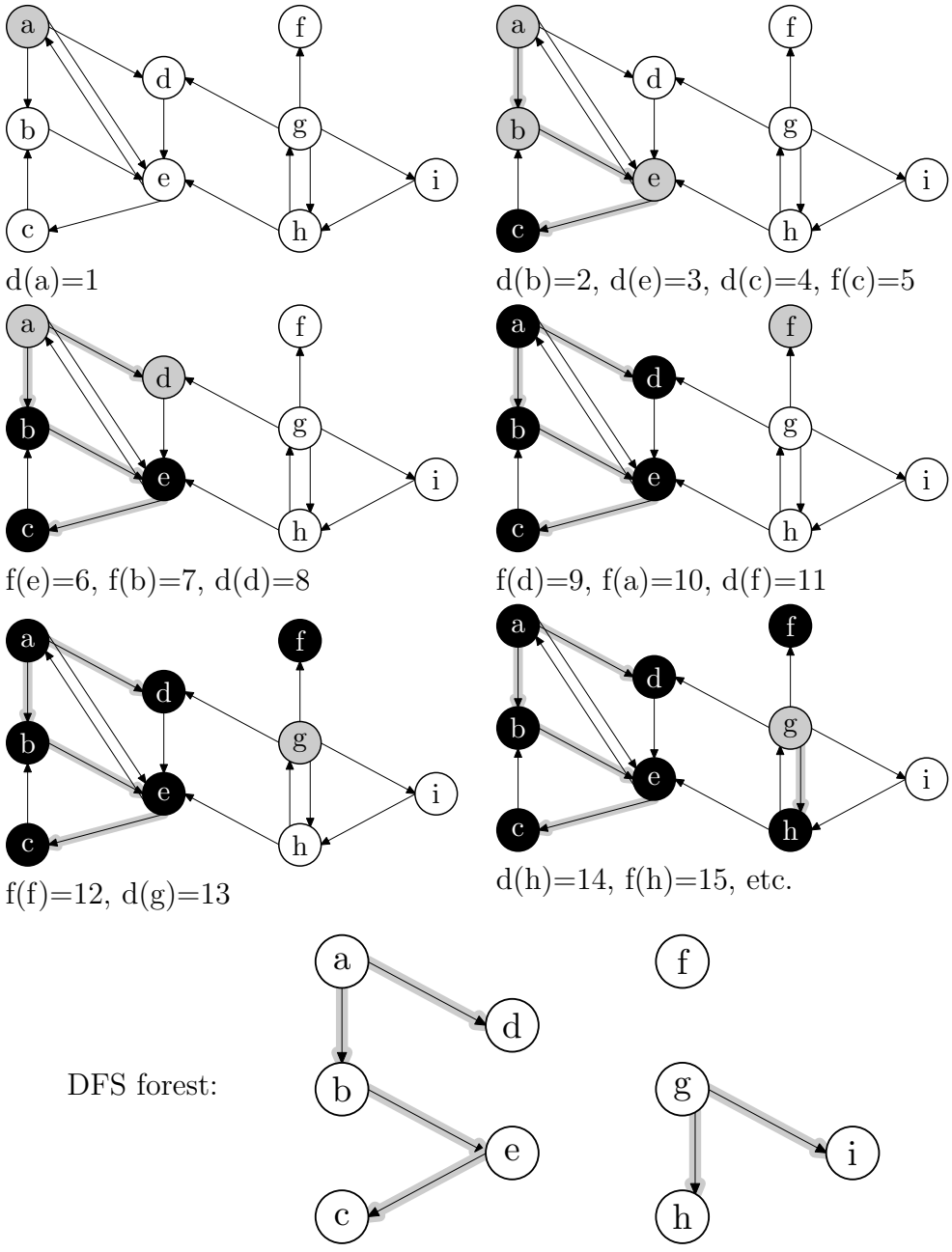


Fig. 5: Example of Depth-first Search (DFS).

defined as $T = (V, E_\pi)$ with $(u, v) \in E_\pi$ if $u = \pi(v)$. The edges $e \in E_\pi$ are called tree edges. However, as opposed to the BFS algorithm E_π can be a collection of trees, often termed a forest.

Let us evaluate the runtime of the DFS algorithm. The initialization phase clearly takes $O(|V|)$ time. If we exclude the time required for the DSF-Visit calls, the entire DFS routine requires only $O(|V|)$ time. A call to DFS-Visit can only occur on a White vertex and the first thing DFS-Visit does is color it Gray, after which it will never become White again. The **for** loop in DFS-Visit(u) is executed at most $|\Gamma(u)|$ times, leading to an overall cost of $O(|V| + |E|)$.

We now prove two important properties of the DFS search:

Theorem 3.1 (Parenthesis theorem)**.** *Given $G = (V, E)$ and $u, v \in V$, then either of the following two cases holds after running the DFS algorithm:*

1. *The intervals $[d(u), f(u)]$ and $[d(v), f(v)]$ do not overlap and v is not a descendant of u.*
2. *The interval $[d(u), f(u)]$ completely engulfs the interval $[d(v), f(v)]$ and v is a descendant of u in the DFS tree/forest (or vice versa with the role of u and v switched).*

Thus, v is a descendant of u if and only if $d(u) < d(v) < f(v) < f(u)$.

Proof. Without loss of generality assume $d(u) < d(v)$. If $f(u) < d(v)$ then clearly both intervals are disjoint (there is no overlap) as $d(v) < f(v)$ and v is not a descendant of u in the DFS tree, because the descendants are exactly those vertices that are discovered while u is Gray. If, on the other hand, $d(v) < f(u)$, we find that v is discovered while u is still Gray. Due to the recursive nature, v will first explore all its neighbors (and turn Black) before u can be finished; hence, $f(v) < f(u)$ as required. □

Theorem 3.2 (White path theorem)**.** *Given that we perform a DFS search on $G = (V, E)$ with $u, v \in V$, v will be a descendant of u in the DFS tree if and only if at time $d(u)$ there exists a* White *path from u to v in G.*

Proof. $\Rightarrow$) Assume v is a descendant of u in the DFS tree and w is a vertex on the path from u to v in this tree (and is therefore also a descendant of u). By Theorem 3.1, we have $d(u) < d(w)$ for all such vertices w, meaning at time $d(u)$, w is still White.
$\Leftarrow$) Assume we have a white path from u to v at time $d(u)$, but v is not a descendant of u. Further, we can select v such that w, the vertex just prior to v on this path, is a descendant of u in the DFS tree. Now, due to Theorem 3.1, we have

$$f(w) < f(u)$$

and as v is WHITE at time $d(u)$, we have

$$d(u) < d(v).$$

Now, either v has been discovered before w explores the edge (w, v) or it is discovered at this time and thus before w finishes; hence,

$$d(v) < f(w) < f(u).$$

Combining the last two inequalities we see that v is after all a descendant of u due to Theorem 3.1. □

In many cases it helps to classify the edges E of the graph $G = (V, E)$ that is traversed by the DFS:

1. Tree edges are those edges $(u, v) \in E$ that are also part of the depth-first forest described by E_π.
2. Back edges $(u, v) \in E$ connect a vertex u with one of its ancestors in the DFS tree/forest (i.e., u is a descendant of v).
3. Forward edges $(u, v) \in E$ are those connecting a vertex u with one of its descendants v in the DFS tree.
4. Finally, cross edges are all the remaining edges. These can be edges in a single DFS tree or edges between different trees in the DFS forest.

While executing the DFS algorithm we can easily classify all the edges as follows: when exploring (u, v), we encounter a tree edge if v is WHITE, a back edge if v is GRAY and a forward or cross edge when v is BLACK. We can distinguish between a forward and cross edge by comparing $d(u)$ and $d(v)$. If $d(u) < d(v)$ it is a forward edge and otherwise it is a cross edge. To understand this first note that the parenthesis theorem implies that

- $[d(u), f(u)] \subset [d(v), f(v)]$ for a back edge (u, v),
- $[d(v), f(v)] \subset [d(u), f(u)]$ for a forward edge (u, v) and
- $[d(v), f(v)]$ and $[d(u), f(u)]$ are disjoint for a cross edge (u, v).

Hence, for a back edge (u, v) is explored in $[d(v), f(v)]$, meaning v is GRAY when exploring (u, v). For a forward edge v is not discovered by exploring the edge (u, v), but by exploring some other edge to v (otherwise (u, v) is a tree edge). The recursive nature of the algorithm therefore implies that v is finished before (u, v) is explored. As such v is BLACK with $d(u) < d(v)$ for

a forward edge. Finally, for a cross edge we must have $d(v) < f(v) < d(u) < f(u)$ as $(u, v) \in E$ and both $[d(v), f(v)]$ and $[d(u), f(u)]$ must be disjoint. This implies that v is BLACK with $d(u) > d(v)$ for a cross edge.

In an undirected graph an edge is classified according to the *first* time it is encountered. When the original graph G is undirected we can only encounter tree and back edges as proven below:

THEOREM 3.3. *A DFS search of an undirected graph $G = (V, E)$ only encounters tree and back edges.*

Proof. Let (u, v) be an arbitrary edge in E. Suppose that v is BLACK when exploring the edge (u, v) from u. This implies that v has been discovered and all its outgoing edges have been explored. Thus, as $(u, v) \in E$, so is the edge (v, u) and this edge must have been explored. At this time u was either WHITE or GRAY (as it is still GRAY) and hence the edge is either a tree or back edge. □

For a directed graph G we can easily check whether it contains a cycle as explained below. Directed graphs without cycles are called directed acyclic graphs or in short **dags**.

THEOREM 3.4. *A directed graph G contains no cycles if a DFS traversal of the graph does not encounter a back edge.*

Proof. Clearly, if the DFS tree contains a back edge, then G contains a cycle. The reverse statement can be proven as follows. Assume C is a cycle with vertices $u_1, \ldots, u_k$ in G and without loss of generality, let u_1 be the first vertex on G that is discovered by the DFS search. Then, by the white path theorem, the entire cycle is part of the DFS tree of u_1, including u_k. When we we visit u_k, we will explore the edge (u_k, u_1) and this edge is a back edge as u_1 was already discovered at that time (but not finished as u_k is a descendant of u_1). □

EXERCISES 3.1. *DFS searches:*

1. *Show that the following conjecture is false:* If there is a path from u to v in a directed graph G and $d(u) < d(v)$ in a DFS search of G, then v is a descendant of u.

2. *Give an example of a graph G such that a DFS search results in a forest of DFS trees where one of the trees contains only one vertex u that has both incoming and outgoing edges in G.*

3. *Argue that a DFS on an undirected graph will result in a DFS forest where each tree is a connected component of G. Also, modify the DFS algorithm such that each vertex has a variable $cc(u)$ that indicates to which component it belongs.*

4. *Give an $O(|V|)$ algorithm to check whether an undirected graph contains a cycle.*

5. *Give an algorithm to test whether a selected vertex s is singly connected to all other vertices, that is, check whether there is at most one simple path from s to any other vertex $u \in V$. Use this algorithm to develop a $O(|V||E|)$ algorithm to check whether a graph is singly connected, meaning there is at most one simple path between any two vertices u and v.*

6. *Give an example of a graph G such that the DFS algorithm can produce k output trees for any $k \in \{1, \ldots, |V|\}$.* ☆

7. *An undirected connected graph $G = (V, E)$ is biconnected if G remains connected after removing any one of its vertices. Give an $O(|V|^2 + |V||E|)$ algorithm to check whether G is biconnected. Show that the removal of v, the root of a DFS tree, disconnects the graph if and only if v has 2 or more outgoing tree edges.* ☆

8. *Let b be the number of back edges encountered during the execution of the DFS algorithm. Show that in general b is not equal to the number of cycles in the graph.* ☆

9. *True or false?* ☆

   (a) *If one output of the DFS algorithm contains a back edge, all outputs contain a back edge.*

   (b) *If one output of the DFS algorithm contains a forward edge, all outputs contain a forward edge.*

   (c) *If one output of the DFS algorithm contains a cross edge, all outputs contain a cross edge.*

4 Topological sort

Directed acyclic graphs can be sorted topologically using the DFS algorithm. We state that G is topologically sorted if for all $(u, v) \in E$, vertex u must

appear before vertex v. Clearly, when G contains cycles such a sort is impossible.

To perform a topological sort we simply run the DFS algorithm on G and each time a vertex is finished, we add it as the first element of a linked list. Initially, the list is empty, as more and more vertices are finished, the list grows until it has size $|V|$ and all vertices are finished. The following property shows that this approach sorts G topologically.

Theorem 4.1. *A directed acyclic graph G can be sorted topologically by calling the DFS algorithm and adding the vertices in the order they are finished to the front of a linked list.*

Proof. Suppose we run the DFS algorithm on G and $f(u)$ is the finishing time of a vertex $u \in V$. Thus, if $f(v) < f(u)$, u appears before v in the linked list. Assume that $(u, v) \in E$, we need to show that $f(v) < f(u)$. At some point we explore this edge from u (which is Gray). At this time v cannot be Gray as this would mean that (u, v) is a back edge (contradicting Theorem 3.4). If v is White, it is a descendant of u with $d(u) < d(v)$, implying $f(v) < f(u)$ as required (due to the parenthesis Theorem). If v is Black then $f(v) < f(u)$ as u is still Gray. □

Exercises 4.1. *Topological searches:*

1. *Let t be a node with out-degree 0 in a dag $G = (V, E)$. Give a $O(|V| + |E|)$ algorithm to determine the number of paths from each vertex $u \in V - \{t\}$ to t.*

2. *Let s and t be two vertices in a dag G such that the in-degree of s and the out-degree of t is 0. Use the result of the previous exercise to determine, for each edge $(u, v) \in E$, the number of paths from s to t that run through (u, v) and this in $O(|V| + |E|)$ time.*

3. *One can also sort a dag G by repeatedly selecting a vertex u with in-degree 0 and removing all its outgoing edges from G. How do you implement this in $O(|V| + |E|)$ time?*

4. *Suppose that we execute the topological sort on a graph G that is not acyclic in order to sort the vertices of G. Show that the number of edges pointing in the wrong direction is not necessarily minimal using a graph with 4 vertices.* ☆

5. *Argue that every acyclic graph contains at least one vertex with zero outgoing edges.* ☆

6. *Given a weighted acyclic graph $G = (V, E)$. Give an $O(|V| + |E|)$ algorithm to find a path with maximum weight in G.* ☆

5 Strongly Connected Components

The vertices V of a graph $G = (V, E)$ can always be partitioned into k subsets $C_1, \ldots, C_k$ (for some $k \geq 1$) such that two vertices $u, v \in V$ belong to the same subset if and only if there is a path from u to v and from v to u in G. These k subsets are called the strongly connected components (SCCs) of the graph G. Notice, if u and v belong to the same component C, any node on a path from u to v (or from v to u) also belongs to C.

The following algorithm, that relies on two DFS searches, computes the SCCs of a directed graph G.

1. Perform a DFS on G and denote the finishing time of a node u as $f(u)$
2. Construct the graph $G' = (V, E')$, where $(u, v) \in E'$ if and only if $(v, u) \in E$
3. Perform a DFS search on G', where the nodes of G' in the main for-loop of the DFS algorithm are ordered in decreasing value of $f(u)$

Each tree produced by the second DFS will correspond to a SCC, an example is given in Figure 6.

Theorem 5.1. *The algorithm above correctly computes the SCCs of a graph G*

Proof. We start by defining $f(C) = \max_{u \in C} f(u)$ and $d(C) = \min_{u \in C} d(u)$ as the finishing time and discovery time of an SCC C during the first DFS, respectively. We first show that if $(u, v) \in E$, $u \in C$ and $v \in C'$, where C and C' are two different SCCs, then $f(C) > f(C')$.

We distinguish two cases. First, assume $d(C) < d(C')$ and let x be the first node of C that is discovered. Thus, at time $d(x)$ all the nodes in C and C' are still White. Further, as $(u, v) \in E$, all the nodes in C and C' are reachable from x, thus, due to the white path theorem, all the nodes in $C \cup C'$ are descendants of x. This implies that $f(x) = f(C) > f(C')$. In the second case, $d(C') < d(C)$ and by the white path theorem $f(x) = f(C')$, where x is the first node of C' that is discovered. Further, as $(u, v) \in E$, none of the vertices in C are reachable from x as this would imply that all the nodes in C and C' belong to the same SCC. Hence, $f(C) > f(C')$ as required. This implies that if $(u, v) \in E'$, with u and v belonging to two different SCCs C and C', then $f(C) < f(C')$.

Now, assume by induction that the first m trees produced by the second DFS search correspond to the first m SCCs C_1 to C_m such that $f(C_1) > \ldots > f(C_m)$. Denote x as the root node of the next tree produced by the

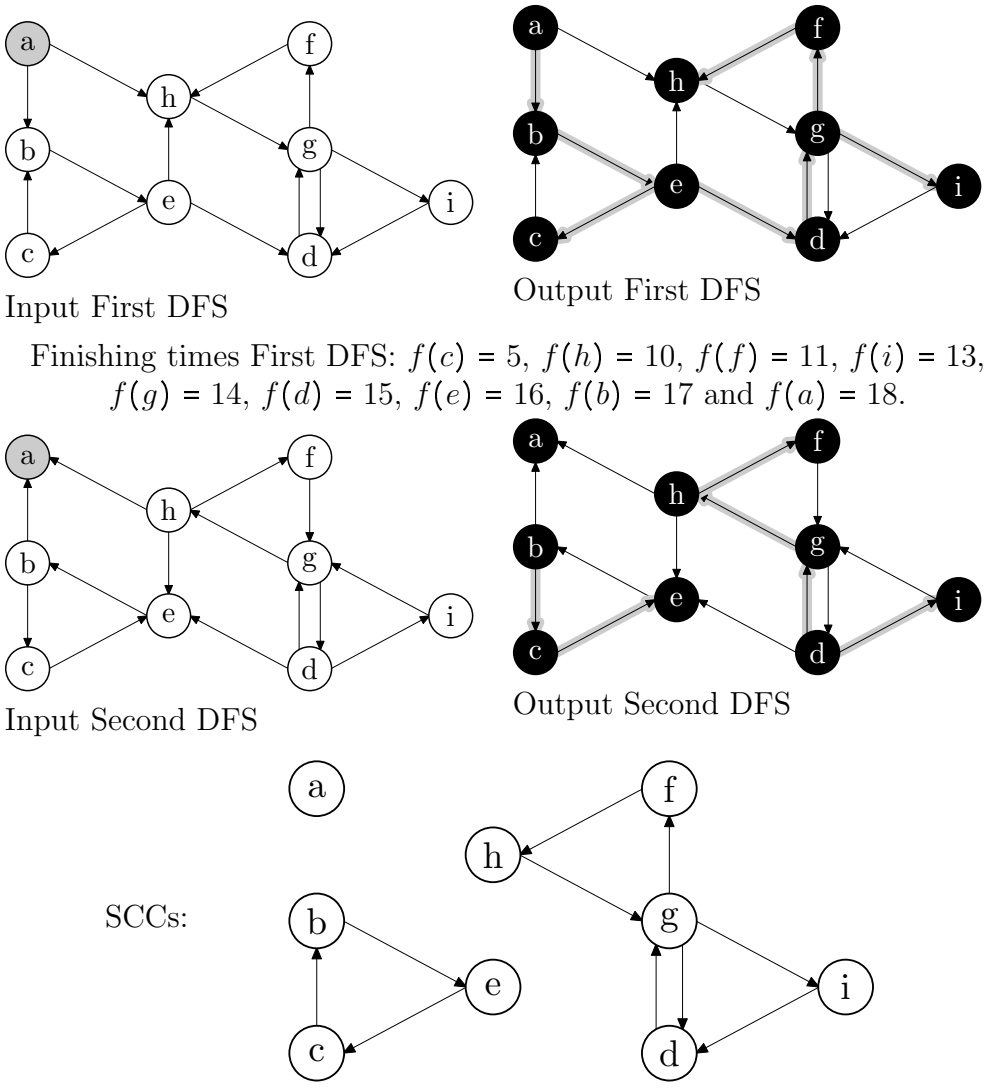


Fig. 6: Example of Strongly Connected Components algorithm.

second DFS and denote C as its SCC. Clearly, all the nodes in C will be part of the tree rooted by x (as they do not belong to $C_1 \cup \ldots \cup C_m$ by induction). Further, as the vertices of G' in the main for-loop of the second DFS algorithm are ordered in decreasing value of $f(u)$, we have $f(x) = f(C)$.

Next, suppose there exists an edge $(u, v) \in E'$, with $u \in C$ and $v \notin C_1 \cup \ldots \cup C_m \cup C$, but part of SCC C'. Let $y \in C'$ be the vertex with $f(y) = f(C')$. Then, $f(x) = f(C) < f(C') = f(y)$ as shown earlier, but this is impossible as $f(x) > f(y)$ (as the nodes in the main for-loop of the second DFS were ordered in decreasing value of $f(u)$). As a result, tree $m+1$

will consist of the nodes of C only and therefore corresponds to the $m+1$-st SCC. □

The overall time complexity of this algorithm is $O(|V| + |E|)$ as the adjacency list of the graph G' can also be constructed in linear time. The above algorithm has been attributed to some unpublished work by S.R. Kosaraju (by Aho, Hopcroft and Ullman). Other linear time algorithms that avoid the construction of G' are by Tarjan and Gabow.

Exercises 5.1. *Strongly Connected Components:*

1. *What is the maximum number of edges in a directed graph $G = (V, E)$ that has exactly two SCCs?* ☆
2. *Determine the minimum number of edges $|E|$ in a graph $G = (V, E)$ given that it has $k \in \{2, \ldots, |V|\}$ strongly connected components.* ☆
3. *Show that replacing an edge (u, v) by (v, u) can increase or decrease the number of SCCs in a graph by more than one.* ☆
4. *Assume $G = (V, E)$ has k SCCs with $|V(C_i)|$ vertices in component C_i. What is the smallest possible value for $|E|$?* ☆

FLOW NETWORKS

1 Definitions and basic properties

We start with some basic definitions.

Definition 1.1. *A flow network $G = (V, E)$ is a* directed *graph in which each edge $(u, v) \in E$ has a capacity $c(u, v) \geq 0$. Let s, t be two specific vertices in V called the* **source** *and* **sink** *of the network. If $(u, v) \notin E$, we define $c(u, v) = 0$.*

To simplify our discussion, we assume that for each $v \in V$, there exists a path from s via v to t. Nodes in V for which there is no such path have no use as they cannot carry any flow from s to t.

Definition 1.2. *A flow f in a flow network G, is a real-valued function f from $V \times V$ to $\mathbb{R}$ satisfying:*

1. *Capacity constraint: $f(u, v) \leq c(u, v)$ for all $u, v \in V$.*
2. *Skew symmetry: $f(u, v) = -f(v, u)$ for all $u, v \in V$.*
3. *Flow conservation: $\sum_{v \in V} f(u, v) = 0$ for all $u \in V - \{s, t\}$.*

The value of $f(u, v)$ is the **net** *flow from u to v and the* **value** *of f, denoted as $|f|$, equals $\sum_{v \in V} f(s, v)$, where s is the source vertex.*

The first constraint is quite natural, we do not wish to have more flow on a link than its capacity allows. The second requirement states that the flow from u to v must equal minus the flow from v to u. This might seem odd at first as we might have a positive capacity in both directions. However, $f(u, v)$ gives the net flow. Thus if we have a flow of $g(u, v) > 0$ from u to v and a flow $g(v, u) > 0$ from v to u, then the net flow $f(u, v) = g(u, v) - g(v, u)$ and $f(v, u) = g(v, u) - g(u, v)$; hence, $f(u, v) + f(v, u) = 0$ as required. The third property indicates that apart from the source and sink vertex, all flow entering a vertex v, should also leave it (this is equivalent to the circuit laws of Gustav Kirchhoff in physics: all current entering a vertex, must exit it).

