BACHELOR

Connectivity algorithms for undirected graphs in Sage

Oosterhof, F.M.

Award date:
2014

Link to publication
Bachelor Project

Connectivity Algorithms for Undirected Graphs in Sage

Author:
Florian Oosterhof [0766329]
F.M.OOSTERHOF@STUDENT.TUE.NL

Project Mentor:
dr. R.A. (Rudi) Pendavingh,
MF 4.104A

July 7, 2014   Eindhoven University of Technology
Abstract

This paper focuses on improving some implementations for the mathematical software package Sage. The problems addressed are connectivity problems in undirected graphs. The main result is that the newly implemented Push-Relabel algorithm for calculating a maximum flow is both a theoretical and practical improvement over the already available Ford-Fulkerson implementation. Furthermore, bugfixes have been proposed to improve the correctness of the implementations of the Minimal $s-t$ Edge-Cut algorithm and Gomory-Hu tree algorithm. Both of these implementations indirectly profit from the improvement of the Push-Relabel implementation.
Contents

1 Definitions and Terminology 4
   1.1 General Notions .................................................. 4
   1.2 Undirected Graphs ................................................ 4
   1.3 Directed Graphs ................................................... 5

2 Algorithmic Problems 6

3 Preliminary Research 8
   3.1 Outline and Complexity of the Algorithms .................. 8
   3.2 Implemented Sage Algorithms ................................. 11
   3.3 Pythons Time Complexity ...................................... 11

4 Maximum Flow and Push-Relabel 14
   4.1 Flow and Maximum Flow ......................................... 14
   4.2 The Push-Relabel Method ....................................... 15
   4.3 Correctness of the Push-Relabel .............................. 18

5 Implementation Details and Encountered Problems 22
   5.1 Implementation Details .......................................... 22
   5.2 Encountered Problems and Provided Bugfixes .............. 23

6 Optimizing the Implementation 25
   6.1 Dictionary Elimination ......................................... 25
   6.2 List Length Calculation ........................................ 25
   6.3 Residual Graph Construction and Maintenance ............. 25

7 Experimental Results 27
   7.1 Test Environment and Performed Tests ........................ 27
   7.2 Performance Results ............................................. 27

8 Conclusion 32
Connectivity in undirected graphs is a broad subject with many applications. Already in 1956, Ford and Fulkerson proved a seminal theorem, the Max-Flow Min-Cut Theorem, which is applicable to the Maximum Flow problem, a problem closely related to graph connectivity. It was used to estimate how the Russians would transport troops and equipment from Moscow to Berlin during the Cold War. Another result in graph theory is the so-called 'Small World Problem', which states that on average every person in the world is an acquaintance through 5 other people. Recent surveys on social media, such as Facebook, confirm these results. However, there are also groups of people completely disconnected from the rest of the world.

The goal of this project is to find out what the best algorithms are for several selected connectivity problems in undirected graphs and how these can be used to improve the mathematical software package Sage. When deciding on the best algorithm, the time complexity is used as criterion of comparison.

Chapter 1 is an introductory chapter on notation and terminology used throughout the paper. Chapter 2 defines the investigated algorithmic problems in terms of input and output. In chapter 3, an overview of algorithms to solve the discussed problems is given, as well as an overview of algorithms implemented in Sage that already solve these problems. Chapters 4, 5, 6 and 7 discuss the newly implemented algorithm in detail, how it was implemented, how it was optimized and how it performs in relation to the implementation that was already available, respectively.
1. Definitions and Terminology

Before discussing the investigated problems, some basic definitions and terminology will be described. Most of the notation that is used follows the notation in [1].

1.1 General Notions

Throughout this paper, \( N \) and \( \mathbb{R} \) denote the sets of non-negative integers and all real numbers respectively. \( \mathbb{R}_+ \) is \( \mathbb{R} \) restricted to non-negative numbers.

A multiset is a set in which multiplicities are considered, but still no order is available. An inclusionwise maximal subset \( U \subseteq V \) under property \( P \) is a set with property \( P \) such that there is no set \( W \supseteq U \) with property \( P \).

Let \( X \) be a set and \( f,g: X \to \mathbb{R} \). We say \( f \) is of order \( g \), notation \( f = O(g) \), if there exists a \( c \geq 0 \) such that \( f(x) \leq cg(x) + c \) for all \( x \in X \).

A problem is defined in terms of input and corresponding output, and an instance of a problem is input that is valid for the problem. An algorithm is a method to solve any instance of a problem, defined by specific steps to undertake. We refer to [1] for more detailed definitions of problem, instance and algorithm.

1.2 Undirected Graphs

An (undirected) graph is a pair \( G = (V,E) \) with \( V \) a finite set of vertices and \( E \) a multiset of sets of two vertices, called edges. \( E \) is a multiset because we do not restrict ourselves to only simple graphs, i.e. graphs that consider edges without multiplicities and that have no loops; edges of the form \( \{v,v\} \) for \( v \in V \). This is also the reason to make \( E \) a multiset of edges, as opposed to a regular set. An edge \( e = \{u,v\} \in E \) will be denoted by the shorthand \( e = uv \) and therefore \( uv = vu \).

A set \( U \subset V \) spans an edge \( e \) if \( e \subset U \), and is incident with \( e \) if \( U \cap e \neq \emptyset \). We denote \( \delta(U) = \{\{u,v\} \in E | u \in U, v \notin U \} \), the set of edges incident to \( U \), but not spanned by \( U \).