The value $|f|$ of a flow f represents the amount of flow leaving the source s. Given the flow conservation property, it is intuitively clear that $|f|$ also equals the amount of flow entering the sink t (a formal proof is given in Lemma 2.2). Our interest in flow networks exists in finding a flow f such that its value $|f|$ is maximum, this problem is known as the **maximum flow problem**. The following lemma will help in constructing a method to determine the maximum flow f of a flow network G.

Lemma 1.1. *Let f_1 and f_2 be two flows in G with $f_1(u,v)+f_2(u,v) \leq c(u,v)$ for all $u, v \in V$, then f defined as $f(u,v) = f_1(u,v)+f_2(u,v)$ for all $u, v \in V$, defines a flow in G with $|f| = |f_1| + |f_2|$.*

Proof. Evidently, the capacity constraint is fulfilled as $f_1(u,v) + f_2(u,v) \leq c(u,v)$ for all $u, v \in V$. Moreover,

$$f(u,v) = f_1(u,v) + f_2(u,v) = -f_1(v,u) - f_2(v,u) = -f(v,u)$$

and

$$\sum_{v \in V} f(u,v) = \sum_{v \in V} f_1(u,v) + \sum_{v \in V} f_2(u,v) = 0 + 0 = 0,$$

for $u \in V - \{s,t\}$. Further, we have

$$|f| = \sum_{v \in V} f(s,v) = \sum_{v \in V} f_1(s,v) + \sum_{v \in V} f_2(s,v) = |f_1| + |f_2|.$$

□

Thus, adding two flows, creates a new flow, provided that the capacity constraints remain valid. Before we move on to residual networks, let us first focus on a network flow problem with **multiple sources** $s_1, \ldots, s_k$ and **multiple sinks** $t_1, \ldots, t_l$. The maximum flow problem of such a network tries to find the maximum flow from the set $\{s_1, \ldots, s_k\}$ to $\{t_1, \ldots, t_l\}$, such that the capacity is respected and there is flow conservation. We can easily reduce this problem to a single source and sink problem as follows.

Let G be the original flow network, define $G^+ = (V^+, E^+)$ with $V^+ = V \cup \{s,t\}$, $E^+ = E \cup \{(s,s_i) | i = 1, \ldots, k\} \cup \{(t_j,t) | j = 1, \ldots, l\}$ and let $c(s,s_i) = c(t_j,t) = \infty$ for $i = 1, \ldots, k$ and $j = 1, \ldots, l$. It should be clear that a maximum flow f in G^+ corresponds to a maximum flow in the multiple source, sink setting and vice versa.

Exercises 1.1. *Flow networks:*

1. *Assume that the vertices $v \in V$ also have a capacity $c(v) \geq 0$ and assume that we demand that the amount of flow entering v is limited by $c(v)$. Show that we can reduce this more general flow network problem to a standard flow network problem without vertex capacities.*

2. *Consider a flow network in which each vertex v has a (positive or negative) demand $d(v)$. Reduce the following problem to a max flow problem: find a circulation f such that $f(u,v) \leq c(u,v)$, $f(u,v) = -f(v,u)$ and $\sum_{u \in V} f(u,v) = d(v)$ for all v.* ☆

2 The Ford-Fulkerson method (1956)

Definition 2.1. *Given a flow network G and a flow f in G, we can define the* **residual** *flow network G_f by setting $G_f = (V, E_f)$ and $c_f(u,v) = c(u,v) - f(u,v)$. The set of edges $E_f = \{(u,v) | c_f(u,v) > 0\}$.*

The residual network G_f thus represents the remaining capacity given flow f. Notice, if $(u,v) \notin E$, that is, $c(u,v) = 0$ and there was a flow from v to u, i.e., $f(u,v) < 0$ due to the skew symmetry, then $c_f(u,v) = 0 - f(u,v) > 0$, meaning $(u,v) \in E_f$.

Lemma 2.1. *Let f be a flow in G and f' in G_f, then $f + f'$ defined as $(f + f')(u,v) = f(u,v) + f'(u,v)$, for all $u, v \in V$, is a flow and $|f + f'| = |f| + |f'|$.*

Proof. The proof is completely analogue to Lemma 1.1, except that we also need to verify the capacity constraints: $(f + f')(u,v) = f(u,v) + f'(u,v) \leq f(u,v) + c_f(u,v) = c(u,v)$. □

We are now in a position to define the notion of an augmenting path.

Definition 2.2. *An* **augmenting** *path p with respect to a flow network G and a flow f is a simple path from s to t in G_f, meaning each vertex on the path is visited only once.*

Notice, all edges E_f in G_f have a positive capacity, meaning an augmenting path p has a positive **residual** capacity $c_f(p)$ defined as $\min\{c_f(u,v) | (u,v) \in p\}$. An augmenting path p therefore immediately results in a flow f_p in the residual network G_f, by defining $f_p(u,v) = c_f(p)$ for the edges (u,v) on p and $f_p(u,v) = -c_f(p)$ for (v,u) on p. Combining this with the previous two lemmas we have

Corollary 2.1. *Let f be a flow in G and p an augmenting path with respect to G and f, then $f' = f + f_p$ defines a new flow in G with $|f'| > |f|$.*

This gives cause to the Ford-Fulkerson method:

Algorithm 2.1 (Ford-Fulkerson)**.** *;*

1. *initialize* $f(u,v) = 0$ *for all* u, v*;*
2. **while** *there exists an augmenting path* p
3. **do** *augment flow* f *with* f_p*;*
4. **return** *f;*

We need to introduce two more concepts before we can prove the max-flow min-cut theorem and thus the correctness of the algorithm above.

Definition 2.3. *A* **cut** (S,T) *in a flow network* $G = (V,E)$ *is defined as a partitioning of* V *into* S *and* T *such that* $s \in S$*,* $t \in T$ *(and* $S \cap T = \emptyset$ *as it is a partitioning).*

Given a flow f in G, we define the **net flow** $f(S,T)$ across the cut (S,T) as

$$f(S,T) = \sum_{u \in S, v \in T} f(u,v).$$

The **capacity** $c(S,T)$ of the cut (S,T) on the other hand, is defined as

$$c(S,T) = \sum_{u \in S, v \in T} c(u,v).$$

Notice, some of the terms $f(u,v)$ may be negative, which is never the case for $c(u,v)$.

Lemma 2.2. *Let* f *be a flow in* G *and let* (S,T) *be any cut, then* $f(S,T) = |f| \leq c(S,T)$.

Proof. Let X and Y be two subsets of V, then define

$$f(X,Y) = \sum_{u \in X, v \in Y} f(u,v).$$

One readily checks that $f(X,X) = 0$ due to the skew symmetry, $f(X, Y_1 \cup Y_2) = f(X,Y_1) + f(X,Y_2)$ and $f(X_1 \cup X_2, Y) = f(X_1,Y) + f(X_2,Y)$ if $Y_1 \cap Y_2 = X_1 \cap X_2 = \emptyset$. As a result, since $V = S \cup T$ and $S \cap T = \emptyset$, we have

$$f(S,T) = f(S,V) - f(S,S) = f(S,V) = f(s,V) + f(S-s,V) = f(s,V) = |f|,$$

where $f(S-s,V) = 0$ due to the flow conservation property and the fact that $t \notin S - s$. The inequality can be established as follows $|f| = f(S,T) = \sum_{u \in S, v \in T} f(u,v) \leq \sum_{u \in S, v \in T} c(u,v) = c(S,T)$. □

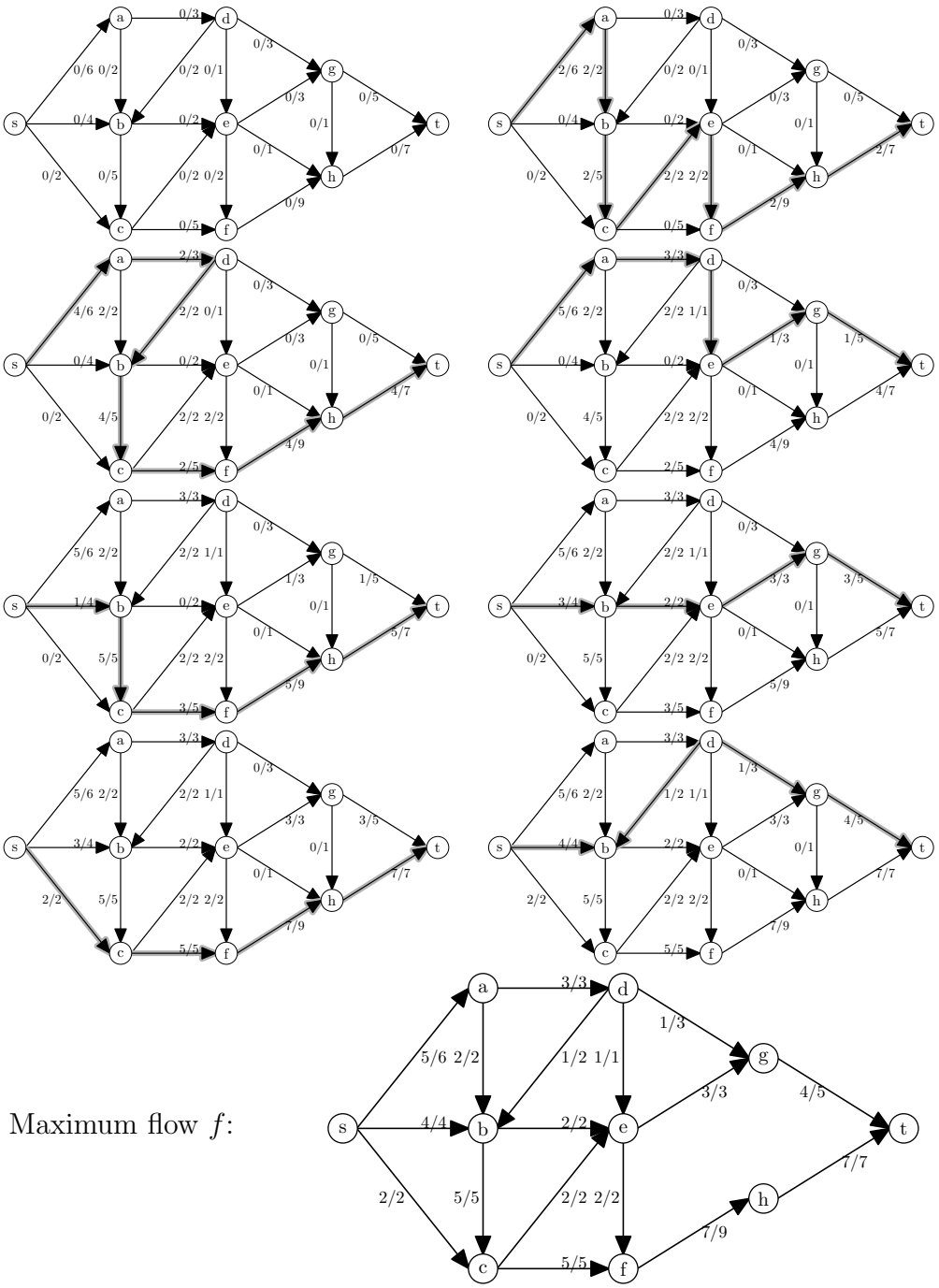


Fig. 7: Example of Ford Fulkerson method with $|f| = 11$ (min-cut $S = \{s, a\}$).

With this lemma we can now proof the max-flow min-cut theorem that proves the correctness of the Ford-Fulkerson method:

Theorem 2.1 (Max-flow Min-cut theorem)**.** *Let f be a flow in a flow network G with source s and sink t, then the following three statements are equivalent:*

1. *f is a maximum flow in G,*
2. *The residual network G_f contains no augmenting paths (that is, there is no simple path from s to t in G_f),*
3. *$|f| = c(S,T)$ for some cut (S,T) of G.*

Proof. $(1) \rightarrow (2)$: If f is maximum then there cannot be an augmenting path with respect to G and f due to Corollary 2.1.
$(2) \rightarrow (3)$: Assume there is no simple path from s to t in G_f. Define the cut (S,T) as

$$S = \{v \in V \mid \text{there is a path from } s \text{ to } v\},$$

and $T = V - S$. Notice, $s \in S$ and $t \in T$. For each (u,v) with $u \in S$ and $v \in T$, we have $f(u,v) = c(u,v)$ as $(u,v) \notin E_f$ (therefore $0 = c_f(u,v) = c(u,v) - f(u,v)$). Lemma 2.2 implies that $|f| = f(S,T)$ and $f(S,T) = c(S,T)$ as we just argued.
$(3) \rightarrow (1)$: Lemma 2.2 indicates $|f| \leq c(S,T)$ for all cuts (S,T) and flows f in G. Thus, f is a maximum flow. □

Exercises 2.1. *Flow network problems:*

1. *The edge connectivity number $\lambda(G)$ of a graph is equal to the number of edges one needs to remove to disconnect a graph. Given an undirected graph $G = (V,E)$, show that $\lambda(G)$ can be determined by running at most $|V|$ maximum-flow algorithms on a flow network with at most $O(|V|)$ vertices and $O(|E|)$ edges.*
2. *The vertex connectivity number $\kappa(G)$ of a graph is equal to the number of vertices one needs to remove to disconnect a graph. Given an undirected graph $G = (V,E)$, show that $\kappa(G)$ can be determined by running at most $|V|(\kappa(G)+1)$ maximum-flow algorithms on a flow network with at most $O(|V|)$ vertices and $O(|E|)$ edges.*
3. *True or False?* ☆

(a) *Let (S,T) be a minimum cut, then (S,T) is still a minimum cut if we increase all the edge capacities by one.*

(b) *There is a unique minimum cut if and only if there is a unique maximum flow.*

(c) *Suppose we always pick a simple path p in G_f such that $c_f(p)$ is maximized, then $c_f(p)$ cannot increase while using the method of Ford and Fulkerson.*

4. *Let (S_1,T_1) and (S_2,T_2) be minimum cuts of G. Argue that $(S_1 \cap S_2, V\backslash(S_1 \cap S_2))$ and $(S_1 \cup S_2, V\backslash(S_1 \cup S_2))$ are also minimum cuts.* ☆[2]

5. *Given the minimum cut (S,T) where S is the set of vertices reachable from s in G_f. How can we check in $O(|V| + |E|)$ time whether this cut is unique? [Hint: use the previous exercise.]* ☆[2]

6. *Indicate how we can find an augmenting path p with $c_f(p)$ maximized in a residual network $G_f = (V, E_f)$ in $O((|V| + |E_f|)\log|E_f|)$ time. Start by sorting the edges in E_f by weight.* ☆

7. *Find an efficient method to determine a minimum cut (S,T) such that the number of edges between S and T is minimized given that all the edge capacities are integers. [Hint: augment all edge capacities by a constant.]* ☆

3 Performance of the Ford-Fulkerson algorithm

When running the Ford-Fulkerson algorithm we start with a flow value $|f| = 0$. After each iteration $|f|$ increases. The question is however whether $|f|$ converges to the maximum flow value for the flow network $G = (V, E)$. The answer is that if we allow irrational capacities $c(u, v)$, the algorithm might not converge to the maximum flow value.

An example is shown in Figure 8, that depicts a network $G = (V, E)$ consisting of 6 nodes and 9 edges with edge capacities of 1, $r = (\sqrt{5}-1)/2 < 1$ and M, where $M \geq 2$ is an integer. Note, r is an irrational number that satisfies the equation $1 - r = r^2$. Define the paths p_0, p_1, p_2 and p_3 as

$$p_0 : s \to x_2 \to x_3 \to t,$$
$$p_1 : s \to x_4 \to x_3 \to x_2 \to x_1 \to t,$$
$$p_2 : s \to x_2 \to x_3 \to x_4 \to t,$$
$$p_3 : s \to x_1 \to x_2 \to x_3 \to t.$$

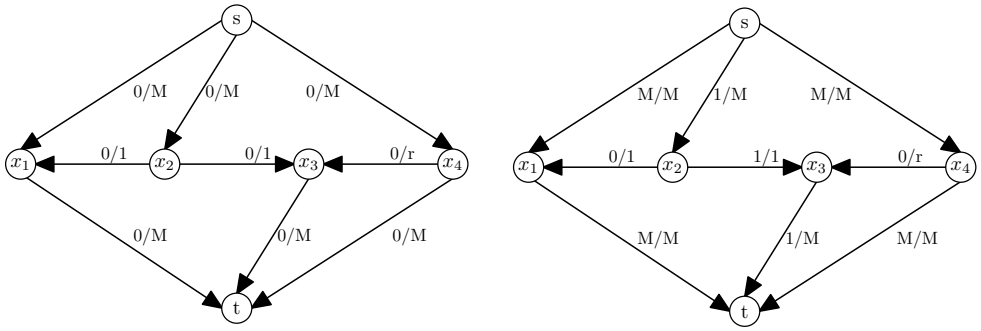


Fig. 8: Ford Fulkerson method: the termination problem.

Now consider what happens to the residual capacities $c_f(x_2, x_1), c_f(x_3, x_2)$ and $c_f(x_4, x_3)$ if we use the paths p_0 followed by p_1, p_2, p_1, p_3. We denote f_n as the flow obtained after using n paths. If we start with path p_0, with $c_{f_0}(p_0) = 1$, we get the following residual capacities: $c_{f_1}(x_2, x_1) = 1 = r^0$, $c_{f_1}(x_3, x_2) = 1$ and $c_{f_1}(x_4, x_3) = r$. Hence, $c_{f_1}(p_1) = r$ and p_1 decreases the capacity on (x_2, x_1) such that $c_{f_2}(x_2, x_1) = 1 - r = r^2$. The residual capacity on (x_4, x_3) and (x_3, x_2) is also reduced by p_1, however, path p_2 will restore these capacities as $c_{f_2}(p_2) = r$ (due to $c_{f_2}(x_2, x_3) = r$), meaning $c_{f_3}(x_2, x_1) = r^2$, $c_{f_3}(x_3, x_2) = 1$ and $c_{f_3}(x_4, x_3) = r$. Similarly, if we now use the paths p_1 followed by p_3, we find that $c_{f_3}(p_1) = c_{f_4}(p_3) = r^2$. Path p_1 reduces the residual capacity on the edge (x_4, x_3) to $r - r^2 = r^3$, while path p_3 restores the residual capacities of (x_2, x_1) and (x_3, x_2) back to 1 and r^2. In other words $c_{f_5}(x_2, x_1) = r^2$, $c_{f_5}(x_3, x_2) = 1$ and $c_{f_5}(x_4, x_3) = r^3$.

If we now repeatedly use the paths p_1, p_2, p_1, p_3, one finds that $c_{f_7}(x_2, x_1) = r^4$, $c_{f_7}(x_3, x_2) = 1$ and $c_{f_7}(x_4, x_3) = r^3$ and $c_{f_6}(p_1) = c_{f_7}(p_2) = r^3$; $c_{f_9}(x_2, x_1) = r^4$, $c_{f_9}(x_3, x_2) = 1$ and $c_{f_9}(x_4, x_3) = r^5$ and $c_{f_8}(p_1) = c_{f_9}(p_3) = r^4$, etc. In other words the Ford and Fulkerson algorithm does not terminate and the net total flow $|f_i|$ converges to $2\sum_{i=0}^{\infty} r^i - 1 = 2 + \sqrt{5} < 5$, while the maximum flow is $2M + 1 \geq 5$. Hence, the Ford-Fulkerson algorithm does not guarantee convergence in general.

However, things are less problematic when all the capacities are rational (r in the previous example is an irrational number). In this case, we can multiply all capacities by the smallest common multiple of all denominators that appear in a capacity $c(u, v)$, thereby reducing the capacities to integer numbers. When all the capacities are integers, then all augmenting paths clearly have an integer maximum flow. As a result, the Ford-Fulkerson algorithm

guarantees convergence in at most $|f^*|$ steps, where $|f^*|$ is the maximum flow of the network of interest (and which is certainly upper bounded by the sum of all capacities $c(u, v)$). Thus, if a network has a maximum flow of $1,000,000$, the algorithm might potentially take as many as $1,000,000$ steps.

To avoid such a slow convergence one may rely on the Edmonds-Karp algorithm that will select the augmenting path p in the residual network by applying a breadth-first search to determine the shortest path from s to t, in the next section we proof that the time complexity of such an approach is $O(|V||E|^2)$, with $|V|$ the number of vertices and $|E|$ the number of edges in the flow network, independent of the flow capacity.

Exercises 3.1. *Flow network problems:*

1. *Give an example of a small flow network $G = (V, E)$ with integer capacities such that $1,000,000$ or more steps might be required to obtain a maximum flow f.*

2. *Develop a flow network to check whether there exists a $n \times m$ matrix with its entries either equal to 0 or 1 such that the i-th row sum equals r_i, for $i = 1, \ldots, n$, the i-th column sum c_i, for $i = 1, \ldots m$ (with $\sum_{i=1}^{n} r_i = \sum_{i=1}^{m} c_i$).*

3. *A binary matrix M is rearrangeable if and only if an arbitrary number of switches of its rows and columns can set all the diagonal entries equal to 1.* ☆

   (a) *Give an efficient algorithm to determine whether a matrix A is rearrangeable and discuss its time complexity.*

   (b) *Give an example of a matrix A with at least one 1 on each row and column that is not rearrangeable.*

4. *A software house has to handle 3 projects, P1, P2, P3, over the next 4 months. P1 can only start in month 2, and must be completed after 3 months. P2 and P3 can start at month 1, and must be completed, respectively, within 4 and 2 months. The projects require, respectively, 8, 10, and 12 man-months. For each month, 8 engineers are available. Due to the internal structure of the company, at most 6 engineers can work, at the same time, on the same project. Describe how to reduce this problem to the problem of finding a maximum flow on an appropriate graph. [Hint: Find a flow with $|f| = 30$.]* ☆

5. *Consider a directed graph G and some functions $w : V \to \mathbb{R}^+$, $b : V \to \mathbb{R}^+$ and $p : E \to \mathbb{R}^+$. Let $c : V \to \{black, white\}$ be a 2-coloring of the*

vertices of G. Indicate how we can find a coloring that minimizes ☆

$$\sum_{v \in V, c(v)=white} w(v) + \sum_{v \in V, c(v)=black} b(v) + \sum_{(v,v') \in E, c(v) \neq c(v')} p((v,v')).$$

6. *True or False? A flow network has a unique minimum cut (S,T) if there are no two edges with the same capacity.* ☆

7. *An edge e is upwards critical if increasing $c(e)$ increases the max flow and downwards critical if decreasing $c(e)$ decreases the max flow. True or False?* ☆

   (a) *Every flow network with $|f| > 0$ contains at least one upwards critical edge.*

   (b) *Every flow network with $|f| > 0$ contains at least one downwards critical edge.*

4 The Edmonds-Karp algorithm (1969)

The analysis of the Edmonds-Karp algorithm depends on the length of the shortest paths $\delta_f(s,u)$ between the vertex s and $u \in V$ in the residual network G_f, where f is a flow in G. The next lemma proves that the shortest-path distances monotonically increase with each flow augmentation.

Lemma 4.1. *Let f and f' be two successive flows obtained by the Edmonds-Karp algorithm and G_f and $G_{f'}$ their residual networks, respectively. Then, for all $v \in V$: $\delta_f(s,v) \leq \delta_{f'}(s,v)$.*

Proof. Suppose the lemma is false and choose a $v \in V$ with

$$\delta_f(s,v) > \delta_{f'}(s,v),$$

such that $\delta_{f'}(s,v)$ is minimal (that is, if $\delta_{f'}(s,u) < \delta_{f'}(s,v)$, then $\delta_f(s,u) \leq \delta_{f'}(s,u)$). Let p be the shortest-path from s to v in $G_{f'}$ where the last edge is assumed to be edge (u,v). Then,

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1,$$

and by the minimality of v

$$\delta_f(s,v) > \delta_{f'}(s,v) = \delta_{f'}(s,u) + 1 \geq \delta_f(s,u) + 1. \tag{1}$$

Therefore, (u,v) cannot be an edge in G_f, otherwise $\delta_f(s,v) \leq \delta_f(s,u) + 1$. If $(u,v) \in G_{f'}$, while $(u,v) \notin G_f$, then the shortest path from s to t in G_f

must contain the edge (v, u), otherwise (u, v) cannot be part of $G_{f'}$. As a consequence, as the path via v is also a shortest-path to u, we have

$$\delta_f(s, v) = \delta_f(s, u) - 1,$$

which contradicts (1). □

This allows us to prove the following theorem without much difficulty:

Theorem 4.1. *The Edmonds-Karp algorithm to determine the maximum flow of a flow network G requires at most $O(|V||E|)$ augmentations.*

Proof. An (u, v) edge on an augmenting path p in a residual graph G_f is termed critical if it has the lowest capacity $c_f(u, v)$ along the path p, therefore $c_f(u, v) = c_f(p)$ and

$$\delta_f(s, v) = \delta_f(s, u) + 1.$$

Whenever an edge (u, v) becomes critical, the edge (u, v) disappears from the residual network. The edge (u, v) can only reappear in a residual network later on if the edge (v, u) becomes part of an augmenting path p' in some residual network $G_{f'}$. Thus at some later point in time we have

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1,$$

where the inequality stems from Lemma 4.1. Combining both we see

$$\delta_{f'}(s, u) \geq \delta_f(s, u) + 2.$$

As a result, the distance from s to u increases between two successive augmentations where uv is critical by at least two. Given that this distance is always at most $|V|$, the edge uv can be at most $O(|V|)$ times critical.

Also, the number of edges that can become critical is $O(|E|)$ (because whenever (u, v) and (v, u) are not part of E, neither (u, v) or (v, u) can become part of a residual network). Further, each augmentation requires some edge to be critical, which proves the statement. □

Using a breath-first search one can find the shortest-path in a residual network in time $O(|E|)$, meaning the Edmonds-Karp algorithm has an overall complexity of $O(|V||E|^2)$.

Exercises 4.1. *Flow problems:*

1. *Show that a maximum flow in a network can always be found via $|E|$ augmenting paths (*Hint*: determine the paths after finding the maximum flow).*

2. *Let f be a flow in $G = (V, E)$, f^* a maximum flow and $G_f(\Delta)$ the residual network of G for the flow f with all edges (u, v) with $c_f(u, v) < \Delta$ removed. Show that $|f^*| \leq |f| + \Delta|E|$ if there is no path from s to t in $G_f(\Delta)$. [Hint: Consider a particular cut (S', T') and note that $|f^*| \leq c(S, T)$ for any cut (S, T).]* ☆

3. *Using the notation of the previous exercise, assume $c(e) \in \{1, 2, \ldots, C\}$ for $e \in E$. Consider the Capacity Scaling Algorithm:*

   $f = 0$; $\Delta = 2^{\lfloor \log_2 C \rfloor}$
   while $\Delta \geq 1$ ***do***
   while *There exists a path p from s to t in $G_f(\Delta)$* ***do***
   augment f with f_p
   end while
   $\Delta = \Delta/2$
   end while

   (a) *Does this algorithm end after a finite number of iterations and does it yield a maximum flow?*

   (b) *Give an upper bound on the number of times that the outer and inner while loop is executed (use the previous exercise).*

   (c) *What is the overall time complexity?*

5 Preflow-push algorithms

Preflow-push algorithms are different in nature than the Ford and Fulkerson method, as these algorithms do not construct a sequence of flows whose value $|f|$ increases at each iteration. Instead a sequence of **preflows** f is constructed. Preflows are like flows in the sense that they also respect the skew symmetry and capacity constraints, but they are allowed to violate the flow conservation property. However, if there is no flow conservation in a vertex, then the amount of flow entering the vertex must dominate the amount of flow leaving it. Formally, a preflow is a function $f : V \times V \to \mathbb{R}$ for which

1. Capacity constraint: $f(u, v) \leq c(u, v)$ for all $u, v \in V$.
2. Skew symmetry: $f(u, v) = -f(v, u)$ for all $u, v \in V$.
3. Excess property: $e(u) = \sum_{v \in V} f(v, u) \geq 0$ for all $u \in V - \{s\}$.

If $e(u) > 0$, we state that the vertex $u \neq t$ is **overflowing**. The idea of preflows was introduced by Karzanov (1973), while the preflow-push algorithm discussed here is due to Goldberg (1987).

Intuitively the preflow-push method works as follows. We start by giving vertex s a height $h(s)$ of $|V|$ and all other vertices v a height $h(v)$ of 0. Next, we push the maximum possible flow from s to its neighbors, causing these vertices to overflow. The algorithm repeatedly selects an overflowing vertex. This vertex u will either have a neighbor v (in the residual network) with a height $h(v) = h(u) - 1$ such that some of the excess flow in u can flow *down* to v (possibly overflowing v). Notice, if there is an edge in the residual network, then there still remains some capacity from u to v. If an excess vertex does not have such a neighbor, we lift the vertex (by increasing its height $h(u)$) creating such a neighbor.

Thus, the idea is that the initial flow in the neighbors of s will gradually flow downward through the network, while all the vertices (except the sink t and source s) gain height. At some point this causes a maximum amount of flow $|f|$ to reach the sink t. However, the initial flow pushed by s may be considerably larger. What happens next is that the vertices $u \neq s, t$ continue to gain height, such that the redundant flow returns to the source s. When all excess capacity has been removed, the preflow becomes a flow with the required maximum capacity $|f|$.

A preflow-push algorithm will allow two operations (on overflowing vertices): a **push** and a **lift**. The applicability of these operations is regulated by the height of a vertex. A height function $h : V \to \mathbb{R}$ must fulfill the following requirements:

1. $h(s) = |V|$ and $h(t) = 0$,
2. If $(u, v) \in E_f$: $h(v) \geq h(u) - 1$.

Thus, the height may *decrease* by at most one when following an edge in the residual network $G_f = (V, E_f)$, where E_f with f a preflow is defined in exactly the same manner as for a flow f, i.e., $(u, v) \in E_f$ if $c_f(u, v) = c(u, v) - f(u, v) > 0$. This height property/constraint is very important as it will guarantee the optimality of the end flow in a straightforward manner.

Let us now consider the two operations: the push and lift operation as depicted in Figure 9. The push operation applies to an overflowing vertex u if there is at least one neighbor v in the residual network (i.e., one outgoing link with some remaining capacity) with a height one below u, that is, $h(v) = h(u) - 1$. In this case we increase the flow from u to v, if this causes $f(u, v)$ to become equal to $c(u, v)$ we state that the push operation **saturates** the link (u, v).

PUSH(u, v):

Allowed if:
$e(u) > 0$, $(u, v) \in E_f$ (i.e., $c_f(u, v) > 0$) and $h(v) = h(u) - 1$

Push $d_f(u, v)$ **from** u **to** v**:**
$d_f(u, v) = \min(e(u), c_f(u, v))$
$f(u, v) = f(u, v) + d_f(u, v)$
$f(v, u) = -f(u, v)$
$e(u) = e(u) - d_f(u, v)$
$e(v) = e(v) + d_f(u, v)$

LIFT(u):

Allowed if:
$e(u) > 0$, for all $v \in V$: $(u, v) \in E_f$ implies $h(v) \geq h(u)$

Lift u**:**
$h(u) = 1 + \min(h(v) : (u, v) \in E_f)$

Fig. 9: The PUSH and LIFT operation

If such a neighbor v is not available, we lift u such that its height is one above the lowest neighbor (in G_f). It is important to notice that a vertex u, with $e(u) > 0$, always has at least one neighbor in G_f. Indeed, $e(u) = \sum_{v \in V} f(v, u) > 0$, thus there is at least a $v \in V$ with $f(v, u) > 0$. Hence,

$$c_f(u, v) = c(u, v) - f(u, v) = c(u, v) + f(v, u) > 0,$$

and $(u, v) \in E_f$. Thus, *whenever a vertex u is overflowing, we can either perform a push or a lift operation if h is a height function.*

The preflow algorithm is initialized as follows:

1. $u \in V - \{s\} : h(u) = 0$, $h(s) = |V|$,
2. $(u, v) \in E, u, v \neq s : f(u, v) = 0$,
3. $(s, u) \in E : f(s, u) = c(s, u), f(u, s) = -c(s, u)$ and $e(u) = c(s, u)$.

Note, h is a height function, because $f(s, u) = c(s, u)$ implies $(s, u) \notin E_f$, while all other vertices have a height of 0; hence, following an edge (u, v) in E_f we cannot decrease in height (and thus certainly not decrease by more than one).

The lift operation may increase the height $h(u)$ of a vertex u. Thus, for h to remain a height function we must have that $h(v) \geq h(u) - 1$ for $(u, v) \in E_f$. As $h(u)$ is increased to one plus the height of the lowest neighbor in G_f, this remains valid. A push operation does not alter the height of

any vertex, but it may delete or add a new edge from or to G_f. If the push operation saturates (u, v), the edge (u, v) is removed, which causes no problem. The push operation may also add the edge (v, u) to E_f (as $c_f(v, u) = c(v, u) + f(u, v)$). Following this edge from v to u increases the height by one, clearly fulfilling the height constraint which forbids decreasing the height by more than one along such an edge. Therefore, we have proven that: *Whenever a vertex u is overflowing, i.e., $e(u) > 0$, we can either perform a push or a lift operation.*

Assuming that at some point there are no overflowing vertices left, we can easily prove that f is a maximum flow as follows.

Theorem 5.1. *If the preflow-push algorithm terminates, because $e(u) = 0$ for all $u \in V - \{s, t\}$, it produces a maximum flow f for the flow network $G = (V, E)$.*

Proof. We first prove that at any stage during the execution of the algorithm, there is no simple path from s to t in G_f. Assume there is a simple path from s to t in G_f. Then, this path is composed of at most $|V| - 1$ edges (as it is simple). It must start in s with $h(s) = |V|$ and follow a sequence of $|V| - 1$ edges in E_f. While following each of these edges, the height can decrease by at most one (due to the height constraint), meaning while reaching t, the height must still be at least one. However, $h(t) = 0$.

When the algorithm terminates, there is no excess flow and thus f is a flow. Moreover, there is no path from s to t in G_f, meaning there is no augmenting path p in the Ford and Fulkerson sense. As a result, the min-cut max-flow theorem proves the statement. □

6 Performance of the Preflow-push algorithm

In this section we will place an upper bound on the number of lift, saturated and unsaturated push operations. Recall, a push on (u, v) is saturating if $d_f(u, v) = c_f(u, v)$, causing the removal of the edge (u, v) from E_f.

Lemma 6.1. *When performing the preflow-push algorithm on $G = (V, E)$ a flow network, there is a simple path from any overflowing vertex u to s in E_f and the height $h(u)$ of any vertex is at most $2|V| - 1$.*

Proof. Assume $u \in V$ is overflowing, that is, $e(u) > 0$. Let U be the set of vertices reachable via a simple path from u in E_f. Let $\bar{U} = V - U$ and assume $s \notin U$. If $v \in U$ and $w \in \bar{U}$ then $(v, w) \notin E_f$, meaning

$$f(v, w) = c(v, w) \geq 0,$$

and by skew symmetry we find $f(w,v) \leq 0$. Hence, $f(\bar{U},U) \leq 0$. Now,

$$e(U) = f(V,U) = f(\bar{U},U) + f(U,U) = f(\bar{U},U) \leq 0,$$

where $f(U,U) = 0$ due to the skew symmetry. Notice, $e(U)$ is well-defined as $s \notin U$. But, $e(v) \geq 0$ for all $v \in V$ meaning $e(u)$ must be zero, contradicting $e(u) > 0$.

As this simple path has a length of at most $|V| - 1$, $h(s) = |V|$ and the height can only decrease by one along an edge in E_f, we find $h(u) \leq 2|V| - 1$ for u overflowing. Moreover, any vertex is still overflowing immediately after a lift. □

Next, we upper bound the number of lifts and saturated pushes:

Lemma 6.2. *When performing the preflow-push algorithm on $G = (V,E)$ a flow network, at most $(2|V|-1)(|V|-2)$ lifts and $2|V||E|$ saturated pushes are performed.*

Proof. Only vertices $u \in V - \{s,t\}$ can be lifted. Each lift increases $h(u)$ by at least one with $h(u) = 0$ initially. Hence, at most $(2|V|-1)(|V|-2)$ lifts can occur.

A push operation that saturates an edge (u,v) removes this edge from E_f. During this operation $h(v) = h(u) - 1$. To recreate this edge there has to be a push from v to u, which requires that $h(v) = h(u) + 1$, thus $h(v)$ has to increase by two. Meaning, $h(u) + h(v)$ increases by at least two between any two saturated pushes on either (u,v) or (v,u). Now, during the very first push between u and v, we must have $h(u) + h(v) \geq 1$. Due to Lemma 6.1, we know that during the last such push, we have

$$h(u) + h(v) \leq (2|V| - 1) + (2|V| - 2) = 2(2|V| - 1) - 1.$$

This implies there can be at most $(h(u) + h(v) + 1)/2 \leq 2|V| - 1$ saturated pushes along (u,v) or (v,u), as this sum increases by 2 in between two such pushes. In total we therefore have at most $(2|V|-1)|E| < 2|V||E|$ saturated pushes. □

The next lemma focuses on a bound for the number of nonsaturating push operations:

Lemma 6.3. *When performing the preflow-push algorithm on $G = (V,E)$ a flow network, at most $4|V|^2(|V| + |E|)$ nonsaturating pushes are performed.*

Author(s)	*Year*	*Complexity*
Ford, Fulkerson	1956	——
Edmonds, Karp	1969	$O(\|E\|^2\|V\|)$
Dinic	1970	$O(\|E\|\|V\|^2)$
Karzanov	1973	$O(\|V\|^3)$
Cherkassky	1976	$O(\sqrt{\|E\|}\|V\|^2)$
Malhotra, et al.	1978	$O(\|V\|^3)$
Galil	1978	$O(\|V\|^{5/3}\|E\|^{2/3})$
Galil and Naamad	1979	$O(\|E\|\|V\|\log_2\|V\|)$
Sleator and Tarjan	1980	$O(\|E\|\|V\|\log\|V\|)$
Goldberg and Tarjan	1985	$O(\|E\|\|V\|\log(\|V\|^2/\|E\|))$

Fig. 10: Performance of various maximum flow algorithms [H.S. Wilf]

Proof. Define $\Psi = \sum_{v \in X} h(v)$, where X is the set of overflowing vertices. Initially, $\Psi = 0$. Due to lemma 6.1, each lift operation can increase Ψ by at most $2|V|$ (minus one) as X remains the same. A saturated push along (u, v) may turn v into an overflowing vertex. Thus, v might be added to X, potentially increasing Ψ by $2|V|$ (minus one). Thus, due to the previous bounds on the number of lifts and saturating pushes, Ψ increases by at most

$$(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V| + |E|).$$

An nonsaturating push along (u, v) implies that $e(u)$ becomes 0, i.e., u is no longer overflowing. The vertex v may become an overflowing vertex due to this push, but $h(v) = h(u) - 1$; hence, an unsaturating push decreases Ψ by at least 1 and $\Psi \geq 0$ at all times. This allows us to conclude that at most $4|V|^2(|V| + |E|)$ nonsaturating pushes can occur. □

Combining the last three lemmas, we have

Theorem 6.1. *The preflow-push algorithm by Goldberg requires at most* $O(|V|^2|E|)$ *basic operations (pushes and lifts).*

As a push can be implemented in $O(1)$ and a lift in $O(|V|)$ the overall complexity equals $O(|V|^2|E|)$ as well. Various faster algorithms have been developed since 1973, see Table 10. Some of these improve the preflow-push result by choosing the lift and push operations in a systematic way, mainly causing a reduction in the number of nonsaturating pushes as these contribute most to the time complexity.

Exercises 6.1. *Flow problems:*

1. *Show that the preflow-push algorithm can be changed by initializing* $h(s) = |V| - 2$ *instead of* $h(s) = |V|$*, without affecting the correctness or asymptotic performance of the algorithm.*

2. *Assume that all the edge capacities* $c(u, v) \in \{1, 2, \ldots, k\}$*. Give an expression for the asymptotic time complexity of the preflow-push algorithm in terms of* $|V|, |E|$ *and* k *(Hint: focus on the number of nonsaturating push operations).*

BIPARTITE MATCHING ALGORITHMS

1 The graph matching problem

We start with some basic definitions. Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ be a subset of the edges of G. We call M a **matching** of G if all edges $e \in M$ are vertex disjoint. The vertices incident to M are referred to as matched vertices, whereas the remaining vertices are termed unmatched. A matching M is said to be maximum if for all other matchings M' we have $|M| \geq |M'|$.

This section focuses on efficient algorithms to find a maximum matching of undirected bipartite, i.e., 2-colorable, graphs.

2 A network flow solution

We start by solving the bipartite matching problem through the use of network flow theory. More specifically, we will construct a simple flow network with source s and sink t such that a maximum flow in this network corresponds to a maximum matching and vice versa.

Let $\bar{G} = (\bar{V}, \bar{E})$ be a bipartite graph where $\bar{V}$ is partitioned into a set L and R (being the left and right vertices of $\bar{G}$ such that $(u, v) \in \bar{E}$ implies that $u \in L$ and $v \in R$). Define $G = (V, E)$ as

$$V = \bar{V} \cup \{s, t\}, E = \bar{E} \cup \{(s, u) | u \in L\} \cup \{(v, t) | v \in R\},$$

where each undirected edge (u, v) in $\bar{E}$ contributes one directed edge to E, being (u, v) with $u \in L$ and $v \in R$. The capacity function $c(u, v) = 1$ for all $(u, v) \in E$ (see Figure 11).

The following theorem states how to solve the matching problem via a flow network:

THEOREM 2.1. *Let $\bar{G} = (\bar{V}, \bar{E})$ an undirected bipartite graph and $G = (V, E)$ its corresponding flow network. A maximum matching in $\bar{G}$ corresponds to a maximum flow f in G with $|f| = |M|$ and vice versa.*

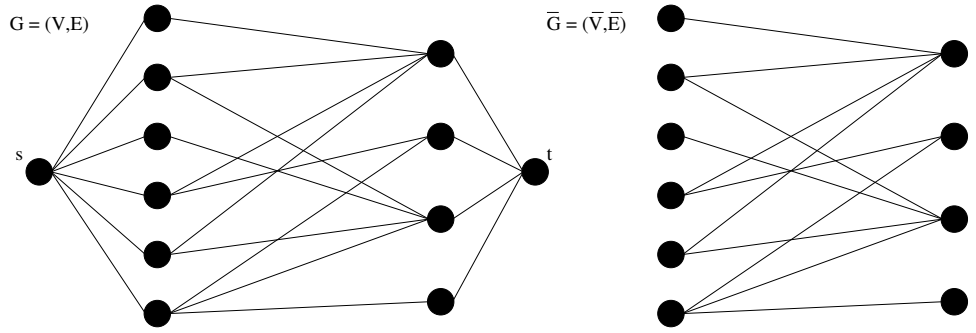


Fig. 11: Bipartite matching and flow networks

Proof. Assume $M \subseteq \bar{E}$ is a matching of $\bar{G}$ and let L_M and R_M be the set of matched vertices in L and R respectively. Define the flow f as

$$f(u,v) = \begin{cases} 1 & u = s, v \in L_M, \\ 1 & (u,v) \in M, \\ 1 & u \in R_M, v = t, \\ 0 & otherwise. \end{cases}$$

Clearly, this is a flow f as the input and output rate of the vertices in L_M and R_M equals one, while all other vertices in $L \cup R$ receive no flow. Moreover, $|f| = |M|$.

Assume on the other hand that f is an integer valued flow, meaning $f(u,v) \in \mathbb{N}$ for all $(u,v) \in E$. All the vertices $u \in L$ have only one incoming edge, being (s,u) with capacity 1, and all vertices in $v \in R$ have only one outgoing edge, being (v,t) with capacity 1. Therefore, all vertices in $\bar{V}$ either have a net input flow of 1 or 0 and thus all edges have $f(u,v) = 0$ or 1. Define $M \subseteq E$ as $M = \{(u,v) | f(u,v) = 1, u \in L, v \in R\}$. Let (u,v) and (u',v') be in M, then $u = u'$ implies that u has an outgoing flow of at least two, which contradicts our earlier statement. Similarly, $v = v'$ would imply that v has an incoming flow of 2, thus M is a matching. We also immediately see that $|M| = |f|$.

Due to the Ford and Fulkerson algorithm we know that an integer valued maximum flow of $G = (V,E)$ exists and hence, determines a maximum matching M. □

Recall, the number of augmenting paths for an integer valued flow network that can be generated via the Ford and Fulkerson method was bounded by $|f^*|$ where f^* is a maximum flow. Clearly, the size of the maximum matching $|M^*| \leq |L| \leq |V|$, meaning $O(|V|)$ iterations suffice. To find

an augmenting path, we can use the Edmonds-Karp algorithm which determines the shortest augmenting path via a breath-first search in $O(|E|)$ time. The overall performance of matching a bipartite graph via a flow network therefore equals $O(|V||E|)$.

Exercises 2.1. *Matching problems:*

1. *Let $G = (V, E)$ be a bipartite graph with $|L| = |R|$. Prove Hall's theorem which states that a perfect matching M, i.e., one with $|M| = |L|$, exists if and only if for all $A \subseteq L$, $|A| \leq |\Gamma(A)|$ where $\Gamma(A)$ are all the neighbors of A in R. Make use of the min-cut max-flow theorem.*

2. *Use the previous exercise to show that if $G = (V, E)$ is a bipartite graph with $|L| = |R|$ and each node has exactly d neighbors, then there exists a matching M with $|M| = |L|$.*

3. *Consider a bipartite graph G with $|L| = |R| = n$ where each vertex has at least $n/2$ neighbors. Prove that a perfect matching M exists for G.* ☆

4. *Let M be a matching such that there exists no mathing M' with $M \subset M'$. Give an $O(|V| + |E|)$ algorithm to find such a matching M. Let M^* be a matching with $|M^*|$ maximized. Show that $|M^*| \leq 2|M|$ or give a counter example.* ☆

5. *A vertex cover of a bipartite graph $G = (V, E)$ with $V = L \cup R$ is a subset C of V such that for any $(u, v) \in E$ we have $u \in C$ or $v \in C$. Given a vertex cover C and the flow network used to find a maximum matching M in G.* ☆

   (a) *Show that there is a cut (S, T) with $c(S, T) = |C|$.*

   (b) *Conclude that the maximum flow is upper bounded by the minimum vertex cover size.*

3 The Hopcroft-Karp algorithm (1973)

The Hopcroft-Karp algorithm was developed in 1973 and is to date still among the fastest sequential algorithms available for bipartite graph matching. We will demonstrate that its complexity equals $O(\sqrt{|V|}|E|)$.

3.1 The algorithm and its number of iterations

Let $G = (V, E)$ be a bipartite graph where V is partitioned into a set L and R. A simple path P is said to be an **augmenting path with respect to a matching** M, if it starts in an unmatched vertex of L, its edges alternatively belong to $E - M$ and M and it ends in an unmatched vertex of R.

Further, let A and B be two sets of edges of G, then the **symmetric difference** $A \oplus B$ equals $(A - B) \cup (B - A)$, that is, $A \oplus B$ contains all edges in A and B, except for the ones they have in common.
The following property shows that augmenting paths with respect to a matching M, allows us to obtain a larger matching M' by taking the symmetric difference.