For a set \( U \subset V \) we denote \( G[U] \) the subgraph with vertex set \( U \) and all edges in \( E \) spanned by \( U \). Similarly \( G - U = G[V \setminus U] \), the graph with \( U \) and all incident edges removed. For a set \( F \subset E \) we denote \( G \setminus F \) the graph with the edges in \( F \) removed. The graph induced by an edge set \( E \) is the graph \( G = (V,E) \) with \( V = \bigcup_{e \in E} e \).

In a graph, a walk is a sequence of vertices \( P = \langle v_0,\ldots,v_k \rangle \) such that for each \( i \) with \( 0 \leq i < k \), \( \{v_i,v_{i+1}\} \in E \). \( P \) has length \( k \) and goes from \( v_0 \) to \( v_k \). If \( v_0,\ldots,v_k \) are all distinct, \( P \) is a path. An \( s-t \) path is a path where \( v_0 = s \) and \( v_k = t \). A closed walk is a walk where \( v_0 = v_k \). A cycle is a closed walk where \( v_1,\ldots,v_k \) are all distinct.
A graph $G$ is connected if for each $u, v \in V$ there exists a path starting at $u$ and ending in $v$. Conversely, $G$ is disconnected if it is not connected. A connected component of a graph $G$ is an inclusionwise maximal subset $U$ of $V$ under the property that $G[U]$ is a connected graph.

$G$ is $k$-connected if for every set $U \subset V$ with $|U| < k$, $G - U$ is connected. $G$ is $l$-edge-connected if for every set $F \subset E$ with $|F| < l$, $G \setminus F$ is connected.

A vertex-cut, is a set $U \subset V$ such that $G - U$ is disconnected. Note that this implies that $|U| \geq k$ for every $k$ for which $G$ is $k$-connected. An edge-cut, or simply cut, is a set $F \subset E$ such that $G \setminus F$ is disconnected. This implies that $|F| \geq l$ for every $l$ for which $G$ is $l$-edge-connected.

A tree is an undirected graph $T = (V, F)$ with a unique path between every two vertices.

1.3 Directed Graphs

A directed graph or digraph is a pair $D = (V, A)$, with $V$ a finite set of vertices and $A$ a multiset of ordered pairs of vertices $(u, v)$, called arcs. An arc $(u, v)$ is has tail $u$ and head $v$.

A set $U \subset V$ spans an arc $a = (u, v)$ if $\{u, v\} \subset U$, and is incident with $a$ if $U \cap \{u, v\} \neq \emptyset$. We denote $\delta^+(U) = \{(u, v)|u \in U, v \notin U\}$. Similarly, $\delta^-(U) = \{(u, v)|u \notin U, v \in U\}$. We denote $\delta(U) = \delta^+(U) \cup \delta^-(U)$.

The digraph induced by an arc set $A$ is the digraph $D = (V, A)$ with $V = \bigcup_{(u, v) \in A} \{u, v\}$.

For every digraph $D = (V, A)$, there is an underlying (undirected) graph $G = (V, E)$ on the same vertices with an edge $\{u, v\} \in E$ for every arc $(u, v) \in A$. $D$ is an orientation of $G$. A special type of orientation of a graph $G = (V, E)$ is the so-called dual orientation, a digraph $D = (V, A)$ where for every $\{u, v\} \in E$ both $(u, v) \in A$ and $(v, u) \in A$. This is the orientation with each edge replaced by two oppositely directed arcs.

A directed walk is a sequence of vertices $\langle v_0, ..., v_k \rangle$ such that for each $i$ with $1 \leq i < k$, $(v_i, v_{i+1}) \in A$. Such a walk has length $k$ and goes from $v_0$ to $v_k$. A directed path is a directed walk where $v_0, ..., v_k$ are all distinct. A closed directed walk is a directed walk where $v_0 = v_k$. A directed cycle is a closed directed walk where $v_1, ..., v_k$ are all distinct.

A digraph is strongly connected if for each $u, v \in V$ there exists a directed path from $u$ to $v$. 


2. Algorithmic Problems

The investigated problems are discussed below. Each problem has an input and corresponding output. Furthermore, some properties are discussed for each problem.

Connected Components

**Problem:** Connected Components  
**Input:** Undirected graph $G = (V, E)$  
**Output:** A partition $U_1, ..., U_k$ of the vertices $V$ such that $G[U_i]$ are the connected components of $G$.

The first problem is to calculate the connected components of $G$, i.e. the inclusionwise maximal sets $U \subset V$ such that $G[U]$ is connected. These sets are a partition of $V$.

2-Connected Components

**Problem:** 2-Connected Components  
**Input:** Undirected graph $G = (V, E)$  
**Output:** Subsets $U_1, ..., U_k$ of the vertices $V$ such that $G[U_i]$ are the 2-connected components of $G$.

A 2-connected component, or block, of $G$ is an inclusionwise maximal set $U \subset V$ such that $G[U]$ is 2-connected. These sets do not partition $V$, as the connected components do, however, the sets of edges of the 2-connected components do partition $E$. The problem is to find all 2-connected components of $G$.

Minimal $s - t$ Vertex Cut

**Problem:** Minimal $s - t$ Vertex Cut  
**Input:** Undirected graph $G = (V, E); s, t \in V, s \neq t$  
**Output:** A vertex-cut $U \subset V$ for $G$ that separates $s$ and $t$ of minimal size.

A vertex-cut $U$ separates $s$ and $t$ if $s$ and $t$ are in different connected components of $G - U$.

Minimal $s - t$ Edge Cut

**Problem:** Minimal $s - t$ Edge-Cut  
**Input:** Undirected graph $G = (V, E); s, t \in V, s \neq t$; optionally $w : E \rightarrow \mathbb{R}_+$  
**Output:** An edge-cut $F \subset E$ for $G$ that separates $s$ and $t$ of minimal size, or of minimal weight in case $w$ is given.
An edge-cut $F$ separates $s$ and $t$ if $s$ and $t$ are in different connected components of $G \setminus F$.

**Vertex-Connectivity**

**Problem:** Vertex-Connectivity  
**Input:** Undirected graph $G = (V, E)$  
**Output:** The value $\kappa$ such that $G$ is $\kappa$-connected, but not $(\kappa + 1)$-connected. Optionally a vertex-cut $U \subset V$ with $|U| = \kappa$

The value of $\kappa$ is equal to the smallest of the sizes of the minimal $s - t$ vertex cuts for all pairs of $s$ and $t$. Optionally, a vertex cut of size $\kappa$ can be determined.

**Edge-Connectivity**

**Problem:** Edge-Connectivity  
**Input:** Undirected graph $G = (V, E)$; optionally $w : E \to \mathbb{R}_+$  
**Output:** The value $\lambda$ such that $G$ is $\lambda$-edge-connected, but not $(\lambda + 1)$-edge-connected. Optionally an edge-cut $F \subset E$ with $|F| = \lambda$

The value of $\lambda$ is equal to the smallest of the sizes of the minimal $s - t$ edge cuts for all pairs of $s$ and $t$. When a weight function $w$ is given, the value is the minimal of all weights of these cuts. Optionally, a cut attaining the size or weight $\lambda$ can be determined.

**Gomory-Hu Tree**

**Problem:** Gomory-Hu Tree  
**Input:** Undirected graph $G = (V, E)$; optionally $w : E \to \mathbb{R}_+$  
**Output:** A tree $T = (V, F)$ such that for each $uv \in F$ and $U$ any of the two connected components of $T \setminus \{uv\}$, $\delta(U)$ is a minimum $u - v$ cut in $G$, possibly weighted by $w$.

A Gomory-Hu tree is a data structure efficiently storing a minimum $s - t$ cut for every pair of $s$ and $t$ in $G$. By construction a Gomory-Hu tree only saves the minimum $u - v$ cut for every pair of vertices $u$ and $v$ for which there is an edge $uv$ in $T$. However it indirectly also contains this information for every pair $s$ and $t$, which can be extracted with a minor calculation.

**Weighted Connectivity Augmentation**

Denote $\bar{E}$ as the set of all possible edges in a graph $G = (V, E)$.

**Problem:** Connectivity Augmentation  
**Input:** Undirected graph $G = (V, E); w : \bar{E} \to \mathbb{R}; k \in \mathbb{N}$  
**Output:** A set $F \subset \bar{E}$ of minimum weight under $w$ such that $G' = (V, E \cup F)$ is $k$-connected.

Connectivity augmentation is the augmentation of a graph $G = (V, E)$ with an edge set $F$ such that $(V, E \cup F)$ is $k$-edge-connected for a given $k$.

A related problem that is slightly easier is to augment the graph such that the edge-connectivity is increased by one, again with minimal weight.
3. Preliminary Research

Before implementation of any algorithms can be done, the theoretically best algorithms need to be researched and found for the given problems. This includes a study of asymptotic time bounds of the algorithms and a basic descriptions of the algorithms. Since the algorithms will be implemented for Sage, algorithms that solve any of these problems and are already implemented in Sage should be identified. The last part of the preliminary study is the efficiency of Pythons implementations of basic data structures.

3.1 Outline and Complexity of the Algorithms

For a graph $G = (V, E)$, we use $n = |V|$ and $m = |E|$ when stating time bounds.

Connected Components

An efficient algorithm for finding the connected components is by simply using breadth-first search or depth-first search. Both algorithms run in $O(n + m)$ time on a connected graph. By running the algorithm repeatedly on a graph, each time selecting an arbitrary unvisited vertex as starting vertex, the connected components can be identified. These are the vertices visited in a single run.

The above algorithm runs in $O(n + m)$ time, which is linear, so the connectedness of a graph can be determined in linear time.

2-Connected Components

Since 2-connected components are always connected, we will assume the graph to be connected. This can be tested in linear time by running the algorithm described in 3.1. Alternatively, the connected components can be determined and the 2-connected components can be determined for each component separately. Either case will assure that the input of this algorithm is a connected graph. The algorithm to determine the 2-connected components is based on depth-first search, and also described in [1].

Choose $s \in V$ arbitrarily and apply depth-first search with starting vertex $s$. Direct the resulting depth-first search tree such that for all $v \in V \setminus \{s\}$ there is a directed $s - v$ path, and call this digraph $T = (V, A_T)$. For every remaining edge $uv$ of $G$, there now is a directed $u - v$ path $P = \langle v_0, v_1, ..., v_k \rangle$ in $T$. Create the digraph $D = (V, A)$ by taking $T$ and adding the arc $(v, v_1)$ to $D$ for each of these remaining edges $uv$.