Lemma 3.1. *Let M be a matching for G and $P_1, P_2, \ldots, P_k$ a set of k vertex-disjoint augmenting paths with respect to M, then $M' = M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k)$ is a matching for G with $|M'| = |M| + k$.*

Proof. We can prove this by induction on k. Let $k = 1$. P_1 is a simple path, of length $2l + 1$ for some l, that starts and ends in an unmatched vertex. Edge number $2i$ belongs to M, for all $1 \leq i \leq l$, whereas the remaining $l + 1$ edges belong to $E - M$. Thus, $M \oplus P_1$ contains $|M| + 1$ edges and is clearly a matching for G as the first and last vertex of P_1 did not belong to M, while all others did, but their associated edge in M is removed by $M \oplus P_1$. The result for $k > 1$ follows immediately as

$$M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k) = (M \oplus (P_1 \cup P_2 \cup \ldots \cup P_{k-1})) \oplus P_k,$$

because $P_1, \ldots, P_k$ are vertex-disjoint (to see this, simply draw the edge sets and their intersections). Furthermore, as $P_1, \ldots, P_k$ are vertex-disjoint, P_k is also an augmenting path with respect to $(M \oplus (P_1 \cup P_2 \cup \ldots \cup P_{k-1}))$; therefore, we can make use of the $k = 1$ result to complete the proof. □

The next theorem shows that for any matching M, one can always find a set of vertex-disjoint augmenting paths $P_1, \ldots, P_k$ such that $M' = M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k)$ is a maximum matching.

Theorem 3.1. *Let M and M^* be two arbitrary matchings for G with $|M^*| = |M| + k$, for $k > 0$. Then, $M \oplus M^*$ contains at least k vertex-disjoint augmenting paths with respect to M*

Proof. Consider the graph with edges $M \oplus M^*$. As all vertices in M and M^* have a degree equal to 1, all vertices in $M \oplus M^*$ have degree 1 or 2. Hence, $M \oplus M^*$ is a set of vertex-disjoint simple paths and cycles. Moreover, all

edges part of these paths and cycles alternatively belong to M and M^*. The number of edges belonging to M and M^* is therefore equal for the cycles. For the simple paths there can be one additional edge belonging to M or M^*, if the first and the last edge belong to the same matching. Further, the number of edges in M and M^* that are no longer part of $M \oplus M^*$ is identical for both (and equals the number of common edges).

As $|M^*| = |M| + k$, $M \oplus M^*$ must contain at least k paths that start and end with an edge in M^*. Neither, the first nor the last vertex of each of these k paths was matched by an edge in $M \cap M^*$, as M^* is a matching. Thus, these k simple paths are augmenting paths with respect to M □

We proceed with the following two results before presenting the Hopcroft-Karp algorithm.

Theorem 3.2. *Let l be the length of the shortest augmenting path with respect to a matching M. Let $P_1 \cup \ldots \cup P_k$ be a maximum vertex-disjoint set of augmenting paths of length l with respect to M. If P is a shortest augmenting path with respect to $M' = M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k)$, then P must have a length strictly larger than l.*

Proof. First note that the term *maximum* is used to indicate that given $P_1, \ldots, P_k$, one cannot find another augmenting path P_{k+1} of length l with respect to M that is vertex-disjoint with $P_1, \ldots, P_k$.

Consider $A = M \oplus (M' \oplus P)$, where $M' \oplus P$ is a matching of size $|M| + k + 1$ for G due to Lemma 3.1. Further, due to Theorem 3.1, A holds $k + 1$ vertex-disjoint augmenting paths $P'_1, \ldots P'_{k+1}$ with respect to M. As l is the shortest length, these $k + 1$ paths all have length l or more. Hence, $|A| \geq (k + 1)l$ and $P'_1 \cup \ldots \cup P'_{k+1} \subseteq A$.

Now, $A = M \oplus (M' \oplus P) = (M \oplus M') \oplus P = (P_1 \cup \ldots \cup P_k) \oplus P$. Thus, if $|A| > (k + 1)l$, then P must contain strictly more than l edges. Otherwise, suppose $|A| = (k + 1)l$ and P does not share an edge with $(P_1 \cup \ldots \cup P_k)$, meaning $A = P_1 \cup \ldots \cup P_k \cup P$. As $P'_1 \cup \ldots \cup P'_{k+1} \subseteq A$, the set of vertex-disjoint augmenting paths $P'_1, \ldots, P'_{k+1}$ must equal $P_1, \ldots, P_k, P$. As $P_1, \ldots, P_k$ formed a maximum set of augmenting paths of length l, P cannot have length l.

On the other hand, if P and $(P_1 \cup \ldots \cup P_k)$ share at least one edge, while $|A| = |(P_1 \cup \ldots \cup P_k) \oplus P| = (k + 1)l$, we can conclude that P must contain at least $l + 1$ edges. □

Lemma 3.2. *If l is the length of a shortest augmenting path P with respect to a matching M, then the size of the maximum matching is at most $|M| + |V|/l$.*

Proof. Let M^* be a maximum matching with $|M^*| = |M| + k$. Theorem 3.1 tells us that $M \oplus M^*$ contains at least k vertex-disjoint augmenting paths with respect to M. As each of these paths has a length of at least l, there can be at most $|V|/l$ such paths. □

We are now in a position to formulate the Hopcroft-Karp algorithm:

Hopcroft-Karp(G)
$M \leftarrow \emptyset$;
repeat
    Let l be the length of the shortest augmenting path wrt M;
    $P \leftarrow \{P_1, \ldots, P_k\}$ is a maximum set of vertex-disjoint shortest paths with respect to M (all having length l);
    $M \leftarrow M \oplus (P_1 \cup \ldots \cup P_k)$;
until $P = \emptyset$;
return M;

Due to Theorem 3.2 and Lemma 3.2 we find that at most $2\sqrt{|V|}$ iterations are required as after iteration $\sqrt{|V|}$, l must equal $\sqrt{|V|}$ or more and thus the maximum matching is at most $|V|/\sqrt{|V|} = \sqrt{|V|}$ edges larger. Each iteration also adds at least one edge to M, thus after $2\sqrt{|V|}$ iterations P will be empty.

It remains to find an algorithm in $O(|V|+|E|)$ time to determine a maximum vertex-disjoint set of shortest augmenting paths.

3.2 Implementing a single iteration

Given a matching M we need to find a maximum set of vertex-disjoint shortest augmenting paths $P_1, \ldots, P_k$. Notice, maximum means that one cannot find another shortest augmenting path P_{k+1} such that $P_1, \ldots, P_{k+1}$ are vertex disjoint. However, a completely different set might contain more than k paths.

To get a $O(|V|+|E|)$ solution, we start by defining the following directed graph with source s:

$$\begin{aligned} V' &= V \cup \{s\}, \\ E' &= \{(s,u) | u \in L - L_M\} \cup \{(u,v) |\ (u,v) \in E - M, u \in L, v \in R\} \\ &\quad \cup \{(v,u) |\ (u,v) \in M, u \in L, v \in R\}, \end{aligned}$$

where M is the matching, L_M and R_M are the matched vertices in L and R, respectively. As a first step, we perform a BFS search on the graph (V', E')

with source s. However, we stop the search as soon as we encounter an unmatched vertex u^* with $d(u^*) > 1$ (meaning an unmatched vertex that is not a neighbor of s). Notice, any edge followed from R to L is matched and therefore ends in a matched vertex, hence, $u^* \in R - R_M$. More specifically, we end the BFS search when the next vertex we are about to discover has $d(v) = d(u^*) + 1$, that is, we complete the level of u^*.

By construction, the graph $\hat{G}$ consisting only of the vertices discovered by the BFS search (without s) and the edges $(u, v) \in E$ with $d(v) = d(u) + 1$ holds all the shortest augmenting paths with respect to M. Indeed, the BFS shortest path property guarantees that the path from s to u^* is a shortest augmenting path (without s). Further any unmatched vertex that is undiscovered when completing level $d(u^*)$ cannot be part of a shortest augmenting path. Also, the graph G contains no edges (u, v) with $d(v) > d(u) + 1$ as this would violate the shortest path property of the BFS and any edges (u, v) with $d(v) \leq d(u)$ cannot be part of a shortest augmenting path, as the distance to s must increase by one along any shortest path.

The graph $\hat{G}$ is often termed a layered graph and can clearly be constructed during the BFS search in $O(|V| + |E|)$ time. Notice, unmatched vertices either have $d(u) = 1$ (if they are part of L) or $d(u) = d(u^*)$ (if they belong to R).

The second step is to perform a DFS search on $\hat{G}$ where the set of initial vertices is restricted to $u \in L - L_M$. We will use $d(u)$ to identify the distance of u in the BFS search and $\hat{d}(u)$ to indicate the discover time in the DFS algorithm. Each time we discover an unmatched vertex v with $d(v) = d(u^*)$ (where $d(u^*)$ is known from the BFS search), we add the path that led to v to the set of vertex disjoint augmenting paths and mark all the vertices on the path as forbidden. Next, we move on to the next unmatched vertex in $L - L_M$ and continue the DFS search. During this search, edges that lead to a vertex marked as forbidden are not further explored. When the search finishes a maximum set of vertex disjoint augmenting paths $P_1, \ldots, P_k$ is obtained as argued below.

Assume there is another path from a vertex $u \in L - L_M$ to an unmatched vertex v with $d(v) = d(u^*)$ that is vertex disjoint from $P_1, \ldots, P_k$ (meaning, its vertices were never marked as forbidden by the DFS search). This path cannot be a path of the DFS forest (as it was otherwise discovered). Hence, by the white path theorem, at time $\hat{d}(u)$, one of the vertices on this path must be Black (there are no *back* edges, hence we never encounter Gray vertices). However, in this DFS search vertices only turn black (without being marked as forbidden) if none its descendants in $\hat{G}$ is an unmatched vertex. As v was unmatched we get a contradiction.

DISJOINT-SETS DATA STRUCTURES

1 Disjoint-sets operations and the linked-list representation

In many cases one needs to keep track of a number of sets $S_1, \ldots, S_k$ that are disjoint, that is, $S_i \cap S_j = \emptyset$ for $i \neq j \in \{1, \ldots, k\}$. In this course we will mainly rely on such a structure when discussing Kruskal's algorithm to find minimum weight spanning trees. The idea is that the contents and number of these sets can vary while performing the algorithm via the MakeSet, FindSet and Union operation:

1. MakeSet(x): This operation creates a set holding exactly one item x.

2. FindSet(x): Searches the set in which item x is located.

3. Union(x, y): Takes two sets as input, being the set holding x and y respectively, and creates a single set by taking all items of both sets as its contents.

Each set is also said to have a representative (typically called the *root* item). A MakeSet(x) operation automatically makes x the representative, while a Union operation typically selects one of the two representatives to become the new representative. The FindSet does not alter the representative.

A first implementation of the disjoint-sets data structure is to maintain a linked-list per set S_i: this list links all items by placing a pointer from item i to $i-1$, where item 0 is the representative. Further, all items also carry a pointer to the representative. This pointer, called the root pointer, allows us to perform the FindSet operation in time $O(1)$ as the root item (representative) carries the identifier of the set. The MakeSet operation obviously also runs in $O(1)$ time. The Union operation is somewhat more involving as we need to append one linked list, say holding x, to the back of the other list, holding y, (which can be done in $O(1)$ time by keeping a pointer to the last element) and to update the root pointers of all the entries in the list of x.

In order to reduce the amount of updates required by the UNION operation, it is best to append the *shorter* list to the *longer* one. This leads to the following theorem:

THEOREM 1.1. *The worst-case overall cost of a sequence of m operations, including exactly n* MAKESET *operations is $O(m + n \log n)$ using a linked list representation.*

Proof. The only operation that costs more than $O(1)$ is the UNION operation which requires us to adapt multiple root pointers. Consider an item x. Its root pointer is adapted whenever its list is appended to a list holding y that contains at least as many items as the list of x.

As the list of x is initially size 1, it is at least size 2 after the first adaptation of the root pointer of x, size 4 after the second and eventually size 2^k after the k-th adaptation. Thus, as there are only n elements, the root pointer of x can be adapted at most $\log n$ times (as after $\log_2 n$ adaptations, the list has a size of n or more). Hence, the total cost of all the UNION operations is at most $n \log n$. □

EXERCISES 1.1. *Disjoint-sets and Linked lists:*

1. *Assume we do not append the* shortest *to the* longer *list, meaning we do not need to keep track of the length of the list. Give an example of m operations, with at most n* MAKESET *operations, that requires $\Theta(m^2)$ time.*

2 Disjoint-sets forest

Instead of storing each disjoint set as a linked-list, one can also store them as a tree, creating a disjoint-set forest. In this case, the representative will be the root of the tree and each item x holds a pointer $p(x)$ to its parent in the tree, where the representative has $x = p(x)$.

The advantage is that we can now perform a UNION operation by simply making the root of one tree a child of the root of the other tree. However, a FINDSET(x) operation requires us to follow the $p(x)$ pointers until we reach the root item that carries the set identifier. Notice, each item does not carry the set identifier as all these values would require an update when performing a UNION operation.

To achieve a good overall performance, two heuristics are used to improve the performance of a disjoint-set forest. First, all items keep track of their **rank**, which is initially 0 and may increase due to a UNION operation. The variable rank(x) will hold the length of the longest path from x to one of its

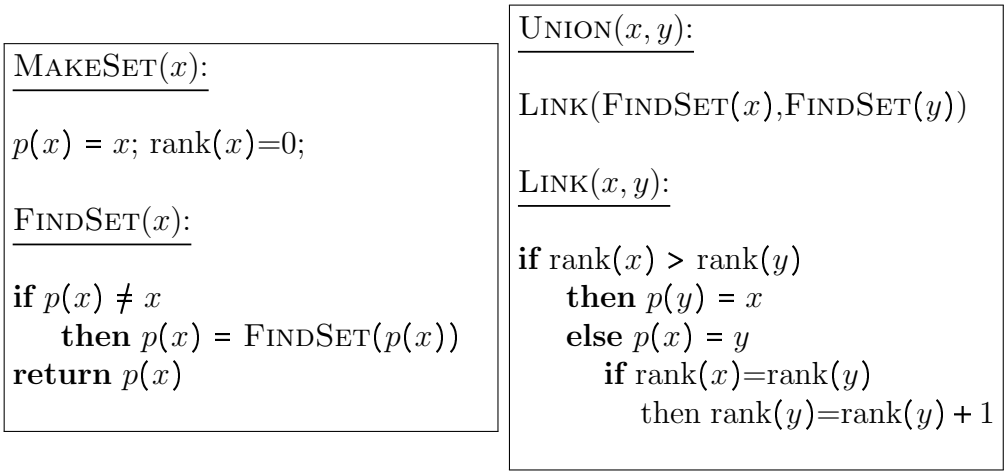


Fig. 12: The MakeSet, FindSet and Union operation

descendants in the tree (or an upper bound in case it is combined with path compression). Whenever we link two trees, the tree whose root item has the lowest rank is appended to the one with the larger rank. This heuristic is called the **union by rank** heuristic.

Second, while traversing the path from an item x to the root, we could afterward also adapt all the pointers $p(x)$ on this path such that they directly point to the root item. Hence, any future FindSet operation on one of these items will be performed more efficiently. We will refer to this improvement by **path compression**.

Pseudo code for each of the operations is given in Figure 12. The function FindSet uses a two-pass method as we first track down the root element by following the $p(x)$ pointers, while upon return we set all the $p(x)$ pointers equal to the root element.

Let us first look at the efficiency if we only make use of the **union by rank** heuristic:

Theorem 2.1. *The worst-case overall cost of a sequence of m operations, including exactly n MakeSet operations is $O(m \log n)$ using a disjoint-set forest and the union by rank heuristic.*

Proof. The only operation that costs more than $O(1)$ is the FindSet operation where we need to traverse multiple $p(x)$ pointers to find the root (and the FindSet calls needed by the Union calls). Consider an item x, whose rank is initially equal to 0. In order for x to increase its rank from k to $k+1$, a link between two rank k trees is required. Thus, by induction, if x gets

rank k it is the root of a tree with at least 2^k items. As there are n items in total, $2^{\text{rank}(x)} \leq n$ or $\text{rank}(x) \leq \log n$.

Moreover, by induction we easily see that the length of the longest path from the root x to a descendant equals $\text{rank}(x)$ (as there is no path compression). Meaning, any FINDSET operation takes at most $\log n$ time. Resulting in a total cost of $O(m \log n)$ □

Let us now analyze the performance when the **path compression** is combined with union by rank. First, we observe the following properties:

1. If $p(x) \neq x$ then $\text{rank}(p(x)) > \text{rank}(x)$ and if $p(x)$ is updated then $\text{rank}(p(x))$ only increases.

2. The number of rank k elements is at most $n/2^k$.

The first property is immediate from the algorithm. The second from repeating the argument of the previous proof to show that a rank k root item x requires at least $2^k - 1$ items each having a rank less than k (being its descendants). Notice, due to the path compression, the number of descendants of an arbitrary rank k item x may no longer be larger or equal to 2^k. Finally, any two rank k items never share a descendent.

To express an upper bound on the efficiency we define the $\log_*$ function. The result of $\log_* n$ equals the number of times we need to apply $\log_2$ to n in order to get a value equal to or below 1. For instance, $\log_* 2 = 1$, $\log_* 2^2 = \log_* 4 = 2$, $\log_* 2^4 = \log_* 16 = 3$, $\log_* 2^{16} = \log_* 65536 = 4$, $\log_* 2^{65536} = 5$, etc. Thus, even for n very large, $\log_* n$ is small. In practice it is at most 5 as $n \leq 2^{65536}$ always applies. We now prove the following theorem:

THEOREM 2.2. *The worst-case overall cost of a sequence of m operations, including exactly n* MAKESET *operations is $O(m \log_* n)$ using a disjoint-set forest, the union by rank and the path compression heuristic.*

Proof. We start by partitioning the set of all values a rank can have, being 0 to $\lfloor \log n \rfloor$, into a number of blocks by stating that rank k belongs to block r if $\log_* k = r$. Meaning there are at most $1 + \log_*(\log n) = \log_* n$ blocks (block 0 to $\log_*(\log n)$). Next, we determine the maximum cost of executing m FINDSET operations. Consider a path $x_0, x_1, \ldots, x_l$ from $x = x_0$ to x_l the root of the tree taken as a result of performing a FINDSET operation. The cost of following this path is split in two parts:

(1) We will call the cost of visiting x_i, with $\text{rank}(x_i)$ in block r, a block cost if the rank of x_{i+1} is not part of block r (but of block $s > r$, meaning the block number of the parent's rank is larger) or if $x_{i+1} = x_l$ (its parent is the root). As the rank increases along the path $x_0, \ldots, x_l$ and we only have

$\log_* n$ blocks, the block cost of a single FindSet operation is therefore at most $\log_* n + 1$ (at most $\log_* n - 1$ due to the block boundaries and two for visiting the child of the root and the root itself). Meaning the overall block cost is $O(m \log_* n)$.

(2) The cost of visiting x_i when the previous condition is not met, is called a path cost. A first observation is that once x_i causes a block cost, it will never again cause a path cost (except for the root and its child). The reason being that if the rank of $p(x_i)$ belongs to block $s > r$, any update on $p(x_i)$ (due to path compression) will only further increase the rank of $p(x_i)$, while the rank of x_i will remain identical (as it is not the root).

Let us now determine the maximum path cost a rank k item x can experience. An item is said to be a rank k if it has rank k while becoming a child of some other node (before this occurs, x has no path costs). Assume rank k belongs to block r. Let $B(r)$ be the largest rank $j \in \mathbb{N}$ such that j belongs to block r. By definition $B(0) = 1$ and $B(r+1) = 2^{B(r)}$. As the rank of $p(x)$ increases by at least one each time a path cost occurs by visiting x (because x is NOT a child of the root x_l), the path cost of x is bounded by $B(r) - B(r-1)$. By property 2 we can have at most

$$\sum_{k=B(r-1)+1}^{B(r)} n/2^k = n/2^{B(r-1)+1} \sum_{k=0}^{B(r)-B(r-1)-1} 1/2^k < n/2^{B(r-1)} = n/B(r)$$

items with a rank belonging to block $r > 0$ and at most n items with a rank in block 0. Thus, in general, we have at most $n/B(r)$ items in block r, each having a path cost bounded by $B(r) - B(r-1)$. Hence, the total path cost is bounded by

$$\sum_{r=0}^{\log_* n - 1} n/B(r)(B(r) - B(r-1)) < n \log_* n.$$

This proves that all m FindSet operations have a total cost of at most $O(m \log_* n)$. As all other operations are either $O(1)$ or require 2 FindSet operations, the proof is complete. □

It is possible to improve this bound slightly by showing that the runtime is at most $O(m\alpha(m,n))$, where $\alpha(m,n)$ is the inverse Ackermann function. The inverse is defined as $\min\{i | A(i, \lfloor m/n \rfloor) > \log n\}$. This function grows even more slowly than the $\log_*$ function. In practice, one finds that $\alpha(m,n) \leq 4$ always applies.

Exercises 2.1. *Disjoint-sets and forests:*

1. *Suppose a tree in a disjoint-sets forest implementation contains* 6 *items. Draw all the possible structures of such a tree with and without path compression and explain how these trees can be constructed (by listing the operations used).* ☆

2. *Give an example of a sequence of operations in a disjoint-sets forest implementation such that a tree T_p becomes a child of T_r, while the longest path in T_p is larger than the longest path in T_r.* ☆

3. *Bob and Alice play a game where Bob picks an integer $n \geq 1$ that Alice needs to guess. During each round Alice can write down as many guesses as she likes. However, if Alice writes more than n numbers, Bob wins. After each round Bob indicates which of the numbers written down by Alice are smaller than n. If Alice wrote n as one of the guesses, Alice wins. Indicate how Alice can always win in $O(log_* n)$ rounds.* ☆

4. *Suppose we replace the union-by-rank with a union-by-weight heuristic. In this case each set stores the number of elements in its set and when a union occurs the smaller set becomes a child of the larger set (breaking ties arbitrarily). Show that union-by-rank and union-by-weight are not equivalent. What is the worst case time complexity of a* FindSet *operation for the union-by-weight heuristic?* ☆

5. *Given a graph $G = (V, E)$, when does the algorithm below return True? Should we use a linked-list or forest structure for its implementation?*

```
for u ∈ V do
    MakeSet(u);
end for
Select random e' = (u', v') ∈ E; E' = E \ e'.
while E' ≠ ∅ do
    for e = (u, v) ∈ E' do
        if FindSet(u) = FindSet(v) then
            Return False;
        else
            if FindSet(u) = FindSet(u') then
                Union(FindSet(v),FindSet(v')); E' = E' \ e;
            end if
            if FindSet(u) = FindSet(v') then
                Union(FindSet(v),FindSet(u')); E' = E' \ e;
            end if
            if FindSet(v) = FindSet(u') then
                Union(FindSet(u),FindSet(v')); E' = E' \ e;
```

```
            end if
            if FindSet(v) = FindSet(v') then
                Union(FindSet(u),FindSet(u')); E' = E' \ e;
            end if
        end if
    end for
end while
Return True
```

6. *Suppose G is a graph with k connected components. Give a simple algorithm to determine these components. How often do we need to execute the* MakeSet, FindSet *and* Union *operation (as a function of k, $|V|$ and $|E|$)?* ☆

7. *Show that any sequence of m* MakeSet, FindSet *and* Union *operations runs in $O(m)$ time if all the* Union *operations take place before the* FindSet *operations (and are executed on root elements), given that path compression and union by rank is used.*

SPANNING TREES

1 Basic properties and definitions

A connected graph is called a **tree** if the removal of any of its edges makes the graph disconnected. A tree T is called a subtree of some graph G if $T \subseteq G$. A subtree $T \subseteq G$ is said to be a **spanning tree** if it contains all the vertices of G, i.e., $V(T) = V(G)$. We call a graph G weighted if G has a weight function $\alpha : E(G) \to \mathbb{R}^+$ (positive reals) on its edges, where $\alpha(e)$ denotes the weight of $e \in E(G)$. For each spanning tree T, we can define its **weight** as the sum of the weights of its edges:

$$\alpha(T) = \sum_{e \in E(T)} \alpha(e).$$

Whenever we wish to build a network connecting n nodes (towns, computers, chips in a computer) it is desirable to decrease the cost of construction of the links to the minimum. In graph theoretical terms, we wish to find an a spanning tree T of a weighted graph G, with $\alpha(T)$ minimal. We will discuss two important algorithms to determine optimal spanning trees: Kruskal's algorithm and Prim's algorithm.