Now, for every strongly connected component $K$ of $D$, with $K \neq \{s\}$ there is a unique arc entering $K$. Let $u_K$ be its tail and define for every $K \neq \{s\}$: $K' := K \cup \{u_K\}$. Then $\{K'|K$ strong component of $D, K \neq \{s\}\}$ is the set of 2-connected components of $G$. 
This algorithm can be programmed to run in linear time $O(n + m)$. Therefore, its time bound is not affected by the connectedness check or determination of the connected components beforehand.

**Minimal $s$–$t$ Edge-cut**

First we note that a minimum $s$–$t$ cut $F$ is always of the form $\delta(U)$ for some $U \subset V$.

A minimum $s$–$t$ (edge-)cut can be found by running the $O(n^{2/3}m)$ algorithm described in [1] on the dual orientation of $G$. This algorithm is optimal for unweighted digraphs, however, a better bound can be obtained specifically for undirected graphs. In [2] an algorithm designed for undirected graphs for finding a minimum $s$–$t$ cut is found to have a time bound of $O(n^{3/2}m^{1/2})$.

In the weighted case, the problem is equal to the Maximum Flow problem, for which a bound of $O(nm \log(n^2/m))$ is optimal according to [1]. However, the more recent work [3] shows that a time bound of $O(nm)$ can be achieved. A new algorithm that runs in $O(nm + m^{(31/16)} \log^2 n)$ is suggested to be used along with the $O(nm + n^{2+\epsilon})$ for any $\epsilon > 0$ algorithm described in [4]. When a case distinction is made on the input size the $O(nm)$ bound can be achieved. Let $\epsilon > 0$, then for $m = O(n^{(16/15) - \epsilon})$ the new algorithm runs in $O(nm)$, and for $m > n^{1 + \epsilon}$ the other algorithm runs in $O(nm)$ time.

**Minimal $s$–$t$ Vertex-cut**

A minimum $s$–$t$ vertex-cut can be found in $O(\sqrt{nm})$ time by running the algorithm described in [1] on the dual orientation. This bound can even be improved to $O(\sqrt{\tau m})$ with $\tau$ the vertex-cover number of $G$, which is obviously less than or equal to $n$.

However, a slightly better bound, is one of $O(\sqrt{nm} \log_n(n^2/m))$, described in [5]. Rather than a specific algorithm, [5] describes a compression method for graphs that gives the extra factor $\log_n(n^2/m)$. The bound can be shown to be better by noting that $n^2/m \leq n^2$ when $m \geq 1$ and thus $\log_n(n^2/m) \leq 2$. This means that, depending on $m$, the algorithm is either faster or as fast as the above $O(\sqrt{nm})$ algorithm. Mainly for dense graphs, graphs where $m$ is big relatively to $n$, this method is better.

**Edge-Connectivity**

For the edge-connectivity $\lambda$ of a graph, [1] gives an $O(nm)$ algorithm, based on finding a legal ordering of the vertices. Let $d(X, v)$ denote the number of edges between $X \subset V$ and $v \notin X$.

A legal ordering is an ordering $v_1, \ldots, v_n$ of $V$ such that $d(V_{i-1}, v_i) \geq d(V_{i-1}, v_j)$ for each pair $i, j$ with $2 \leq i < j \leq n$, where $V_0 := \{v_1, \ldots, v_{i-1}\}$.

Such an ordering can be computed in $O(m)$ time. Denote $d(v_i) = d(V_{i-1}, v_i)$. First compute a legal ordering for $G_1 := G$, and put $v_n$ along with $d(v_n)$ in a list. Then shrink $v_n$ and $v_{n-1}$ to a single vertex, composing $G_2$. By repeating this a total of $n - 1$ times a list of pairs $(v_i, d(v_i))$ is created, which has a $k$ for which $d(v_k)$ the minimum of all $d(v_i)$. Let $U$ denote the vertices that have been shrunk to $v_k$, then $\delta(U)$ is a minimum cut. Besides in [1], this algorithm is elaborated on in [6] and [7].

A minimum-weight cut can be found by almost trivially expanding this algorithm, using a heap to calculate the legal ordering. When using a fibonacci heap, the time bound becomes $O(nm + n^2 \log n)$.
However, in [8] a faster algorithm for the unweighted edge-connectivity is proposed, with running time $O(m + \lambda^2 n \log(n/\lambda))$. By noting that $\lambda \leq 2m/n$ we can see that this bound is tighter than $O(nm)$.

**Vertex-Connectivity**

The vertex-connectivity $\kappa$ of a digraph can be found in time $O((\kappa + \sqrt{n}) \kappa \sqrt{nm})$, as stated in [1], by testing for $k$-connectivity in time $O((k + \sqrt{n}) \sqrt{nm})$ for powers of 2 as values of $k$. For each of these values of $k$, determine:

1. for all $i, j \in \{1, \ldots, k\}$ a minimum $i - j$ vertex-cut of size at most $k$
2. for all $i \in \{k + 1, \ldots, n\}$ a minimum $\{1, \ldots, i - 1\} - i$ vertex-cut and a minimum $i - \{1, \ldots, i - 1\}$ vertex-cut of size at most $k$

This can be done in $O(\min\{k, \sqrt{n}\} m)$ time, by trying to find $k$ $s-t$ edge-disjoint paths in the auxiliary graph $G'$. The auxiliary graph is constructed by creating for each vertex $v \neq s, t$ two vertices $v_{in}$ and $v_{out}$ with an edge between them. Then for each edge $\{u, w\}$ in $G$ there is an edge $\{u_{out}, w_{in}\}$ in $G'$.

Next, we note that the minimum vertex-cut in an undirected graph is the same as the minimum vertex-cut in the dual orientation. This means that a minimum vertex-cut can be found in the same time for an undirected graph.

However, a slightly better bound can be obtained, as stated in [9]. The algorithm described is the so-called Gap algorithm which is based on some properties of the quantity $d m - \kappa$, where $d_m$ is the minimum over all degrees in $G$. Its running time is $O((n + \min(\kappa^{5/2}, \kappa n^{3/4})) kn)$, which is as good as the above algorithm for $\kappa \leq n^{1/3}$ and better for $\kappa > n^{1/3}$.

**Gomory-Hu tree**

As already stated in [1], a Gomory-Hu tree can be calculated in $n - 1$ maximum flow computations. The method is based on recursion. First a minimum cut is computed in $G$, after which $G$ is split up in two subgraphs, $G'$ and $G''$, based on the split that the calculated cut represents. Though details are more subtle, the algorithm then recurs on both subgraphs and combines the trees with an extra edge representing the minimum cut calculated for the entire graph. If $\xi$ denotes the time needed to find a minimum-weight $s-t$ cut in $G$, then this algorithm runs in $O(n \xi)$ time.

**(Weighted) Connectivity Augmentation**

As already stated in [1], a $k$-edge-connector for $G$ can be found in strongly polynomial time. However, a time bound is not given. Two algorithms are proposed in [10] and [11], with time complexities of $O(m + k^2 n \log n)$ and $O(n^6)$ respectively. Though the first algorithm is not strongly polynomial, it can quite easily be seen that it has a better time bound than the second. In [12] an algorithm for graphs with integral weights is described which runs in $\tilde{O}(nm \min\{n, \Delta\})$ time, where $\Delta = k - \lambda$, the connectivity increase. The notation $f = \tilde{O}(g(n))$ means that there exists an $l \geq 1$ such that $f = O(g(n) \log^l g(n))$. 

10
3.2 Implemented Sage Algorithms

For each of these problems, except the Connectivity Augmentation, Sage has an implementation of an algorithm solving the problem. The implementations for calculating the Connected Components and 2-Connected Components are (almost) as described above. However, the implementations for Vertex-Connectivity, Edge-Connectivity and Minimal Vertex-cut use (Integer) Linear Programming methods to solve their respective problems.

The Minimal Edge-cut is computed by first calculating a maximum flow in the dual orientation. The maximum flow implementation uses either an (Integer) Linear Program or an implementation of the Ford-Fulkerson algorithm with Edmonds-Karp path determination. This choice is left to the user as a parameter of the call, but defaults to the Ford-Fulkerson implementation.

The implementation of the algorithm for calculating a Gomory-Hu tree follows the recursive algorithm as described above. It uses the discussed Minimal(-weight) Edge-cut implementation, also giving the user a choice for which method to use, defaulting to Ford-Fulkerson.

The algorithms that could therefore be newly implemented or improved are:
1. Maximum Flow, and indirectly Minimum Edge-cut and Gomory-Hu tree
2. Minimal Vertex-cut
3. Edge-Connectivity
4. Vertex-Connectivity
5. Connectivity Augmentation

The decision of which problem to improve has mainly been based on personal interests. Before starting this project I had never heard of the Gomory-Hu tree, and when I read about it, I was fascinated by it. Though the algorithm for calculating a Gomory-Hu tree is already implemented, it heavily relies on the computation of the edge cut and in fact most time is spent calculating such cuts. The implementation of the edge cut algorithm is basically a maximum flow computation, along with some post processing. This is why the decision was made to improve the algorithm to calculate the maximum flow in a graph.

Though the stated time bounds are nice theoretical results, they often heavily rely on a substantial amount of earlier research and results. Although still correct, it becomes quite hard to implement them without a substantial amount of research and in-depth knowledge. This was a reason to shift the work from implementing the best known algorithm stated in chapter 3.1 to implementing an easier algorithm, the Push Relabel algorithm.

3.3 Pythons Time Complexity

In order to be able to efficiently implement the investigated algorithms in Python and Sage, the asymptotic running times of some basic operations that Python has implemented need to be investigated. This is necessary to preserve the asymptotic complexity of the algorithms when implemented. The basic data structures of Python are lists, deques, sets and dictionaries. The information in Table 3.1 is taken from [13], where \( n \) is the size of the data structure, i.e. the number of elements, \( k \) is an integer parameter and \( l \) is the size of the parameter if it is another data structure. For dictionaries \( N \) is the maximum size ever attained. For some set operations no worst case bound is given in [13], in which case it is left blank here as well.

The “Average Case” assumes input generated uniformly at random. The “Amortized
Worst Case” means that in the worst case a sequence of \( m \) operations of time \( O(f(n)) \) take time \( O(mf(n)) \). Therefore, in the worst case scenario, each operation will take \( O(f(n)) \) time on average, but individual operations may take much more time.