Minimum weight spanning trees are useful in a variety of disciplines. For instance, minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire. Much of the work on minimum spanning (and related Steiner) trees has been conducted by the phone company. They also provide a reasonable way for clustering points in space into natural groups. When the cities are points in the Euclidean plane, the minimum spanning tree provides a good heuristic for traveling salesman problems. The optimum traveling salesman tour is at most twice the length of the minimum spanning tree.

Before we present Kruskal's and Prim's algorithm, let us remark that we can characterize whether a graph G is a tree in a variety of ways (we do not include a proof as these statements are visually clear).

THEOREM 1.1. *The following statements are equivalent:*

1. *G is a connected graph where the removal of any edge of G makes G disconnected.*
2. *There exists a unique path between every two vertices of G.*
3. *G does not hold any closed paths and for $n > 2$ every additional edge creates a closed path in G.*
4. *G is connected and contains no closed path.*
5. *G does not contain a closed path and $e = n - 1$.*
6. *G is connected and $e = n - 1$.*

2 Kruskal's algorithm (1956)

Let us now consider the following algorithm to compute a spanning tree T of a weighted graph G with minimum weight $\alpha(T)$.

Algorithm 2.1 (Kruskal's algorithm (1956))**.** *For a connected and weighted graph G^α of order n:*

Step 1:

Order the set of edges $E(G)$ by nondecreasing weight, set $i = 1$.

Step 2:

Set $E(T) = \emptyset$ and for each edge $(u, v) \in E(G)$ in order by nondecreasing weight, check whether $E(T) + (u, v)$ contains a closed path. If there is no closed path, set $E(T) = E(T) + (u, v)$, $e_i = (u, v)$ and $i = i + 1$.

The final outcome $T = (V(G), E(T) = \{e_1, \ldots, e_{n-1}\})$ is a spanning tree with minimum weight.

Notice, $E(T)$ need not to be connected during the execution of the algorithm. An example of Kruskal's algorithm on a graph with 13 vertices is shown in Figure 13.

Before we prove the correctness, let us have a look at the complexity. Step 1 orders the edges, which can be done in $O(|E| \log |E|)$ time. For Step 2, we can rely on the disjoint-sets data structure. Indeed, initially we perform $n = |V(G)|$ MakeSet operations, one for each node $v \in V(G)$. This creates n trees each having 1 vertex. Next, each time we need to determine whether there is a closed path by adding an edge (u, v) to $E(T)$, we test whether FindSet(u) = FindSet(v). If not, we perform a Union(u, v) to merge the

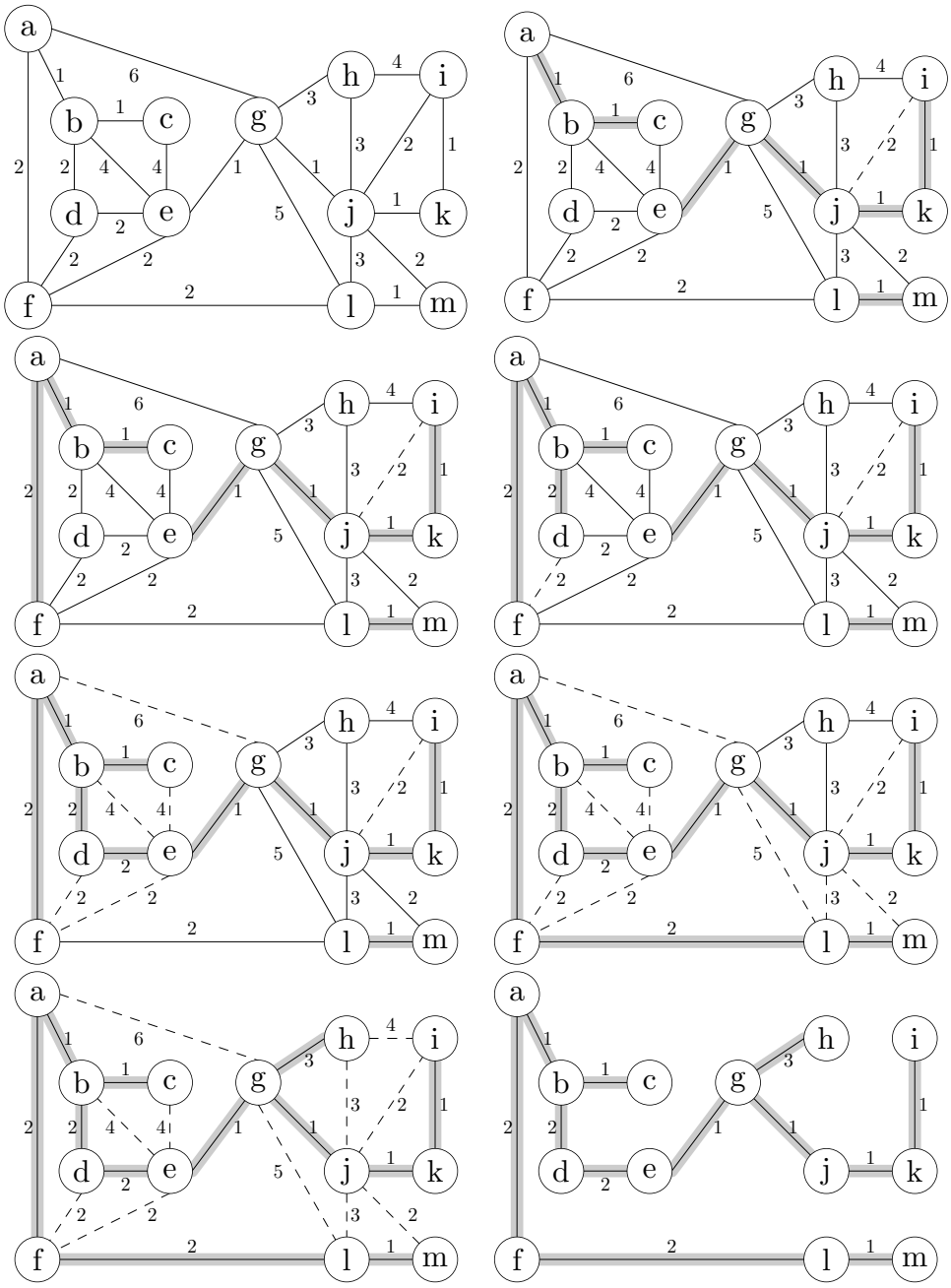


Fig. 13: Example of Kruskal's algorithm.

component of u with the component of v. If u and v did belong to the same set, this implies that there already existed a path from u to v; hence, adding (u, v) creates a closed path.

Thus, using a disjoint-sets forest structure, we can implement Step 2 in time $O(|E| \log_* |V|)$ as we perform at most $|V|$ MakeSet operations, $2|E|$ FindSet operations and $|V| - 1$ Union operations.

Theorem 2.1. *Kruskal's algorithm when applied on a connected weighted graph G^α returns a minimum spanning tree T.*

Proof. We first argue that $T = (V(G), E(T))$ is a spanning tree of G. By construction it contains no cycles. Further, T must be connected, otherwise T has components C_1 and C_2. Any edge (u, v) with $u \in C_1$ and $v \in C_2$, should have been added to T as it does not create a closed path (such an edge exists as G is connected).

Next, we show that T has the minimum total weight among all spanning trees of G. Suppose T_1 is a spanning tree of G. Let e_k be the first edge produced by the algorithm that is not in T_1, meaning $\{e_1, \ldots, e_{k-1}\} \in E(T_1)$. If we add e_k to T_1, then a cycle C containing e_k is created. Also, C must contain an edge e that is not in T (as T is a tree and therefore has no cycles). Replace e by e_k in T_1, to obtain a new spanning tree $T_2 = T_1 + e_k - e$ (as T_2 is still connected and has $|E(T_2)| = |V(T_2)| - 1$).

However, by construction, $\alpha(e_k) \leq \alpha(e)$, because $\{e_1, \ldots, e_{k-1}, e\} \subseteq E(T_1)$ does not contain a cycle. Therefore, $\alpha(T_2) \leq \alpha(T_1)$. Note that T_2 has more edges in common with T than T_1. Repeating the above procedure, we can transform T_1 to T by replacing edges, one by one, such that the total weight does not increase. We deduce that $\alpha(T) \leq \alpha(T_1)$. □

The outcome of Kruskal's algorithm need not be unique. Indeed, often multiple optimal spanning trees exist (with the same weight, of course) for a single weighted graph G^α. The reason why we may have different outcomes lies in Step 1 of the algorithm. That is, when we order the edges in a nondecreasing manner, there are several possible orders if some of the edges have identical weights.

3 Prim's algorithm (1957)

Another algorithm for constructing a minimum spanning tree is Prim's algorithm.

Algorithm 3.1 (Prim's algorithm (1957))**.** *For a connected and weighted graph G^α of order n with $V(G) = \{v_1, \ldots, v_n\}$*

Step 1:

Define $f(v_1) = 0$ and $f(v_i) = \alpha((v_1, v_i))$ if $(v_1, v_i) \in E(G)$ and $f(v_i) = \infty$ otherwise (for $i > 1$). Let $E(T) = \emptyset$ and set $U = \{v_1\}$.

Step 2:

Choose $w \in V(G) - U$ with $f(w)$ minimal. Replace $E(T)$ by $E(T) \cup \{e\}$, where e is an edge incident to w and U for which $\alpha(e) = f(w)$ and set $U = U \cup \{w\}$. If $U = V(G)$ stop.

Step 3:

For each $v \notin U$ for which $wv \in E(G)$: $f(v) = \min\{f(v), \alpha(w,v)\}$. Return to Step 2.

The final outcome $T = (V(G), E(T))$ is a spanning tree with minimum weight.

Notice, $f(v)$ equals the minimal weight of any edge connecting v with U ($f(v)$ equals ∞ if there is no such edge). Thus, at each iteration we simply add a new edge e to T with a weight $\alpha(e)$ as small as possible. An example of Prim's algorithm per two iterations is given in Figure 14.

A straightforward implementation of Prim's algorithm has a runtime complexity of $O(|V|^2)$. In the next subsection we indicate how a binary heap can be used to achieve a $O(|E| \log |V|)$ runtime complexity. In the next chapter, we further demonstrate that Prim's algorithm can be implemented in $O(|E| + |V| \log |V|)$ time using Fibonacci heaps). Thus, Prim's algorithm is faster on dense graphs, while Kruskal's is faster on sparse graphs. Prim's algorithm was also invented earlier by Jarnick (1930). Many other algorithms exist to solve this problem, e.g., Boruvka's algorithm (1926). The first randomized linear algorithm was proposed by Karger, Klein and Tarjan (and thus runs in an expected time of $O(|E|)$). The fastest deterministic algorithm developed thus far, with a known time complexity, is by Chazelle (2000) and makes use of the soft heap data structure. It has a runtime of $O(|E|\alpha(|E|, |V|))$, with α the inverse of the Ackermann function. An algorithm with a proven optimal runtime complexity was established by Pettie and Ramachandran (2002), but its runtime complexity is still unclear, obviously it resides between $O(|E|)$ and $O(|E|\alpha(|E|, |V|))$.

Let us first prove the correctness of Prim's algorithm.

Theorem 3.1. *Prim's algorithm when applied on a connected weighted graph G^α returns a minimum spanning tree T.*

Proof. Let G^α be a connected, weighted graph. By induction we can show that $T = (U, E(T))$ is a spanning tree on the vertices of U during each iteration. Indeed, at step 1 we have $|U| = 1$, $|E(T)| = 0$, hence, it is a spanning tree on $U = \{v_1\}$. At every other iteration we augment $|U|$ and

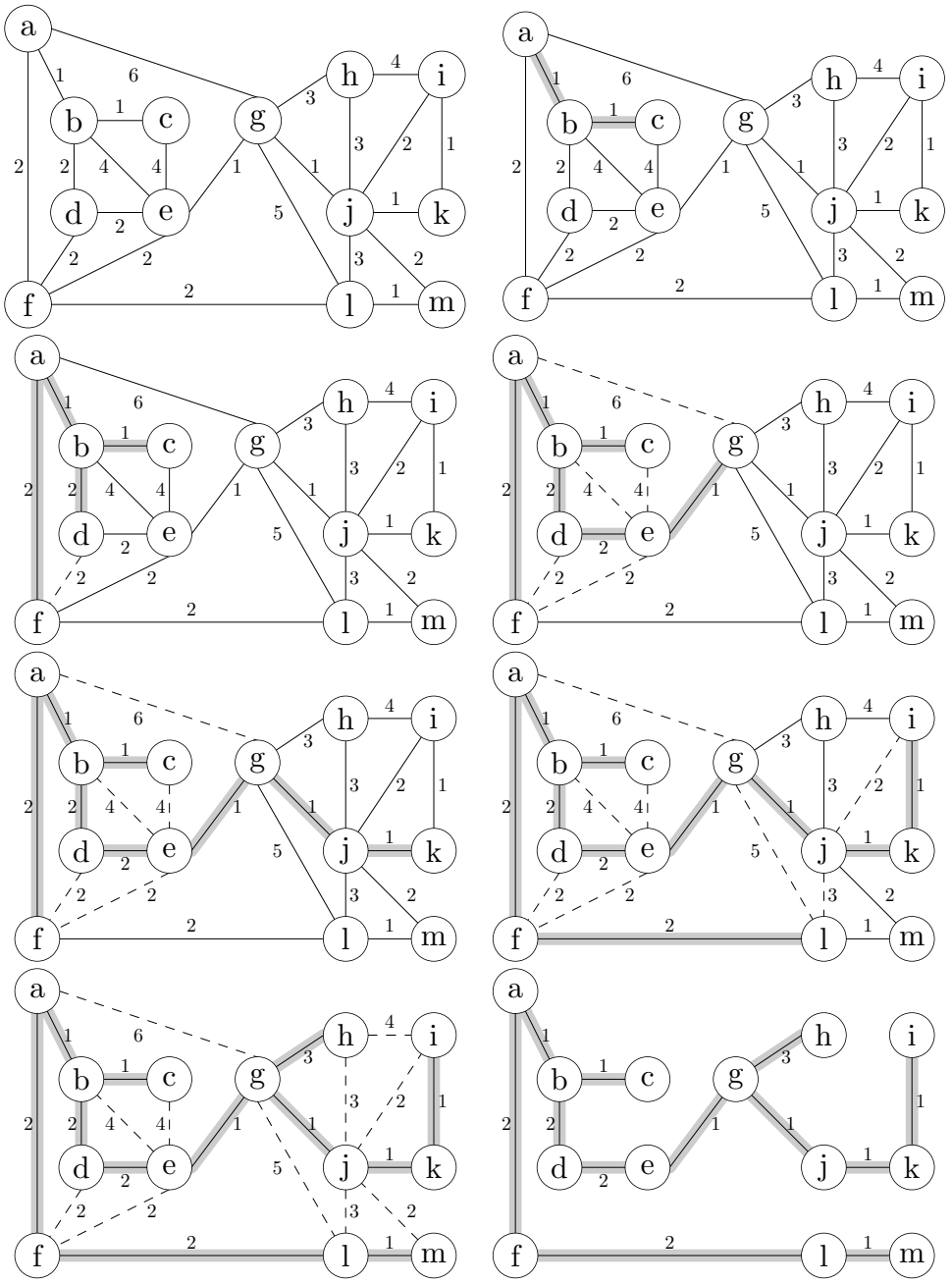


Fig. 14: Example of Prim's algorithm.

$|E(T)|$ by one, meaning $|U| = |E(T)| + 1$ and T is connected. As a result, T is a spanning tree.

Let T_1 be a spanning tree for G. Let $e = (u, v)$ be the first edge not

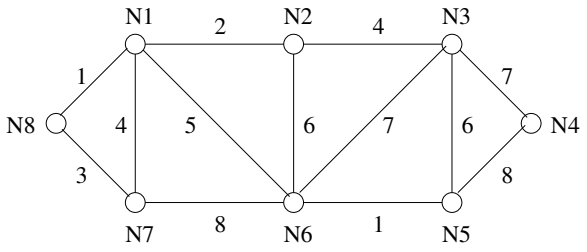


Fig. 15: Apply Kruskal's and Prim's algorithm on the graph G

belonging to T_1 that was added when $T = (U, E(T))$ was constructed. Then, by construction, one endpoint u of $e = (u, v)$ was in U and the other v was not. Since T_1 is a spanning tree of G, there is a path in T_1 joining the two endpoints u and v of e. As one travels along this path, one must encounter an edge $f = (u', v')$ joining a vertex $u' \in U$ to $v' \notin U$ (note, it could be that $v' = v$ or $u' = u$, but not both). Now, at the iteration when $e = (u, v)$ was added to $E(T)$, $f = (u', v')$ could also have been added and it would be added instead of $e = (u, v)$ if $\alpha(f) < \alpha(e)$. Since $f = (u', v')$ was not added, we conclude that $\alpha(f) \geq \alpha(e)$. Let $T_2 = T_1 + e - f$, then $\alpha(T_2) \leq \alpha(T_1)$ and T_2 is a spanning tree as T_2 is connected and the number of vertices still matches the number of edges plus one.

Repeating the above procedure, we can transform T_1 to T by replacing edges, one by one, such that the total weight does not increase. We deduce that $\alpha(T) \leq \alpha(T_1)$. □

Exercises 3.1. *On Spanning Trees, Kruskal's and Prim's Algorithm:*

1. *Show that for any edge $e \in E(G)$, with G connected, there exists at least one spanning tree T that contains e.*

2. *Apply Kruskal's and Prim's Algorithm to the graph G depicted in Figure 15.*

3. *Let T_1 and T_2 be two spanning trees of minimal weight for G^α. Denote $\alpha^*(T)$ as the largest weight of an edge e belonging to T, i.e., $\alpha^*(T) = \max_{e \in E(T)} \alpha(e)$. Does $\alpha^*(T_1)$ equal $\alpha^*(T_2)$ in general ?*

4. *Assume that G^α does not have a unique minimum spanning tree. Argue that any of these trees can be returned as output of Kruskal's algorithm.*

5. *Argue that any maximum weight edge (u, v) on some closed path C cannot be part of a minimum spanning tree.*

6. *Consider the Reversed Kruskal algorithm: Set $T = E$, go through E in decreasing order of edge weights. For each edge e, check if deleting e disconnects the graph. If not, remove e from T. Prove that this algorithm produces a minimum spanning tree. [Hint: show that T contains a minimum spanning tree during each step of the algorithm]* ☆

7. *Assume we wish to find a minimum spanning tree with the additional constraint that the degree of v in T is bounded by $b(v)$. Can we use a modified version of Kruskal's algorithm that only adds an edge to T if it creates no closed path and does not violate a degree bound to solve this problem? Prove or give a counterexample.* ☆

8. *True or false: Assume p is a path of minimum weight between u and v, then there exists a minimum spanning tree T that contains p.* ☆

9. *Let $G = (V, E)$ be an undirected connected graph and $V_1, V_2 \subset V$ such that $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. Let $G_i = (V_i, E_i)$, for $i = 1, 2$, with $E_i = \{(u, v) \in E | u, v \in V_i\}$. Suppose e is such that $\alpha(e) = \min_{u \in V_1, v \in V_2} \alpha((u, v))$. Let T_i be an MST for G_i, for $i = 1, 2$. Prove that $T_1 \cup T_2 \cup e$ is an MST for G or give a counterexample.* ☆

10. *Let T_1 be an minimum spanning tree for G. Give an $O(|V||E|)$ algorithm that uses T_1 as input to create a spanning tree $T_2 \neq T_1$ for G such that $\alpha(T_2) \leq \alpha(T)$ for any spanning tree $T \neq T_1$.* ☆

11. *Let T be a minimum spanning tree for G. Design an $O(|E|)$ algorithm that constructs a minimum spanning tree from T if (a) the weight of an edge e is decreased, (b) the weight of an edge e is increased.* ☆

12. *Develop a fast algorithm to compute a maximum weight spanning tree.* ☆

13. *We label (u, v) as a light edge if there exists a cut $(S, V \setminus S)$ such that $u \in S$, $v \notin S$ and there is no edge between S and $V \setminus S$ with a weight smaller than $\alpha(u, v)$. True or false?* ☆

    (a) *If (u, v) is part of a minimum spanning tree, it must be a light edge.*

    (b) *The set of all the light edges of G forms a minimum spanning tree.*

14. *Let (u, v) be an edge with a weight strictly smaller than the weight of any other edge that is connected to u. Prove that (u, v) must be part of any minimum spanning tree.* ☆

15. *Adding an edge (u, v) to a spanning tree T creates a closed path $C_{(u,v)}$ in $T + (u, v)$. Prove that a spanning tree $T \subseteq G^{\alpha}$ is a minimum spanning tree if and only if for each edge $(u, v) \in E(G) - E(T)$, (u, v) has the maximum weight of all the edges part of $C_{(u,v)}$.*

3.1 Prim's algorithm and binary heaps

This section is not intended as an introduction to binary heaps, but briefly summarizes the main characteristics and properties of binary heaps. A detailed discussion should have appeared in some other courses. A binary heap is a **complete** binary tree[1], where each vertex v in the tree has a **key value** $k(v)$ such that *the value of any parent vertex is smaller than or equal to the key value of both its children.* Thus, the root of the tree holds the element with the smallest key value. A binary heap with n elements is typically stored in an array A of size n. Children of vertex i will be stored in entries $2i$ and $2i + 1$, whereas the parent of vertex i is stored in entry $\lfloor i/2 \rfloor$. There are four important heap operations in the context of Prim's algorithm.

(1) The first operation concerns the HEAPIFY(A, i) operation. It is invoked when both the children of i are the root of a binary heap, but the key value $k(i)$ of vertex i is larger than that of one of its children. HEAPIFY(A, i) will exchange vertex i with the smallest child and will repeat this operation recursively until both children have a key value larger than or equal to $k(i)$, the key value of vertex i at the start of the operation. Given that A has size n, the time complexity equals the depth of the tree, i.e., it is $O(\log n)$.

(2) The BUILDHEAP(A) operation takes an array A as input and turns it into a heap, by making sure the heap property holds at the end of the operation. It is implemented by calling the HEAPIFY(A, i) operation for $i = \lfloor length[A]/2 \rfloor$ to $i = 1$. Notice, we only need to call HEAPIFY on the first half of the array A as a leaf vertex is always the root of a heap (with one element). As a HEAPIFY operation takes $O(\log n)$ time, one might conclude that the BUILDHEAP operation takes $O(n \log n)$ time, however, a somewhat more careful analysis shows that the runtime is $O(n)$.

(3) The DELETEMIN(A) operation removes the vertex with the smallest key value, which is the root node. It is implemented by exchanging the root element with the last element in the heap (i.e., the right most leaf at the last level). After removing the root vertex, we perform the HEAPIFY$(A, 1)$ operation to restore the heap structure. The runtime is therefore clearly $O(\log n)$.

[1] Meaning level k of the tree holds 2^k elements, except for the last level, which potentially holds fewer elements.

(4) The last operation of interest is the DecreaseKey(A, i, v) which decreases the key value of i to v. If the new key value is smaller than the key value of its parent, we recursively exchange the vertex i with its parent vertex, until the heap property is restored. This requires at most $O(\log n)$ time.