<table>
<thead>
<tr>
<th>List Operations</th>
<th>Average Case</th>
<th>Amortized Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Append*</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Insert</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Get Item</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Set Item</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Delete Item</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Iteration</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Get Slice</td>
<td>( O(k) )</td>
<td>( O(k) )</td>
</tr>
<tr>
<td>Del Slice</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Set Slice</td>
<td>( O(l + n) )</td>
<td>( O(l + n) )</td>
</tr>
<tr>
<td>Extend*</td>
<td>( O(l) )</td>
<td>( O(l) )</td>
</tr>
<tr>
<td>Sort</td>
<td>( O(n \log n) )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Multiply</td>
<td>( O(nl) )</td>
<td>( O(nl) )</td>
</tr>
<tr>
<td>( x \in s )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Min(s), Max(s)</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Get Length</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deque Operations</th>
<th>Average Case</th>
<th>Amortized Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Append</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Appendleft</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Pop</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Popleft</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Extend</td>
<td>( O(l) )</td>
<td>( O(l) )</td>
</tr>
<tr>
<td>Extendleft</td>
<td>( O(l) )</td>
<td>( O(l) )</td>
</tr>
<tr>
<td>Rotate</td>
<td>( O(l) )</td>
<td>( O(l) )</td>
</tr>
<tr>
<td>Remove</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set Operations</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \in s )</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Union ( s</td>
<td>t )</td>
<td>( O(len(s) + len(t)) )</td>
</tr>
<tr>
<td>Intersection ( s&amp;t )</td>
<td>( O(min(len(s), len(t))) )</td>
<td>( O(len(s) * len(t)) )</td>
</tr>
<tr>
<td>Difference ( s - t )</td>
<td>( O(len(s)) )</td>
<td></td>
</tr>
<tr>
<td>( s\text{.difference}_\text{update}(t) )</td>
<td>( O(len(s)) )</td>
<td></td>
</tr>
<tr>
<td>Symmetric Difference ( s^*t )</td>
<td>( O(len(s)) )</td>
<td>( O(len(s) * len(t)) )</td>
</tr>
<tr>
<td>( s\text{.symmetric_difference}_\text{update}(t) )</td>
<td>( O(len(t)) )</td>
<td>( O(len(t) * len(s)) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dictionary Operations</th>
<th>Average Case</th>
<th>Amortized Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Average Case</td>
<td>Amortized Worst Case</td>
</tr>
<tr>
<td>( x \in s )</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Union ( s</td>
<td>t )</td>
<td>( O(len(s) + len(t)) )</td>
</tr>
<tr>
<td>Intersection ( s&amp;t )</td>
<td>( O(min(len(s), len(t))) )</td>
<td>( O(len(s) * len(t)) )</td>
</tr>
<tr>
<td>Difference ( s - t )</td>
<td>( O(len(s)) )</td>
<td></td>
</tr>
<tr>
<td>( s\text{.difference}_\text{update}(t) )</td>
<td>( O(len(s)) )</td>
<td></td>
</tr>
<tr>
<td>Symmetric Difference ( s^*t )</td>
<td>( O(len(s)) )</td>
<td>( O(len(s) * len(t)) )</td>
</tr>
<tr>
<td>( s\text{.symmetric_difference}_\text{update}(t) )</td>
<td>( O(len(t)) )</td>
<td>( O(len(t) * len(s)) )</td>
</tr>
<tr>
<td>Operation</td>
<td>$O(N)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>Copy</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Get Item</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Set Item*</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Delete Item</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Iteration</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

Table 3.1: Time Complexities of implemented Python data structures
4. Maximum Flow and Push-Relabel

The push-relabel algorithm is an algorithm to compute the maximum flow in a digraph by intermediate relaxation of one of the properties of the problem. This deviates from other algorithms like Ford-Fulkerson, which determine a maximum flow by iteratively expanding an existing flow. Though it is a problem that is inherently one in digraphs, it can be generalized to graphs by considering the dual orientation of the graph.

4.1 Flow and Maximum Flow

Let $D = (V, A)$ be a digraph, $s$ and $t$ two distinct vertices in $V$, called the source and the sink respectively, and $c : A \to (0, \infty)$ be a capacity function. We first expand $c$ to $V \times V$ by saying $c(u, v) = 0$ if $(u, v) \notin A$. Now we define a flow from $s$ to $t$ in $D$ to be a function $f : V \times V \to \mathbb{R}$ which has the following properties:

\[ f(u, v) \leq c(u, v) \quad \forall (u, v) \in V \times V \]  
\[ f(u, v) = -f(v, u) \quad \forall (u, v) \in V \times V \]  
\[ e_f(v) := \sum_{u \in V} f(u, v) = 0 \quad \forall v \in V \setminus \{s, t\} \]

We note that there is a trivial flow, namely $f \equiv 0$.

The quantity $e_f(v)$ is called the excess at $v$. These properties make sure that the flow over an arc is at most the capacity of the arc, and that for each vertex $v$ which is not $s$ or $t$, the flow into $v$ is equal to the flow out of it, meaning it has no excess. The value of a flow is the netto flow out of $s$, or into $t$, and is denoted $|f|$. This means $|f| = e_f(t)$.