To implement Prim's algorithm using binary heaps, we need to start by building a heap with $n = |V|$ entries. The key value of a vertex v is initialized as $f(v)$ (according to Step 1 of Prim's algorithm). Subsequently, we perform n DeleteMin operations, to find the vertex with the smallest $f(v)$ value and after each DeleteMin operation we perform a DecreaseKey operation on each of the neighbors (in the graph G) of the deleted vertex. Hence, at most $|E|$ DecreaseKey operations are needed, resulting in an overall time complexity of $O(|V| + |E| \log |V|)$.

3.2 Graph preprocessing for sparse graphs

As we will demonstrate further on, the runtime of Prim's algorithm is $O(|E| + |V| \log |V|)$ using Fibonacci heaps. For sparse graphs G we can gain some time by preprocessing G before running Prim's algorithm. The idea is to create a smaller graph G' from G and to run Prim's algorithm on G'. The minimum spanning tree T' of G' is then extended by an edge set T created during the preprocessing phase to find a minimum spanning tree $T + T'$ for G. The pseudo code for the preprocessing phase is given in Figure 16, where initially $w(u, v) = \alpha((u, v))$ and $orig(u, v) = (u, v)$.

The first for-loop initializes all vertices v as False and creates $|V(G)|$ sets with one entry. The second for-loop visits each vertex u and selects the edge (u, v) with minimal weight (by adding it to T) and unifies the set of u and v. Notice, both vertices become True after this step, preventing us from visiting v (if v was not visited thus far). Notice, the set of u only holds one entry when performing the Union(u, v) operation. G' gets as many vertices as there are sets S_i left after the second for-loop. The final for-loop identifies for each couple of sets (S_i, S_j) the edge (x, y) in $E(G)$ such that $x \in S_i$, $y \in S_j$ and $\alpha((x, y))$ minimal.

An example of a preprocessing step on a graph with 13 vertices can be found in Figure 17. One can prove that the solution T' to the minimum spanning tree problem for $(G')^w$ forms a minimum spanning tree for G^α when we add T to T'. Although the proof is elementary, we will not treat it within the scope of this course.

The important thing to notice here is that we may apply the Preprocess procedure a number of times by taking the output of one phase as the input for the next. In this manner we reduce the graph G' step by step. We

```
PREPROCESS(G, T):

for each v ∈ V(G)
    do mark(v) =FALSE; MAKESET(v);
for each u ∈ V(G)
    do if mark(u) = FALSE
       then choose v ∈ Γ(u) with w(u, v) minimal
            UNION(u, v); T = T + orig(u, v)
            mark(u) = mark(v) =TRUE
V(G') = {FINDSET(v)|v ∈ V(G)}
E(G') = ∅
for each (x, y) ∈ E(G)
    do u =FINDSET(x); v =FINDSET(y)
       if u ≠ v
          if (u, v) ∉ E(G')
             then E(G') = E(G') + (u, v)
                  orig(u, v) = orig(x, y); w(u, v) = w(x, y)
          else if w(x, y) < w(u, v)
             then orig(u, v) = orig(x, y); w(u, v) = w(x, y)
construct the adjacency lists Γ(u) for G'
return G' and T
```

Fig. 16: The Preprocessing operation

could even repeat this procedure until G' contains only one or two vertices, meaning the spanning tree problem for G' is trivial (doing so actually corresponds to Boruvka's 1926 algorithm used to construct an efficient electricity network for Moravia). However, as the next exercise indicates we can find the best asymptotic result by performing the PREPROCESS procedure a specific number of times before running Prim's algorithm with a Fibonacci heap data structure.

EXERCISES 3.2. *On Preprocessing and Prim's Algorithm:*

1. *Explain how you can implement the* PREPROCESS *procedure in* $O(|E|)$ *time. There is no need to rely on the advanced implementations of disjoint data sets.*

2. *Argue that* $|V(G')| \leq |V(G)|/2$*, meaning* G' *contains at most half as many vertices as* G*.*

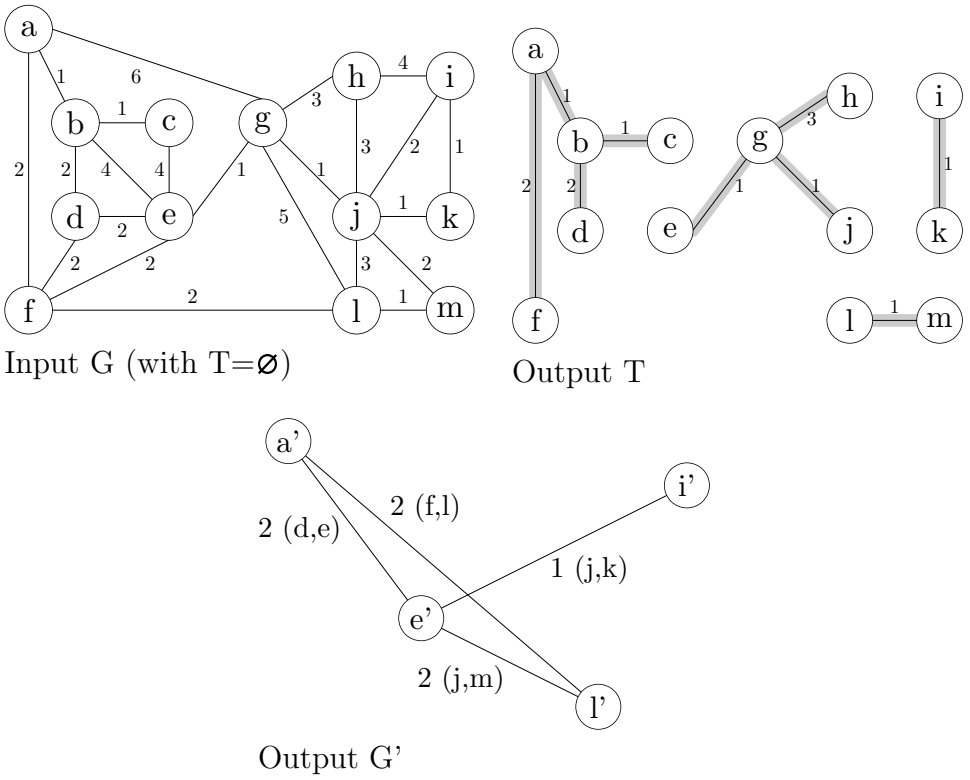


Fig. 17: Example of a preprocessing step.

3. *What would be the runtime complexity if we simply repeated the* PREPROCESS *procedure until G' contains only one or two vertices?*

4. *If we wish to get a $O(|E| \log\log |V|)$ runtime complexity, how often do we need to perform the* PREPROCESS *procedure before running Prim's algorithm? Explain why this results in a $O(|E| \log\log |V|)$ runtime.*

5. *Given a graph G, which algorithm and data structure results in the best asymptotic time complexity to determine a minimal spanning tree if:* ☆

   (a) *The number of edges is linear in $|V|$ (e.g., $|E| = 20|V|$).*

   (b) *The number of edges is quadratic in $|V|$ (e.g., $|E| = |V|^2/10$).*

   (c) *The number of edges is of the form $c|V| \log |V|$ (e.g., $|E| = 3|V| \log |V|$).*

   (d) *All the edge capacities are integers between 1 and 100.*

FIBONACCI HEAPS

1 Introduction

Priority queues are a classic topic in theoretical computer science. The search for a fast priority queue implementation is motivated primarily by two network optimization algorithms: Shortest Path (SP) and Minimum Spanning Tree (MST), i.e., the connector problem. As we shall see, Fibonacci Heaps provide a fast and elegant solution.
The following 3-step procedure shows that both Dijkstra's SP-algorithm or Prim's MST-algorithm can be implemented using a priority queue:

1. Maintain a priority queue on the vertices $V(G)$.
2. Put s in the queue, where s is the start vertex (Shortest Path) or any vertex (MST). Give s a *key* of 0. Add all other vertices and set their key to infinity.
3. Repeatedly delete the minimum-key vertex v from the queue and mark it *scanned*. For each neighbor w of v do: If w is not scanned (so far), decrease its key to the minimum of the value calculated below and w's current key:
   - SP: key(v) + length(vw),
   - MST: weight(vw).

The classical answer to the problem of maintaining a priority queue on the vertices is to use a *binary heap*, often just called a *heap*. Heaps are commonly used because they have good bounds on the time required for the following operations: insert $O(\log n)$, delete-min $O(\log n)$, and decrease-key $O(\log n)$, where n reflects the number of elements in the heap.
If a graph has n vertices and e edges, then running either Prim's or Dijkstra's algorithms will require $O(n \log n)$ time for inserts and deletes. However, in the worst case, we will also perform e decrease-keys, because we may have to perform a key update every time we come across a new edge. This will take $O(e \log n)$ time. Since the graph is connected, $e \geq n$, and the overall time bound is given by $O(e \log n)$. As we shall see, Fibonacci heaps allow us to do much better.

2 Definition and elementary operations

The Fibonacci heap data structure invented by Fredman and Tarjan in 1984 gives a very efficient implementation of the priority queues. Since the goal is to find a way to minimize the number of operations needed to compute the MST or SP, the kind of operations that we are interested in are *insert*, *decrease-key*, *link*, and *delete-min* (we have not covered why *link* is a useful operation yet, but this will become clear later on). The method to achieve this minimization goal is laziness - *do work only when you must, and then use it to simplify the structure as much as possible so that your future work is easy.* This way, the user is forced to do many cheap operations in order to make the data structure complicated.
Fibonacci heaps make use of heap-ordered trees. A heap-ordered tree is one that maintains the heap property, that is, where $key(parent) \leq key(child)$ for all nodes in the tree.

Definition 2.1. *A Fibonacci heap H is a collection of heap-ordered trees that have the following properties:*

1. *The roots of these trees are kept in a doubly-linked list (the root list of H),*
2. *The root of each tree contains the minimum element in that tree (this follows from being a heap-ordered tree),*
3. *We access the heap by a pointer to the tree root with the overall minimum key,*
4. *For each node x, we keep track of the degree (also known as the order or rank) of x, which is just the number of children x has; we also keep track of the mark of x, which is a Boolean value whose role will be explained later.*

For each node, we have at most four pointers that respectively point to the node's parent, to one of its children, and to two of its siblings. The sibling pointers are arranged in a doubly-linked list (the *child list* of the parent node). We have not yet discussed how the operations on Fibonacci heaps are implemented, and their implementation will add some additional properties to H. The following are some elementary operations used to maintain Fibonacci heaps:

Inserting a node x: We create a new tree containing only x and insert it into the root list of H; this is clearly an $O(1)$ operation.

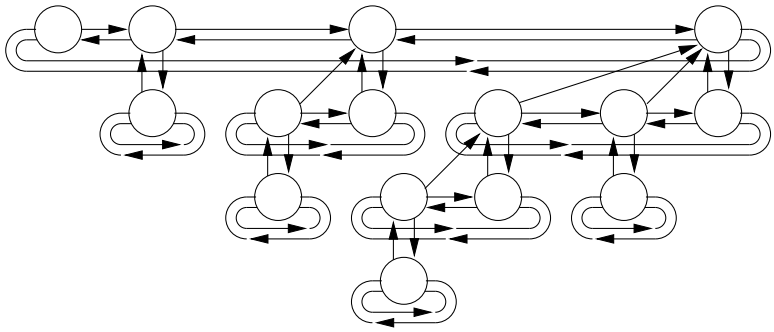


Fig. 18: A detailed view of a Fibonacci Heap. NULL pointers are omitted for clarity.

Linking two trees x and y: Let x and y be the roots of the two trees we want to link; then if $key(x) \geq key(y)$, we make x the child of y; otherwise, we make y the child of x. We update the appropriate node's degrees and the appropriate child list; this takes $O(1)$ operations.

Cutting a node x: If x is a root in H, we are done. If x is not a root in H, we remove x from the child list of its parent, and insert it into the root list of H, updating the appropriate variables (the degree of the parent of x is decremented, etc.). Again, this takes $O(1)$ operations. We assume that when we want to cut/find a node, we have a pointer *hanging* around that accesses it directly, so actually finding the node takes $O(1)$ time.

Marking a node x: We say that x is marked if its mark is set to *true*, and that it is unmarked if its mark is set to *false*. A root is always unmarked. We mark x if it is not a root and it loses a child (i.e., one of its children is cut and put into the root-list). We unmark x whenever it becomes a root. We shall see later on that marked nodes are also cut as soon as they lose a second child.

2.1 The *delete-min* operation

Deleting the minimum key node is a little more complicated. First, we remove the minimum key from the root list and splice its children into the root list. Except for updating the parent pointers, this takes $O(1)$ time. Then we scan through the root list to find the new smallest key and update the parent pointers of the new roots. This scan could take $O(n)$ time in the worst case. To bring down the *amortized* deletion time (see further on), we apply

```
CLEANUP:
newmin ← some root list node
for i ← 0 to ⌊log n⌋
    B[i] ← NULL
for all nodes v in the root list
    parent(v) ← NULL
    unmark v
    if key(newmin) > key(v)
        newmin ← v
    LINKDUPES(v)
```

```
LINKDUPES:
w ← B[deg(v)]
while w ≠ NULL
    B[deg(v)] ← NULL
    if key(w) ≤ key(v)
        swap v and w
    remove w from root list
    link w to v
    w ← B[deg(v)]
B[deg(v)] ← v
```

Fig. 19: The CLEANUP algorithm executed after performing a *delete-min*

a CLEANUP algorithm, which links trees of equal degree until there is only one root node of any particular degree.
Let us describe the CLEANUP algorithm in more detail. This algorithm maintains a global array $B[1 \ldots \lfloor \log n \rfloor]$, where $B[i]$ is a pointer to some previously-visited root node of degree i, or NULL if there is no such previously-visited root node. Notice, the CLEANUP algorithm simultaneously resets the parent pointers of all the new roots and updates the pointer to the minimum key. The part of the algorithm that links possible nodes of equal degree is given in a separate subroutine LINKDUPES, see Figure 19. The subroutine ensures that no earlier root node has the same degree as the current. By the possible swapping of the nodes v and w, we maintain the heap property. We shall analyze the efficiency of the *delete-min* operation further on. The fact that the array B needs at most $\lfloor \log n \rfloor$ entries is proven in Section 4, where we prove that the degree of any (root) node in an n-node Fibonacci heap is bounded by $\lfloor \log n \rfloor$.

2.2 The *decrease-key* operation

If we also need the ability to delete an arbitrary node. The usual way to do this is to decrease the node's key to $-\infty$ and then use *delete-min*. We start by describing how to decrease the key of a node in a Fibonacci heap; the algorithm will take $O(\log n)$ time in the worst case, but the *amortized* time will be only $O(1)$. Our algorithm for decreasing the key at a node v follows two simple rules:

1. If $newkey(v) < key(parent(v))$, promote v up to the root list (this moves the whole subtree rooted at v).

```
PROMOTE:
unmark v
if parent(v) ≠ NULL
    remove v from parent(v)'s child list
    insert v into the root list
    if parent(v) is marked
        PROMOTE(parent(v))
    else
        mark parent(v)
```

Fig. 20: The PROMOTE algorithm

2. As soon as two children of any node w have been promoted, immediately promote w.

In order to enforce the second rule, we now mark certain nodes in the Fibonacci heap. Specifically, a node is marked if exactly one of its children has been promoted. If some child of a marked node is promoted, we promote (and unmark) that node as well. Whenever we promote a marked node, we unmark it; this is the only way to unmark a node (if splicing nodes into the root list during a *delete-min* is considered a promotion). A more formal description of the PROMOTE algorithm is given in Figure 20. This algorithm is executed if the new key of the node v is smaller than its parent's key.

3 Amortized analysis

In an *amortized analysis*, time required to perform a sequence of data structure operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees that *average performance of each operation in the worst case*.

There are several techniques used to perform an amortized analysis, the method of amortized analysis used to analyze Fibonacci heaps is the potential method. When using this method we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method works as follows. We start with an initial data structure D_0 on which s operations are performed. For each $i = 1, \ldots, s$, we let c_i be the actual cost of the i-th operation and D_i be

the data structure that results after applying the i-th operation to the data structure D_{i-1}. A potential function Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the *potential* (energy) associated with the data structure D_i. The *amortized cost* $\hat{c}_i$ of the i-th operation with respect to the potential function Φ is defined by:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}). \tag{2}$$

The amortized cost of each operation is thus its actual cost plus the increase in potential due to the operation. The total amortized costs of the s operations is

$$\sum_i \hat{c}_i = \sum_i c_i + \Phi_{D_s} - \Phi_{D_0}, \tag{3}$$

If we can prove that $\Phi_{D_s} \geq \Phi_{D_0}$, then we have shown that the amortized costs bound the real costs. Thus, we can analyze the amortized costs to obtain a bound on the actual costs. In practice, we do not know how many operations s might be performed. Therefore, if we require that $\Phi(D_i) \geq \Phi_{D_0}$ for all i, then we guarantee that we pay in advance. It is often convenient to define $\Phi(D_0) = 0$ and then to show that $\Phi(D_i) \geq 0$.

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the i-th operation is positive, then the amortized cost $\hat{c}_i$ represents an overcharge to the i-th operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized costs represents an undercharge and the actual cost of the operation is paid by a decrease in the potential.

Exercises 3.1. *On amortized analysis:*

1. *Let S be a string composed of n bits (S represents a number between 0 and $2^n - 1$). Suppose we store S using a circular list with a pointer to the least significant bit. Define the operations $inc(S) = S + 1 \mod 2^n$ and $div2(S) = \lfloor S/2 \rfloor$. How can we implement $inc(S)$ and $div2(S)$ such that their amortized cost is $O(1)$. Use a potential function (you may assume that initially $S = 0$).* ☆

2. *Consider an array T of size $T.size$ and let $T.num$ be the number of elements present in T. We consider a single operation that adds one element to T. When T is full (i.e., $T.num = T.size$)and we need to add another element, we allocate a new array T_{new} of size $2T.size$ and copy the elements from T to T_{new}, remove T and rename T_{new} as T. Define a potential function such that the amortized cost of this operation is $O(1)$. [Hint: Let $\Phi(T)$ depend on $T.num$ and $T.size$. Make sure $\Phi(T)$ cannot be negative.]* ☆

3. *How can you implement an insert operation in a binary heap with n nodes in $O(\log n)$ time? Define a potential function H such that the amortized cost of the Insert operation equals $O(\log n)$ and that of a Delete-Min only $O(1)$.* ☆

3.1 Amortized analysis of the *delete-min* and *decrease-key* operation

Define $\Phi(H)$ as the number of root nodes $t(H)$ plus two times the number of marked nodes $m(H)$ in the Fibonacci heap H, i.e., $\Phi(H) = t(H) + 2m(H)$. We assume that a single unit of potential can pay for a constant amount of work, where the constant is sufficiently large to cover the cost of any of the specific constant-time pieces of work that we might encounter. Assume that a Fibonacci heap application begins with no heaps (this is the case for both the SP and MST algorithm). The initial potential, therefore, is 0, and obviously the potential is nonnegative at all subsequent times. Hence, the total amortized cost is thus an upper bound on the total actual cost for the sequence of operations (see Eq. (3)). We further assume that there is some upper bound $D(n)$ on the maximum degree of any node in an n-node Fibonacci heap. We derive this upper bound in Section 4.

The actual cost of a delete-min operation can be accounted for as follows. An $O(1)$ contribution comes from splicing the (at most $D(n)$) children of the minimum node in the root list (because setting the parent pointers to Null and unmarking the nodes is done by the Cleanup algorithm). The size of the root list upon calling the Cleanup algorithm is $D(n) + t(H) - 1$, since it consist of the original $t(H)$ root list nodes, minus the minimum node, plus the children of the extracted node. Meaning, at most $D(n) + t(H) - 1$ link operations are performed. Thus, the total amount of work performed is at most proportional to $D(n)+t(H)$, i.e., $O(D(n)+t(H))$. The potential before extracting the minimum node is $t(H) + 2m(H)$, and the potential afterward is at most $(D(n) + 1) + 2m(H)$, since at most $D(n) + 1$ roots remain (that is, there is at most 1 node with $0, 1, \ldots, D(n)$ children after the Cleanup) and no nodes become marked during the operation. The amortized cost is thus at most

$$\begin{aligned} & O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ &= O(D(n)) + O(t(H)) - t(H) \\ &= O(D(n)), \end{aligned}$$

since we can scale up the units of the potential to dominate the hidden constant in $O(t(H))$. Intuitively, the cost of performing the link operations

is paid by the decrease in the potential due to reducing the number of nodes on the root list.

Let us now consider the *decrease-key* operation. Decreasing the key has an actual cost of $O(1)$. Suppose that c recursive invocations of the PROMOTE function are called. Each recursive call takes O(1) time except for the recursive calls, hence, the actual cost of *decrease-key* is O(c). Next, we compute the change in potential. Each recursive call, except for the last, cuts a marked node and unmarks the node. Afterward, there are $t(H) + c$ trees (the original $t(H)$, $c - 1$ trees produced by the recursive calls, and the tree rooted at the node whose key was decreased). Whereas the maximum number of marked nodes equals $m(H) - c + 2$ ($c - 1$ were unmarked and the last call may have marked a node). The change in potential is therefore at most

$$((t(H) + c) + 2(m(H) - c + 2) - (t(H) + 2m(H)) = 4 - c. \tag{4}$$

Thus, the amortized cost of the *decrease-key* is at most

$$O(c) + 4 - c = O(1), \tag{5}$$

by scaling up the units of the potential to dominate the hidden constant in $O(c)$.

4 Bounding the maximum degree

In order to prove that the amortized time of the *delete-min* operation is $O(\log(n))$, we must show that $D(n)$ is bounded by $O(\log n)$. In this section we shall show that $D(n) \leq \lfloor \log_\phi n \rfloor$, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio.

LEMMA 4.1. *Let x be any node in a Fibonacci heap, and suppose that $d(x) = k$, where $d(x)$ denotes the degree of x. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, $d(y_1) \geq 0$ and $d(y_i) \geq i - 2$, for $i = 2, 3, \ldots, k$.*

Proof. Obviously, $d(y_1) \geq 0$. For $i \geq 2$, we note that when y_i was linked to x, all of $y_1, y_2, \ldots, y_{i-1}$ were children of x, so we must have had $d(y_i) \geq i-1$, as y_i was only linked to x if $d(x) = d(y_i)$. Since then, node y_i has lost at most one child, otherwise y_i would have been cut. We may conclude $d(y_i) \geq i - 2$. □

Let us now define the Fibonacci numbers F_k, for $k \geq 0$ as follows: $F_0 = 0$, $F_1 = 1$ and $F_k = F_{k-1} + F_{k-2}$, for $k \geq 2$. Then, we can easily show by

induction on k that $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$, for all $k \geq 0$ (simply apply induction on the term F_{k+1}). Moreover, $F_{k+2} \geq \phi^k$ (apply induction on both terms and use the fact that $(1 + \phi) = \phi^2$).