In the current setting we can let $A$ be a regular set again instead of a multiset. Consider two vertices between which there are multiple arcs. If we replace these by one arc with the sum of the capacities of the arcs, then we can send an equal amount of flow over the replacement arc as over the multiple arcs. Furthermore, we can remove loops, since sending flow from a vertex to itself adds no value to the flow. This means $D$ can be considered to be a simple digraph, meaning $m \leq n^2$.

The anti-symmetry constraint (4.2) is not a standard one, but has some technical benefits. The most important one is that it prevents flow from being sent over two opposite arcs at the same time, which would pose some technical difficulties. Furthermore, it simplifies the definition of $e_f(v)$. This constraint and its benefits have already been reported as early as 1980 in [14].

A maximum flow is a flow that has a maximal value over all possible flows. An arc $(v, w)$ is called residual if $f(v, w) < c(v, w)$, and its residual capacity is $u_f(v, w) = c(v, w) - f(v, w)$.  

14
If an arc is not residual, it is called saturated. The residual graph $D_f = (V, E_f)$ is the digraph induced by the residual arcs. The flow graph is the digraph induced by the arcs with positive flow. The Maximum Flow problem is defined as:

**Problem:** Maximum Flow  
**Input:** $D = (V, A); s, t \in V, s \neq t; c: A \rightarrow (0, \infty)$  
**Output:** A valid flow $f$ in $D$ with capacities $c$ from $s$ to $t$ that has maximum value over all flows

### 4.2 The Push-Relabel Method

The idea of the push-relabel method is to relax the third property of a flow. By allowing the algorithm to create a preflow during the execution, instead of a flow, the algorithm gets simple yet efficient.

A **preflow** is a function $f: V \times V \rightarrow \mathbb{R}$ that has properties (4.1) and (4.2), but a relaxed version of property (4.3), namely (4.4):  

$$e_f(v) \geq 0 \quad \forall v \in V \setminus \{s, t\}$$  
**Positive Excess Constraint** (4.4)

A **labeling** is a function $d: V \rightarrow \mathbb{N}$, for which $d(t) = 0$ and for each residual arc $(v, w)$,  

$$d(v) \leq d(w) + 1.$$  

We say a residual arc $(v, w)$ is **admissible** if $d(v) = d(w) + 1$, and we call vertex $v$ **active** if $d(v) < n$ and $e_f(v) > 0$. For each arc $(v, w)$ we associate the three values $u_f(v, w), u_f(w, v)$ and $f(v, w)(= -f(w, v))$ with the edge $\{v, w\}$. Then, for each vertex $v$, we associate an **edge list**, consisting of all edges covering $v$ in an arbitrary but fixed order. This means that every edge is in two edge lists. Furthermore, at all times, each vertex has a **current edge**, one of the edges of its edge list, which is initially the first edge of the list.

The algorithm consists of two phases, of which the second is optional. The first phase creates a preflow that can be converted into a maximum flow. After the first phase, the value of a maximum flow is already computed, and can be extracted as being the value $e_f(t)$.

The second phase converts the calculated preflow to a maximum flow, after which all excess is removed, except for from $t$ and $s$.

**The First Phase**

Before starting the preflow calculation, an initial flow $f$ and a labeling $d$ that is valid for $f$ need to be chosen, after which $e_f$ and $u_f$ can be calculated. A possibility is to start with $d$ and $f$:

$$f(v, w) = \begin{cases} c(v, w) & \text{if } v = s \\ -c(w, v) & \text{if } w = s \\ 0 & \text{if } v, w \neq s \end{cases} \quad \forall v, w \in V \quad (4.5a)$$
$$d(v) = \begin{cases} n & \text{if } v = s \\ 0 & \text{if } v = t \\ 1 & \text{if } v \neq s, t \end{cases} \quad \forall v \in V \quad (4.5b)$$

Initializing $f$ and $d$ as in (4.5) will create a valid starting configuration, however, it does not create optimal starting conditions. Denote $\delta_f(v, w)$ the distance from $v$ to $w$ in the
residual graph, being $\infty$ if there is no path. A valid configuration that creates less useless looping is:

\[
\begin{align*}
f(v,w) &= 0 \quad \forall v,w \in V \quad (4.6a) \\
d(v) &= \min(\delta_f(v,t),n) \quad \forall v \in V \quad (4.6b)
\end{align*}
\]

The labeling in (4.6) can be calculated by running a backward breadth-first-search in the residual graph starting at $t$.

After initialization of $d$ and $f$, the actual calculation can be started. The calculation uses the following two basic operations throughout:

**Procedure** \textsc{Push}(\textit{v, w})

\textbf{Pre}: \textit{v} is active and $(\textit{v, w})$ is admissible

\textbf{Effect}: Pushes as much flow as possible from \textit{v} to \textit{w}

1. $\Delta = \min(e_f(v), u_f(v,w))$
2. $f(v,w) = f(v,w) + \Delta$
3. $f(w,v) = f(w,v) - \Delta$
4. $e_f(v) = e_f(v) - \Delta$
5. $e_f(w) = e_f(w) + \Delta$

**Procedure** \textsc{Relabel}(\textit{v})

\textbf{Pre}: \textit{v} is active and no arc $(\textit{v, w})$ is admissible

\textbf{Effect}: Relabels \textit{v}

1. $d(v) = \min\{d(w) | (v, w) \in E_f\} + 1$
   or \textit{n} if \textit{̸∃}(\textit{v, w}) \in E_f

These operations are combined in the following procedure, \textsc{Discharge}, which is the main calculatory part of the algorithm. The \textsc{Discharge} operation is applied until there are no applicable vertices anymore, after which the first phase is terminated.