Theorem 4.1. *Let x be any node in a Fibonacci heap, and let $k = d(x)$. Then, $size(x) \geq F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5})/2$.*

Proof. Let s_k denote the minimum possible value of $size(z)$ over all nodes z such that $d(z) = k$. That is, s_k denotes the minimum number of nodes in a tree that is rooted by a degree k root node. Let $y_1, y_2, \ldots, y_k$ denote the children of x as in Lemma 4.1 in the order they were linked to x. To compute a lower bound on $size(x)$, we count one for x itself, one for the first child y_1 and then apply Lemma 4.1 for the remaining children. We have

$$size(x) \geq s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}. \tag{6}$$

We now show by induction that $s_k \geq F_{k+2}$, for all $k \geq 0$. The cases for $k = 0$ and 1 are trivial. By induction, we have $s_i \geq F_{i+2}$ for $i = 0, \ldots, k-1$, therefore,

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 1 + F_1 + \sum_{i=2}^{k} F_i = F_{k+2}. \end{aligned}$$

Thus, we have shown $size(x) \geq s_k \geq F_{k+2} \geq \phi^k$. □

Corollary 4.1. *The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is bounded by $\lfloor \log_\phi n \rfloor$, meaning it is $O(\log n)$.*

Proof. Let x be any node of an n-node Fibonacci heap and let k be its degree. By the previous theorem we have $n \geq size(x) \geq \phi^k$. Taking the base-$\phi$ logarithms yields $k \leq \log_\phi n$. Therefore, the maximum degree is $O(\log n)$. □

As a results of this corollary, Prim's MST algorithm (or Dijkstra's SP) has an amortized cost of $O(n \log n + e)$, as a *decrease-key* operation is $O(1)$ and a *delete-min* operations is $O(\log n)$.

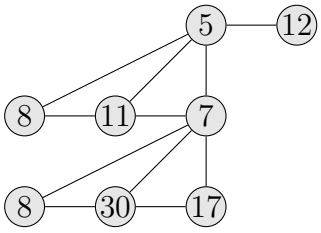
Exercises 4.1. *On Fibonacci heaps:*

1. *Suppose we also wish to support a* Union *and* Delete *operation for Fibonacci heaps. How would you implement these operations? What is the (amortized) cost of your solution?* ☆

2. *What is the maximum number of vertices in the root-list of a Fibonacci heap consisting of* 40 *vertices given that we just executed a* Delete-Min *operation?* ☆

3. *Redo the amortized analysis of the* Delete-Min *and* Decrease-Key *operations if we use* $t(H) + m(H)$ *as a potential function.* ☆

4. *Give a sequence of operations (starting from an empty heap) to obtain the following Fibonacci heap or explain why this is impossible.* ☆



5. *Let* $n(H)$ *be the number of nodes in a Fibonacci heap* H*. Suppose we use the potential function:* $\hat{\Phi}(H) = log(n(H)!) + t(H) + 2m(H)$*. Argue that both the* Delete-Min *and* Decrease-Key *operations have an* $O(1)$ *amortized cost. Why is the overall amortized cost when using* $\hat{\Phi}$ *still* $O(|E| + |V| \log |V|)$ *when executing Prim's algorithm?* ☆

6. *What is the maximum depth of a node in a Fibonacci heap consisting of* n *nodes? Prove your answer and provide a sequence of operations that creates such a node.*

7. *Suppose we allow a node to lose* $s > 1$ *children before promoting it to the root list. Would this alter the overall runtime complexity? Redo the parts of the (amortized) analysis that require changes to support your answer.*

SHORTEST PATH PROBLEMS

1 Problem setting

Consider a directed weighted graph $G = (V, E)$ with weight function $\alpha : E \to \mathbb{R}$. Note that in general we allow edges to have negative weights, though in some cases we restrict ourselves to graphs with positive weights only. We define the weight $\alpha(p)$ of a path p from some vertex v_0 to v_k as the sum of the weights of the edges on the path p. In other words, if p is the path composed of the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then

$$\alpha(p) = \sum_{i=0}^{k-1} \alpha(v_i, v_{i+1}).$$

The shortest path length $\delta(u, v)$ between two vertices u and v is defined as the minimum of the weights of all the paths from u to v in G, that is,

$$\delta(u, v) = \min_{p \in \mathcal{P}_{u,v}} \alpha(p),$$

where $\mathcal{P}_{u,v}$ is the set of all the paths from u to v in G. If there is no path from u to v we set $\delta(u, v) = \infty$. Note that there may be several paths between u and v with the shortest path length.

A shortest path tree rooted in s is defined as a tree T_s with root s that contains the vertices v reachable from s and is such that the unique path from s to v in T_s is a shortest path.

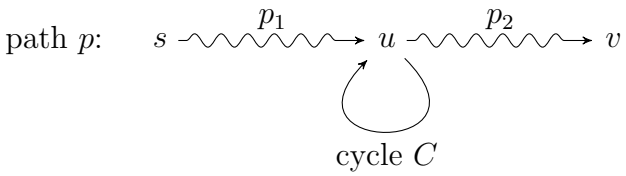
In this chapter we mainly focus on the **single source shortest path (SSSP)** problem: given a source vertex s find

1. the shortest path lengths $\delta(s, v)$ for any $v \in V$ and
2. a shortest path tree T_s with root s.

In practice we may only be interested in finding the shortest path from s to a specific vertex t. However, it turns out that solving the problem of finding the shortest path from s to some t is in general just as complex as solving the SSSP problem. When the weights of all the edges are the same, the

problem becomes less complex and we can use the BFS algorithm that runs in $O(|V| + |E|)$ time.

We start with a number of simple observations. Let C be a cycle in $G = (V, E)$ and define the weight $\alpha(C)$ of the cycle C as the sum of the weights of its edges, hence $\alpha(C) = \sum_{e \in C} \alpha(e)$. If we allow negative weights the weight $\alpha(C)$ of some cycle C could be negative, in which case we refer to C as a negative weight cycle. Assume now that there is some vertex u that is part of a negative weight cycle C and that lies on a path from s to v. Now consider a path p from s to v



that is composed of a path p_1 from s to u, followed by $k \geq 0$ cycles C and finally by a path p_2 from u to v. Then

$$\alpha(p) = \alpha(p_1) + k\alpha(C) + \alpha(p_2)$$

can be made arbitrarily small by picking k large enough. In other words the SSSP problem with source s is only well defined if there are no negative weight cycles in G that can be reached from s. If all the weights are positive, we clearly never have negative weights cycles. The following theorem is easy to prove.

Theorem 1.1 (Shortest Path Properties)**.** *Let p be a shortest path from s to v in G.*

1. *If p is composed of the edges $(s, v_0), (v_0, v_1), \ldots, (v_{k-1}, v_k), (v_k, v)$ then the path p_{ij} composed of the edges $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is a shortest path from v_i to v_j.*
2. *If u is a vertex on p, then $\delta(s, v) = \delta(s, u) + \delta(u, v)$.*
3. *If (u, v) is the last edge on the path p, then $\delta(s, v) = \delta(s, u) + \alpha(u, v)$.*
4. *For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + \alpha(u, v)$.*

Proof. If there was a shorter path p'_{ij} from v_i to v_j, we can construct a path from s to v that is shorter than p by replacing p_{ij} by p'_{ij}. The second statement follows by applying the first statement twice (with $v_i = s$ and $v_j = u$ and with $v_i = u$ and $v_j = v$). The third statement is a special case of the second. The final statement follows by noting that the shortest path from s to u followed by the edge (u, v) is a path from s to v, but it is not necessarily the shortest path from s to v. □

INITIALIZE-SINGLE-SOURCE:

for $v \in V$
  do $d(v) = \infty, \pi(v) = \text{NIL}$
$d(s) = 0$.

RELAX(u, v):

If $d(v) > d(u) + \alpha(u, v)$
  $d(v) = d(u) + \alpha(u, v)$, $\pi(v) = u$

Fig. 21: The INITIALIZE-SINGLE-SOURCE and RELAX operation

2 Relaxation

Instead of simply presenting several algorithms to solve the single source shortest path problem, we first introduce the notion of **relaxation** and we prove two important properties of a sequence of relaxations. These results can then be used to prove the correctness of several shortest path algorithms.

The relaxation approach uses a single value $d(v)$ for each vertex $v \in V$ and a single pointer $\pi(v)$. These values are initialized by the function INITIALIZE-SINGLE-SOURCE as indicated in Figure 21. We can perform a RELAX operation on any edge (u, v) and this operation may decrease $d(v)$ and update $\pi(v)$ as shown in Figure 21. We associate the predecessor graph $G_\pi = (V_\pi, E_\pi)$ with the set of pointers $\{\pi(v) | v \in V\}$ by letting $V_\pi = \{v \in V | \pi(v) \neq \text{NIL}\} \cup \{s\}$ and $E_\pi = \{(\pi(v), v) | v \in V_\pi \setminus \{s\}\}$.

The main idea will be that if we perform a sequence of RELAX operations after the initialization (in any order and allowing multiple RELAX operations on the same edge), then $d(v)$ remains infinite for vertices that cannot be reached from s, $d(v)$ becomes finite at some point for vertices that are reachable from s and for these vertices $d(v)$ gradually decreases to $\delta(s, v)$. Finally, when $d(v) = \delta(s, v)$ for all $v \in V$, the pointers $\pi(v)$ yield a predecessor graph G_π that is a shortest path tree T_s rooted in s. The next theorem shows that the $d(v)$ values are indeed upper bounds and $d(v)$ remains equal to $\delta(s, v)$ once it has reached this value.

THEOREM 2.1 (Relaxation Properties)**.** *After a* RELAX(u, v) *operation we have* $d(v) \leq d(u) + \alpha(u, v)$*. Moreover during any sequence of* RELAX *operations that follow the* INITIALIZE-SINGLE-SOURCE *operation the values* $d(v)$ *do not increase and are such that* $d(v) \geq \delta(u, v)$ *at all times.*

Proof. The first statement is immediate as $d(v) = d(u) + \alpha(u, v)$ after the RELAX operation on (u, v) in case $d(v) > d(u) + \alpha(u, v)$ before the operation and $d(u)$ does not change by a RELAX(u, v) call.

To prove the second statement, we first note that $d(v)$ cannot increase. Further immediately after initialization $d(v)$ is an upper bound on $\delta(s, v)$ as

$d(s) = 0 \geq \delta(s,s)$ (either $\delta(s,s) = 0$ or $\delta(s,s) = -\infty$ if s lies on a negative weight cycle) and $d(v) = \infty \geq \delta(s,v)$ for $v \neq s$. Suppose now that at some point we relax (u,v) and this causes $d(v)$ to decrease such that $d(v) < \delta(s,v)$ for the first time. As $d(v)$ was decreased by the RELAX operation, we have

$$d(u) + \alpha(u,v) = d(v) < \delta(s,v) \leq \delta(s,u) + \alpha(u,v),$$

where the second inequality is due to statement 4 in Theorem 1.1. Thus, we have $d(u) < \delta(s,u)$, which contradicts the fact that v was the first vertex for which $d(v) < \delta(s,v)$. □

As $\delta(s,v) = \infty$ for vertices v that cannot be reached from s, we have that $d(v) = \infty$ and $\pi(v) =$NIL for such vertices after any sequence of RELAX operations.

THEOREM 2.2 (Relaxation Trees)**.** *After any sequence of* RELAX *operations that follow the initialize operation, the predecessor graph G_π forms a tree with root s provided that there are no negative weight cycles reachable from s.*

Proof. Note that $d(v) < \infty$ for any vertex v with $\pi(v) \neq$ NIL, meaning $d(v) < \infty$ for $v \in V_\pi$. We first show that G_π does not contain a cycle. Assume there is a cycle C with vertices $v_0, \ldots, v_k$ with $v_k = v_0$ and assume without loss of generality that (v_{k-1}, v_k) was the last edge on this cycle that was created by a RELAX operation. As $\pi(v_i) = v_{i-1}$, the last RELAX(u,v) operation with $v = v_i$ that decreased $d(v_i)$ was a RELAX(v_{i-1}, v_i) operation that set $\pi(v_i) = v_{i-1}$ and $d(v_i) = d(v_{i-1}) + \alpha(v_{i-1}, v_i)$. Because $d(v_{i-1})$ can only decrease afterwards we have

$$d(v_i) \geq d(v_{i-1}) + \alpha(v_{i-1}, v_i),$$

for $i = 1, \ldots, k-1$ before the RELAX operation is performed on (v_{k-1}, v_k). Since the RELAX operation on (v_{k-1}, v_k) decreased $d(v_k)$, we have just prior to this operation that

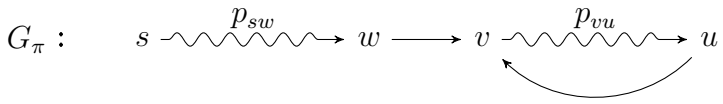
$$d(v_k) > d(v_{k-1}) + \alpha(v_{k-1}, v_k).$$

Combining the last two inequalities we have

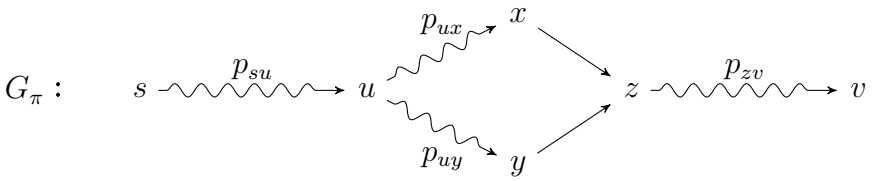
$$\sum_{i=1}^{k} d(v_i) > \sum_{i=1}^{k} (d(v_{i-1}) + \alpha(v_{i-1}, v_i)) = \sum_{i=1}^{k} d(v_i) + \alpha(C),$$

as $v_0 = v_k$. This implies that $\alpha(C) < 0$ (as $d(v_i) < \infty$), meaning C is a negative weight cycle. Due to our assumption of not having negative weight cycles reachable from s, this implies that G_π is a directed acyclic graph.

Next we prove that any vertex $v \in V_\pi$ is reachable from s in G_π. This is clearly true after the initialization and we now argue this remains true whenever we perform a RELAX operation on (u, v). There are three cases. (1) If $d(v)$ does not decrease, G_π remains identical. (2) If $d(v)$ became finite, we add v to V_π and an edge (u, v) to E_π where u was already part of V_π and reachable from s. (3) Finally, if $d(v)$ was already finite and further decreased, some edge (w, v) in G_π is replaced by (u, v). If u is still reachable from s after the RELAX(u, v) operation, v and all the other vertices in V_π remain reachable from s. u can only become unreachable due to the removal of (w, v). However, if the removal of (w, v) makes u unreachable from s in G_π, there must have been a path p_{vu} from v to u in G_π and adding (u, v) to G_π creates a cycle as shown below:



We end by showing that G_π is a tree with s as a root. As all the vertices in V_π are reachable from s, it suffices to show that there cannot be two paths from s to some vertex v in G_π. Assume we have two different paths p_1 and p_2, then there exists an $x, y \in V_\pi$ with $x \neq y$ such that



where p_1 is a path p_{su} from s to some u, followed by a path p_{ux} from u to some x, followed by an edge (x, z) and finally a path p_{zv} from z to v, while p_2 is equal to p_{su}, followed by some path p_{uy}, followed by an edge (y, z) and the path p_{zv}. This however implies that $\pi(z) = x$ and $\pi(z) = y$ at the same time, which is impossible. □

THEOREM 2.3 (Shortest Path Trees)**.** *Assume there are no negative weight cycles reachable from s in G. For any sequence of* RELAX *operations that follow the initialize operation and that ends with $d(v) = \delta(s, v)$ for any $v \in V$, the final predecessor graph G_π is a shortest paths tree T_s with root s.*

Proof. We first argue that V_π is equal to the set of the vertices that are reachable from s in G at the end of the sequence. Clearly a vertex v is reachable if and only if $\delta(s, v)$ is finite. As $d(v) = \delta(s, v)$, this means v is reachable if and only if $d(v)$ is finite. Finally, $d(v)$ is finite if and only if $\pi(v) \neq$NIL, where the latter implies that $v \in V_\pi$.

```
DIJKSTRA(G, s):

INITIALIZE-SINGLE-SOURCE
S = ∅
Q = V
while Q ≠ ∅ do
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for v ∈ Γ(u)
        RELAX(u, v)
    end for
end while
```

Fig. 22: Dijkstra's algorithm for the single source shortest path problem

From the previous theorem we know that G_π is a tree rooted at s, so it suffices to show that the unique simple path p from s to v is a path with weight $\delta(s,v)$. Let $v_0,\ldots,v_k$ be the vertices on the unique path from s to v with $v_0 = s$ and $v_k = v$. Using the same argument as in the previous proof, we know that $d(v_i) = v_{i-1}$ implies that

$$d(v_i) \geq d(v_{i-1}) + \alpha(v_{i-1}, v_i),$$

for $i = 1,\ldots,k$. This implies that $\alpha(v_{i-1},v_i) \leq d(v_i) - d(v_{i-1})$ and therefore

$$\begin{aligned}\alpha(p) &= \sum_{i=1}^{k} \alpha(v_{i-1},v_i) \leq \sum_{i=1}^{k} (d(v_i) - d(v_{i-1})) \\ &= d(v_k) - d(v_0) = d(v) - d(s) = \delta(s,v) - \delta(s,s) = \delta(s,v).\end{aligned}$$

Hence, the path p is a shortest path. □

3 Dijkstra's algorithm

In this section we present Dijkstra's algorithm to solve the single source shortest path problem. This algorithm may fail if some of the weights $\alpha(u,v)$ are negative. As such we assume in this section that all the weights are such that $\alpha(u,v) \geq 0$.

Dijkstra's algorithm starts from the vertex s and constructs a shortest path tree T_s by adding a single vertex u during each step followed by a series

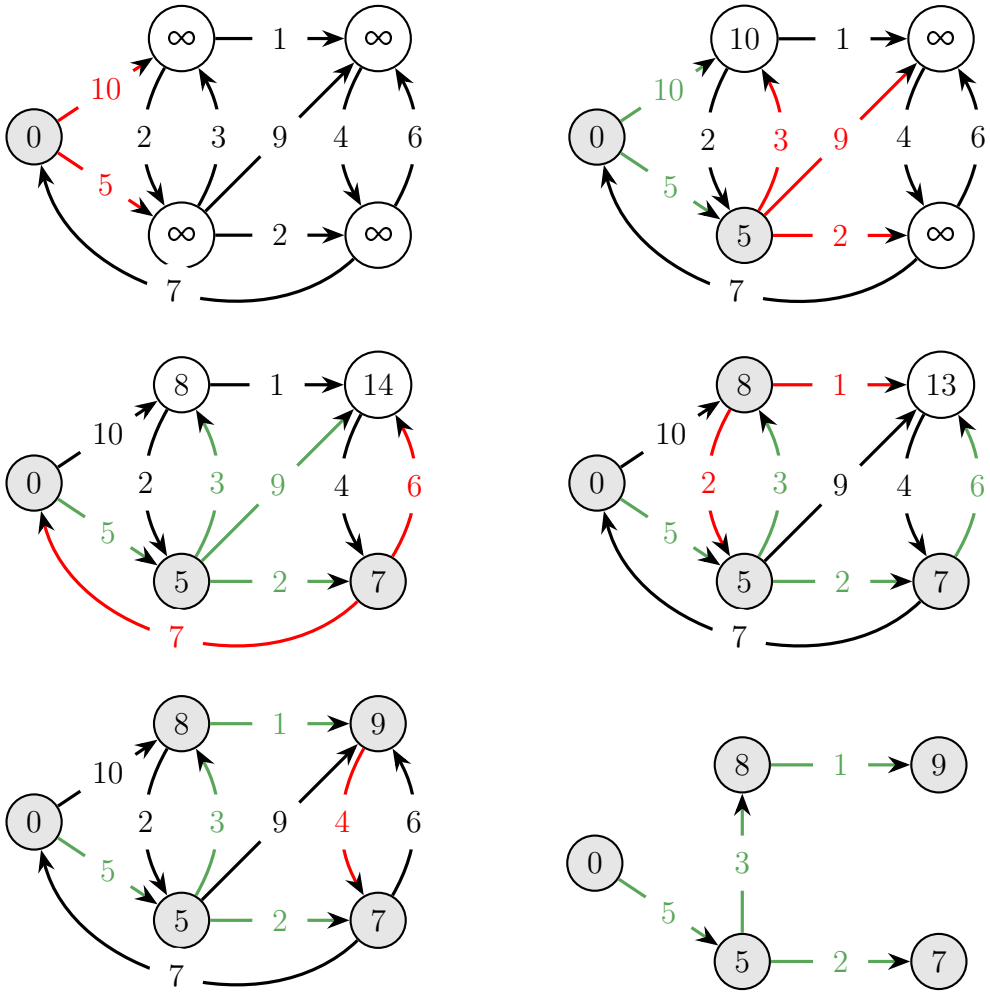


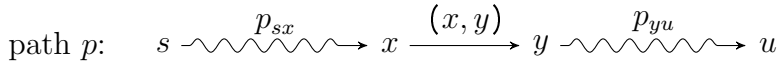
Fig. 23: Dijkstra's algorithm executed on a small example

of RELAX operations on the edges starting in u. The pseudo code is presented in Figure 3.1. Note that initially all $d(u)$ values are infinite except for $d(s)$ which equals zero. Therefore the first EXTRACT-MIN operation adds s to S.

THEOREM 3.1. *If all weights $\alpha(u,v) \geq 0$, then Dijkstra's algorithm ends with $d(v) = \delta(s,v)$ for any $v \in V$. As a result, the final predecessor graph G_π is a shortest paths tree T_s with root s due to Theorem 2.3.*

Proof. We first show that $d(u) = \delta(s,u)$ when u is added to S, which implies that this remains the case thereafter due to Theorem 2.1. Assume the statement is false and let u be the first vertex that is added to S such that $d(u) \neq \delta(s,u)$. For such a vertex u there must be a shortest path p from s

to u as otherwise $d(u) = \delta(s,u) = \infty$:



where y is the first vertex on the path p not belonging to S when u is inserted in S (note that the path p_{yu} may or may not visit S again). Let x be the vertex just before y on p. As x was added to S before u, $d(x) = \delta(s,x)$ when x was added to S and the edge (x,y) was relaxed afterwards, meaning

$$d(y) \leq \delta(s,x) + \alpha(x,y).$$

Because (x,y) lies on a shortest path p, we have

$$\delta(s,y) = \delta(s,x) + \alpha(x,y),$$

allowing us to conclude $d(y) = \delta(s,y)$ when u is inserted in S. Thus, as all weights are non-negative

$$d(y) = \delta(s,y) \leq \delta(s,u) \leq d(u).$$

As u is extracted from Q before y, we further have $d(u) \leq d(y)$, which yields $d(u) = \delta(s,u)$. This is clearly a contradiction on our assumption that $d(u) \neq \delta(s,u)$.

We end by noting that initially $Q = V$, so $S = V$ when Dijkstra's algorithm halts. □

If we simply store the $d(v)$ values in an array, we can implement the Relax operation in $O(1)$ time, while the Extract-Min operation takes $O(|V|)$ time. This implies that we have an overall time complexity of $O(|V|^2)$. Instead of using a simple array, we can also make use of a binary heap. In this case each Relax operation corresponds to a DecreaseKey operation and the Extract-Min operation is equivalent to a DeleteMin operation. As both the DecreaseKey and DeleteMin operation have a time complexity of $O(\log|V|)$, the overall complexity of Dijkstra's algorithm becomes $O(|E|\log|V|)$. Finally, we can also rely on a Fibonacci heap, which similar to Prim's algorithm results in a time complexity of $O(|E| + |V|\log|V|)$ as each DecreaseKey now has a $O(1)$ amortized cost.

Exercises 3.1. *On Dijkstra's algorithm:*

1. *Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers.*

```
BELLMAN-FORD(G, s):

INITIALIZE-SINGLE-SOURCE
for i = 1 to |V| − 1 do
    for (u, v) ∈ E do
        RELAX(u, v)
    end for
end for
for (u, v) ∈ E do
    If d(v) > d(u) + α(u, v) do
        Return FALSE;
    end if
end for
Return TRUE;
```

Fig. 24: Bellman-Ford algorithm for the single source shortest path problem

4 Bellman-Ford algorithm

We now introduce the Bellman-Ford algorithm for solving the single source shortest path problem. This algorithm allows us to use both positive and negative weights. Recall however that the shortest path problem is only well defined if there is no negative weight cycle that can be reached from the source node s. The Bellman-Ford algorithm is very simple as it simply performs $|V| - 1$ times a RELAX operations on all of the edges of the graph, as can be seen in Figure 24. This algorithm not only solves the single source shortest path problem in case there is no negative weight cycle in the graph that is reachable from s, but it also performs a check at the end to see whether such a cycle exists (and returns **FALSE** if this is the case).

THEOREM 4.1. *When G does not contain a negative weight cycles reachable from s, then $d(v) = \delta(s, v)$ for any vertex v reachable from s in G when the Bellman-Ford algorithm ends. As a result, the final predecessor graph G_π is a shortest paths tree T_s with root s due to Theorem 2.3.*

Proof. Let v be any vertex reachable from s. Recall that $\delta(s, v)$ is only well defined if G does not contain any negative weights cycles reachable from s.

Let p be a simple shortest path from s to v composed of the vertices $s = v_0, v_1, \ldots, v_k = v$ with $k \leq |V| - 1$. We argue by induction that $d(v_i) = \delta(s, v_i)$ after the i-th iteration of the Bellman-Ford algorithm. For $i = 0$ we have $d(v_0) = d(s) = 0 = \delta(s, s) = \delta(s, v_0)$. By induction we therefore have

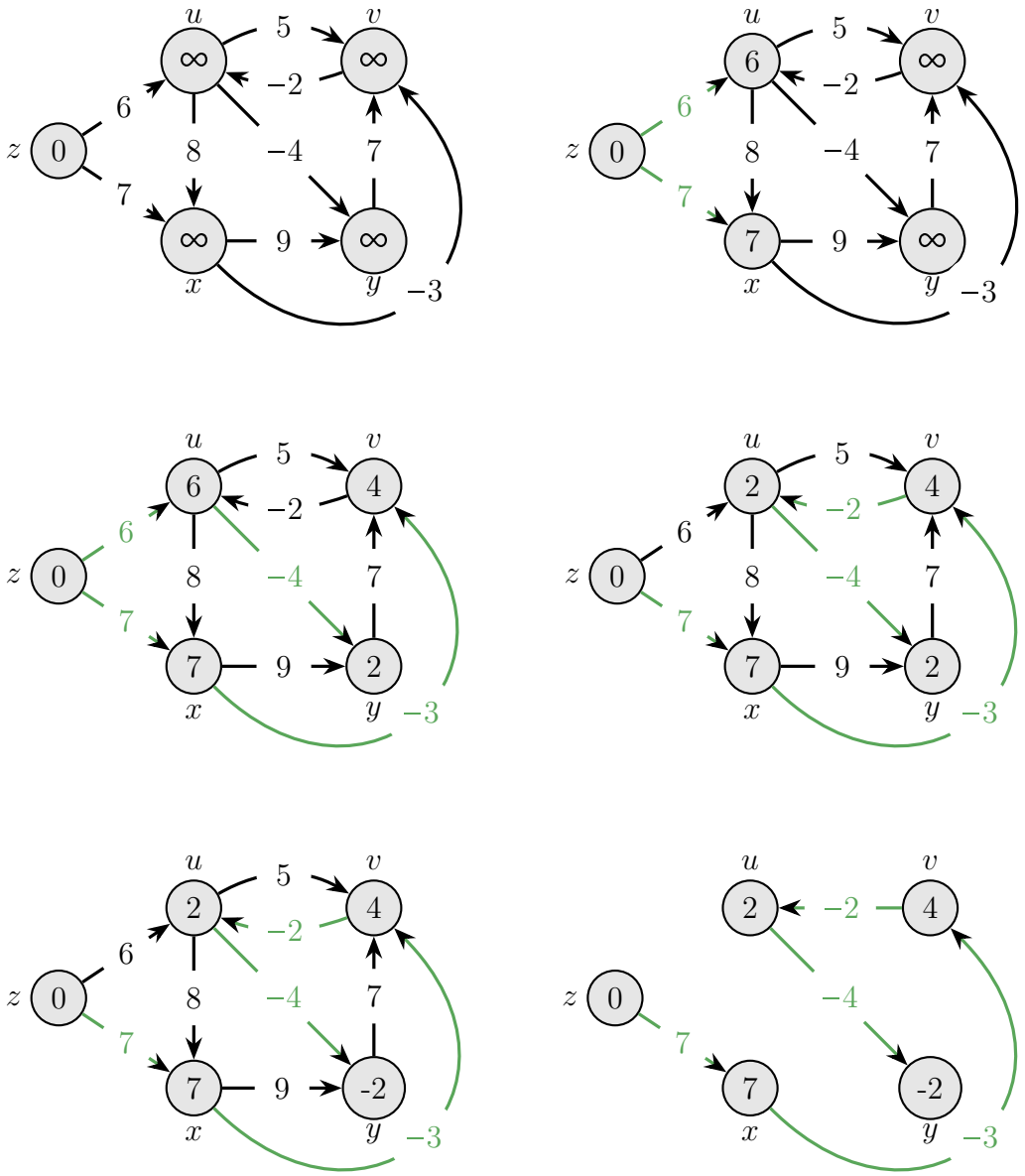


Fig. 25: Bellman-Ford algorithm executed on a small example. The RELAX operations are executed in lexicographical order during each of the 4 steps: (u, v), (u, x), (u, y), (v, u), (x, v), (x, y), (y, v), (z, u) and (z, x).

$d(v_{i-1}) = \delta(s, v_{i-1})$ after $i - 1$ iterations. During the i-th iteration all edges

are relaxed, including edge (v_{i-1}, v_i), meaning

$$d(v_i) \leq d(v_{i-1}) + \alpha(v_{i-1}, v_i) = \delta(s, v_{i-1}) + \alpha(v_{i-1}, v_i).$$

As (v_{i-1}, v_i) lies on the shortest path p, $\delta(s, v_{i-1}) + \alpha(v_{i-1}, v_i) = \delta(s, v_i)$ which completes the proof. □

In the same manner we can also show that $d(v)$ is finite at the end of the Bellman-Ford algorithm if and only if v is reachable from s (even in the presence of a negative weight cycle that can be reached from s).

Theorem 4.2. *The graph G contains a negative weight cycle reachable from s if and only if the Bellman-Ford algorithm returns* **FALSE**.

Proof. If G does not contain a negative weight cycle reachable from s, we know that $d(v) = \delta(s, v)$ due to Theorem 4.1. Therefore for any edge (u, v) we have

$$d(v) = \delta(s, v) \leq \delta(s, u) + \alpha(u, v) = d(u) + \alpha(u, v),$$

indicating that the algorithm returns **TRUE**.

We now consider the case where G contains a negative weight cycle C reachable from s composed of the vertices $v_0, v_1, \ldots, v_k$, with $v_k = v_0$. Assume the algorithm returns **TRUE**, meaning $d(v_{i-1}) \leq d(v_i) + \alpha(v_{i-1}, v_i)$ for $i = 1, \ldots, k$. Then, by summing these inequalities we have

$$\sum_{i=1}^{k} d(v_{i-1}) \leq \sum_{i=1}^{k} d(v_i) + \sum_{i=1}^{k} \alpha(v_{i-1}, v_i).$$

Because $v_0 = v_k$ and the cycle C is reachable from s, the values $d(v_i)$ are finite and we find $0 \leq \sum_{i=1}^{k} \alpha(v_{i-1}, v_i)$. This however means C is not a negative weight cycle and we obtain a contradiction on our assumption that the algorithm returns **TRUE**. □

The Bellman-Ford algorithm can be implemented using a simple array to store the $d(v)$ values as we do not need to perform any DeleteMin operations. This means the Relax operation can be executed in $O(1)$ time and the overall time complexity of the Bellman-Ford algorithm is $O(|V||E|)$.

Exercises 4.1. *On Bellman-Ford algorithm:*

1. *Give an example of a graph $G = (V, E)$ with $|V| = 10$, where $d(v) > \delta(s, v)$ after $|V| - 2$ iterations when running the Bellman-Ford algorithm.*

```
DAG-SHORTEST-PATH(G, s):

INITIALIZE-SINGLE-SOURCE
TOPOLOGICALSORT(G)
for u ∈ V in topologically sorted order do
    for v ∈ Γ(u) do
        RELAX(u, v)
    end for
end for
```

Fig. 26: Algorithm for the single source shortest path problem in a DAG

5 Shortest paths in a DAG

In this secion we consider directed acyclic graphs (DAGs) only, which means there are no (negative weight) cycles present in the graph. In this case the algorithm in Figure 26 can be used to solve the single source shortest path problem in $O(|V| + |E|)$ time (as the topological sort can be performed in $O(|V| + |E|)$ time and each relax operation requires $O(1)$ time).

THEOREM 5.1. *When G is a directed acyclic graph, then $d(v) = \delta(s, v)$ for any vertex v reachable from s when the* DAG-SHORTEST-PATH *algorithm ends. As a result, the final predecessor graph G_π is a shortest paths tree T_s with root s due to Theorem 2.3.*

Proof. For v unreachable from s we have $d(v) = \infty = \delta(s, v)$ (due to Theorem 2.1). When v is reachable from s there is a shortest path p consisting of the vertices $s = v_0, v_1, \ldots, v_k = v$. Due to the topological order the edges of p are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ and the claim easily follows by induction by showing that $d(v_i) = \delta(s, v_i)$ after edge (v_{i-1}, v_i) is relaxed for $i = 1, \ldots, k$. □

6 Shortest simple paths

In the presence of negative weight cycles shortest paths are still well defined if we demand that the shortest paths must be simple, that is, a path is not allowed to visit the same vertex twice. We may consider the problem of finding the shortest *simple* path in case that G contains negative weight cycles. This problem is NP-hard because finding the longest simple path in a weighted graph G with non-negative weights can be reduced to this problem by simply switching the signs of the weights. The NP-hardness of the longest

simple path problem follows from the NP-completeness of the Hamiltonian path problem (which was one of the 21 NP-complete problems in Karp's famous 1972 paper). Indeed if we can determine the longest simple path in polynomial time, we can use this algorithm to see whether a Hamiltonian path exists in G in P time. A Hamiltonian path is a simple path that visits all the vertices in a graph.

When a graph does not contain any cycles, then the longest path problem can be solved in linear time by switching the signs of the weights and running the DAG-SHORTEST-PATHS algorithm. Finding the longest path in a DAG is important as it corresponds to determining so-called critical paths in planning/scheduling problems.

7 Systems of difference constraints

In this section we demonstrate how the single source shortest path problem can be leveraged to solve systems of difference constraints. A system of difference constraints consists of n variables $x_1, \ldots, x_n$ and a set of m equations of the form

$$x_j - x_i \leq b_k,$$

where $b_k \in \mathbb{R}$. For instance the following set of $m = 8$ difference constraints in 5 unknowns is an example of a system of difference constraints:

$$\begin{aligned} x_1 - x_2 &\leq 0, & x_1 - x_5 &\leq -1, \\ x_2 - x_5 &\leq 1, & x_3 - x_1 &\leq 5, \\ x_4 - x_1 &\leq 4, & x_4 - x_3 &\leq -1, \\ x_5 - x_3 &\leq -3, & x_5 - x_4 &\leq -3, \end{aligned} \tag{7}$$

which can also be written in matrix form as $Ax \leq b$. We would like to known whether a given system of difference constrains has at least one solution and to find such a solution if it exists. We will demonstrate that we can solve this problem by running the Bellman-Ford algorithm on a particular single source shortest path problem.

Define the constraint graph $G = (V, E)$ based on a system of difference constraints with m equations and n unknowns by using a single vertex v_i for each variable x_i and a single edge (v_i, v_j) with weight b_k for each constraint of the form $x_j - x_i \leq b_k$. Note that this creates at most one edge between any two vertices. We also add an extra vertex v_0 to V and add the n edges

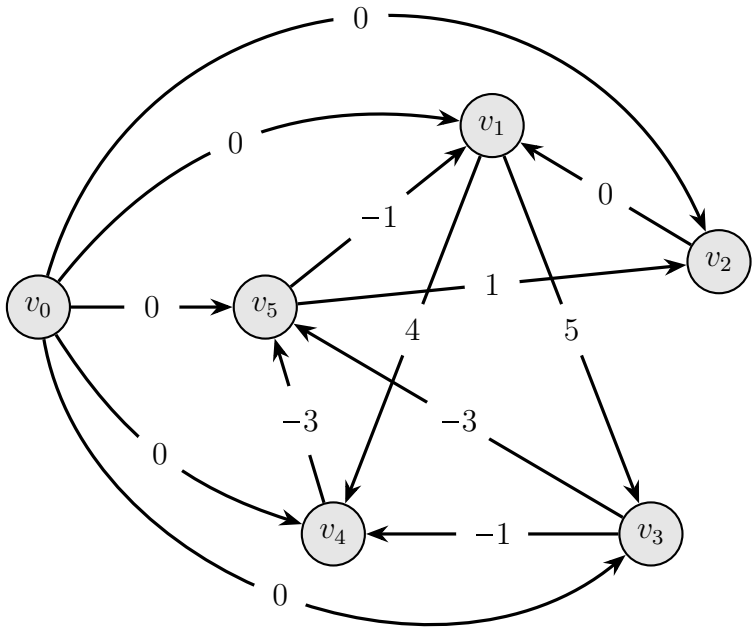


Fig. 27: Constraint graph for the system of difference constraints in (7).

(v_0, v_i) with weight zero. Hence,

$$V = \{v_0, v_1, \ldots, v_n\},$$
$$E = \{(v_0, v_i) | i = 1, \ldots, n\} \cup \{(v_i, v_j) | x_j - x_i \leq b_k\},$$

with $\alpha(v_0, v_i) = 0$ and $\alpha(i, j) = b_k$. The graph associated with the set of constraints in (7) is given in Figure 27. Note that G is constructed such that all the vertices are reachable from v_0, which will be the source vertex s for the SSSP problem. It is also clear that $\delta(v_0, v_i) \leq 0$ and equal to zero if all the weights are non-negative. Indeed if all the b_k values are non-negative, then the system of difference constraints has the trivial solution $(x_1, \ldots, x_n) = (0, \ldots, 0)$.

Theorem 7.1. *Given a system of difference constraints, if its associated constraint graph G does not contain a negative weight cycle, then setting $x_i = \delta(v_0, v_i)$ is a solution to the system of difference constraints.*

Proof. If G does not contain a negative weight cycle, the Bellman-Ford algorithm returns **TRUE** and $d(v) = \delta(s, v)$ for all $v \in V$. Therefore for any edge (v_i, v_j), we have

$$\delta(v_0, v_j) = d(v_j) \leq d(v_i) + \alpha(v_i, v_j) = \delta(v_0, v_i) + b_k.$$

Hence, $\delta(v_0, v_j) - \delta(v_0, v_i) \leq b_k$ for any edge (i, j), which means setting $x_i = \delta(v_0, v_i)$ solves the system of difference constraints. □

Note that if $(x_1, \ldots, x_n)$ is a solution to a set of difference constraints, then so is $(x_1 + d, \ldots, x_n + d)$ for any $d \in \mathbb{R}$. Although the solution is not unique, one can show that the Bellman-Ford algorithm returns the solution that maximizes $\sum_{j=1}^{n} x_j$ given that we demand that $x_i \leq 0$ for all i (see exercises).

Theorem 7.2. *A system of difference constraints has no solution if its associated constraint graph G contains a negative weight cycle.*

Proof. We show that if a solution $(x_1, \ldots, x_n)$ exists, there cannot be a negative weight cycle C in the constraint graph. Assume without loss of generality that we have such a cycle C composed of the vertices $v_1, \ldots, v_{k+1}$ with $v_{k+1} = v_1$ (v_0 cannot be part of a cycle as there are no edges that enter v_0). The k edges $(v_i, v_{i+1}) \in E$, correspond to the difference constraints $x_{i+1} - x_i \leq \alpha(v_i, v_{i+1})$, for $i = 1, \ldots, k-1$ and the constraint $x_1 - x_k \leq \alpha(v_k, v_{k+1})$. Using these k constraints implies

$$0 = x_1 + \sum_{i=1}^{k-1} x_{i+1} - x_k - \sum_{i=1}^{k-1} x_i \leq \sum_{i=1}^{k} \alpha(v_i, v_{i+1}) = \alpha(C),$$

which is a contradiction on the fact that $\alpha(C) < 0$. □

In other words, by constructing the constraints graph and running the Bellman-Ford algorithm, we can determine whether a solution to the system of difference constraints exists. Moreover, if a solution exists, the Bellman-Ford algorithm returns the solution $x_i = \delta(v_0, v_i)$. As the constraint graph contains $n + 1$ vertices and $n + m$ edges, the time complexity of the Bellman-Ford algorithm to solve a system of difference constraints equals $O((n + 1)(n + m)) = O(n^2 + nm)$.

Exercises 7.1. *On systems of difference constraints:*

1. *Argue that the solution determined by the Bellman-Ford algorithm for a system of difference constraints is such that $x_i \leq 0$ for all i.*

2. *Can we also use the Bellman-Ford algorithm if some constraints are of the form* $x_j = x_i + b_k$*?*

3. *Show that the solution determined by the Bellman-Ford algorithm maximizes* $\sum_{i=1}^{n} x_i$ *if we add the constraints* $x_i \leq 0$*. [Hint: show that for any solution* $(y_1, \ldots, y_n)$ *with* $y_i \leq 0$ *for all* i*, we have* $y_i \leq x_i = \delta(v_0, v_i)$*.]*

4. *How can we solve a system of difference constraints if we demand that* x_i *must be an integer for all* i*?*

8 All pairs shortest paths

In this section we consider the problem of finding shortest paths between any pair (u, v) of vertices, known as the **all pairs shortest path (APSP)** problem. An easy way to solve this problem is by solving $|V|$ SSSP problems. If the weights of the edges are all non-negative, we can simply run Dijkstra's algorithm $|V|$ times which results in a time complexity of $O(|V||E| + |V|^2 \log |V|)$ when using Fibonacci heaps. If some of the edge weights are negative and there are no negative weight cycles, running the Bellman-Ford algorithm $|V|$ times yields a time complexity of $O(|V|^2|E|)$.

8.1 Floyd-Warshall

In the presence of edges with negative weights, we can do better than running the Bellman-Ford algorithm $|V|$ times using a dynamic programming (DP) approach. For the APSP problem this corresponds to the Floyd-Warshall algorithm (see Figure 28). Let $V = \{1, \ldots, |V|\}$, this algorithm makes use of the variables $\delta(i, j, k)$ which represents the length of a simple shortest path p between vertices i and j, where all the vertices on the path p belong to the set $\{1, \ldots, k\} \cup \{i, j\}$. Clearly, if $k = 0$ the path p consists of vertices i and j only, meaning

$$\delta(i, j, 0) = \alpha(i, j),$$

where we define $\alpha(i, j) = \infty$ if there is no edge between i and j. Next we determine all the $\delta(i, j, 1)$ values, followed by the $\delta(i, j, 2)$ values and so on. Repeating this process $|V|$ times we find $\delta(i, j, |V|)$, which equals $\delta(i, j)$ as p can make use of all the vertices in the graph.

Assume we have computed $\delta(i, j, k-1)$ for all $i, j \in V$ and we wish to determine $\delta(i, j, k)$. There are two types of paths between i and j that make use of the set $\{1, \ldots, k\} \cup \{i, j\}$: paths that do not use vertex k and paths that use vertex k. If we take the minimum over these two types, we are done.

```
FLOYD-WARSHALL(G):
for i, j ∈ V do
    δ(i, j, 0) = ∞
end for
for (i, j) ∈ E
    δ(i, j, 0) = α(i, j)
end for
for k ∈ {1, ..., |V|}
    for i, j ∈ V do
        δ(i, j, k) = min(δ(i, j, k − 1), δ(i, k, k − 1) + δ(k, j, k − 1))
    end for
end for
```

Fig. 28: Floyd-Warshall algorithm for the APSP problem

The length of a shortest path of the first type clearly equals $\delta(i, j, k-1)$, while for the second type the length of a shortest path can be expressed as $\delta(i, k, k-1) + \delta(k, j, k-1)$. Hence,

$$\delta(i, j, k) = \min(\delta(i, j, k-1), \delta(i, k, k-1) + \delta(k, j, k-1)).$$

This algorithm therefore solves the APSP problem in $O(|V|^3)$ time.

The Floyd-Warshall algorithm works if the shortest path lengths are well defined, meaning in the absence of negative weight cycles. In the presence of negative weight cycles it does not produce the shortest simple paths as the path corresponding to $\delta(i, k, k-1) + \delta(k, j, k-1)$ may contain a negative weight cycle. Without negative weights cycles we are guaranteed to have a simple path with length $\delta(i, k, k-1) + \delta(k, j, k-1)$. To determine whether a negative weight cycle exists, we can add a vertex v_0 to V that is connected to all the vertices in V with an edge of weight zero. If we then run the Bellman-Ford algorithm with v_0 as the source, it returns **FALSE** if there is a negative weight cycle. Running Bellman-Ford once takes $O(|V||E|)$ time which is upper bounded by the $O(|V|^3)$ complexity of the Floyd-Warshall algorithm itself.

8.2 Johnson's algorithm

The $O|V|^3$ time complexity of the Floyd-Warshall algorithm is an improvement over the $O(|V|^2|E|)$ time complexity of running the Bellman-Ford algorithm $|V|$ times, but is not as good as the $O(|V||E| + |V|^2 \log |V|)$

complexity of running Dijkstra's algorithm $|V|$ times when there are no negative weight cycles. There is however a very elegant trick, known as Johnson's algorithm, that allows one to solve the APSP problem by running Dijkstra's algorithm $|V|$ times even when G has edges with negative weights. This trick exists in replacing the weights $\alpha(u,v)$ of G by some weights $\alpha_2(u,v) \geq 0$ without affecting the shortest paths and using Dijkstra's algorithm on G with weight function α_2.

Johnson's algorithm starts by checking whether a negative weight cycle exists by running the Bellman-Ford algorithm once after adding the vertex v_0 as explained in the previous subsection. This check returns a $d(v) = \delta(v_0, v)$ value for each vertex $v \in V$, then the algorithm adapts the weights by $d(u) - d(v)$, that is

$$\alpha_2(u,v) = \alpha(u,v) + d(u) - d(v),$$

for each $(u,v) \in E$. It proceeds by running Dijkstra's algorithm $|V|$ times which results in the shortest path lengths $\delta_2(u,v)$ between any two vertices $u, v \in V$, where the subscript 2 indicates that we used the weight function α_2 instead of α. Finally, the shortest path lengths $\delta(u,v)$ are computed as

$$\delta(u,v) = \delta_2(u,v) - d(u) + d(v).$$

The next theorem shows why Johnson's algorithm works:

Theorem 8.1. *For each edge (u,v), we have $\alpha_2(u,v) \geq 0$. If we replace the weight function α by α_2, then all shortest paths between any two vertices u and v remain the same.*

Proof. We have $d(u) = \delta(v_0, u)$ and $d(v) = \delta(v_0, v)$, therefore

$$d(v) = \delta(v_0, v) \leq \delta(v_0, u) + \alpha(u,v) = d(u) + \alpha(u,v).$$

Therefore $\alpha_2(u,v) = d(u) + \alpha(u,v) - d(v) \geq 0$.

Consider a arbitrary path p between u and v consisting of the vertices $u = u_0, u_1, \ldots, u_k = v$. Then

$$\begin{aligned}
\alpha_2(p) &= \sum_{i=1}^{k} \alpha_2(u_{i-1}, u_i) \\
&= \sum_{i=1}^{k} \alpha(u_{i-1}, u_i) + \sum_{i=1}^{k} d(u_{i-1}) - \sum_{i=1}^{k} d(u_i) \\
&= \alpha(p) + d(u) - d(v).
\end{aligned}$$

Hence the weight of any path between u and v is adapted by the same amount $d(u) - d(v)$, which implies that a shortest path between u and v remains a shortest path if we replace α by α_2. □