**Procedure** \textsc{Discharge}(\textit{v})

\textbf{Pre}: \textit{v} is active

\textbf{Effect}: Push as much flow from \textit{v} until \textit{v} is not active, or relabeled

1. \textit{end-of-list} = \text{False}
2. \{\textit{v, w}\} is the current edge of \textit{v}
3. \textbf{repeat}
4. \textbf{if} $(\textit{v, w})$ is admissible
5. \textsc{Push}(\textit{v, w})
6. \textbf{elseif} \{\textit{v, w}\} is the last edge of the edge list
7. \textsc{make} the first edge of the edge list the current edge
   \textit{end-of-list} = \text{True}
8. \textbf{else}
9. \textsc{make} the next edge of the edge list the current edge
10. \textbf{until} $e_f(v) = 0$ or \textit{end-of-list}
11. \textbf{if} \textit{end-of-list}
12. \textsc{Relabel}(\textit{v})

16
The last decision to be made is the order in which the discharging of active vertices is applied. The selection procedure has several options, among which are Last In First Out (LIFO) order using a queue, and Highest Level (HL) order, in which an active node with the highest label is selected.

The algorithm starts with discharging s until it has label n, after which the selection criterium will select the vertices to discharge next, as depicted in PUSH-RELABEL:

**Algorithm** PUSH-RELABEL\( (G, s, t) \)

- **Pre:** \( s \neq t \)
- **Effect:** Compute a maximum flow in \( G \)

1. repeat
2. \( \text{Discharge}(s) \)
3. until \( d(s) = n \)
4. \( v = \text{Select}() \)
5. repeat
6. \( \text{Discharge}(v) \)
7. \( v = \text{Select}() \)
8. until \( v = \text{nil} \)

Experimental results in [15] show that HL selection is generally a better criterium, in which case Select will look like:

**Procedure** Select()

1. \( k = n - 1 \)
2. while there is no active node with label \( k \) and \( k \geq 0 \)
3. \( k = k - 1 \)
4. if \( k \geq 0 \)
5. return a vertex with label \( k \)
6. else return \( \text{nil} \)

Optionally, a variable stating the highest label can be kept throughout the algorithm. This will reduce the amount of time spent in Select, but it does not always state a valid highest label. For instance when \( v \) was selected, and is the only vertex with label \( d(v) \). If \( v \) is then relabeled to \( n \), the value does not correspond to the true highest label anymore, in which case a search similar to the one in the above procedure should be done.

**The Second Phase**

The second phase of the algorithm constructs an actual flow, from the calculated preflow. It does so by systematically pushing back excess.

To make sure that no excess is being pushed around in directed cycles, the preflow needs to be made acyclic. By starting a backward depth-first search in the flow graph, starting from all vertices with positive excess, except for \( s \) and \( t \), cycles can be detected. Flow on these cycles is reduced until the cycle is removed. When all cycles have been eliminated, the search produces a topological ordering in which excess can be pushed back to the source. When all excess has been pushed back, \( f \) follows property (4.3), meaning it has become an actual flow. At this point, it is also a maximum flow.
Heuristics

To speed up the algorithm, two effective heuristics have been developed, which are also explained in [15].

The first heuristic is global relabeling. The labeling of the vertices is supposed to be an approximation of the distance from the vertices to \( t \) in \( D_f \). However, it may not always be correct. Global relabeling periodically updates the labels to match the distance again, and can be performed by a backward breadth-first search in the residual graph, as in (4.6). The frequency of the application of the heuristic is a design decision. Since it only has effect when vertices have actually been relabeled, it is usually applied after a certain amount of relabelings. Running the heuristic after \( cn \) relabelings is most desirable, for a constant \( c \), and because of different problem families, a value of \( c = 1 \) is a good tradeoff, as stated by [15].

The second heuristic is gap relabeling. We define a gap to be an integer \( h \) with \( 0 < h < n \) such that no vertices have label \( h \). We define \( V^h = \{ v \in V | d(v) > h \} \) and \( V_h = \{ v \in V | d(v) < h \} \). When \( h \) is a gap then all vertices in \( V^h \) can never be active again, meaning we can relabel them to \( n \). A gap can be detected in constant time when relabeling, if for each label a list of vertices with that label are kept in memory. Note that after fully discharging \( s \) there is always a gap, as there are \( n + 1 \) labels for \( n \) vertices and labels 0 and \( n \) are always used. However, the gap relabeling heuristic is only interesting when \( h < n - 1 \).

4.3 Correctness of the Push-Relabel

We will show correctness of the algorithm. First the correctness of the algorithm that does not use heuristics is proven, after which the correctness of the heuristics is proven.

Correctness of the General Algorithm

First we show that the algorithm that does not use the heuristics is correct. We note that the only operation that alters flow is the \textsc{Push} operation, and it maintains a valid flow, as well as maintains the excess value correctly. Also, without heuristics, only the \textsc{Relabel} operation alters the labeling, and it clearly maintains a valid one.

The \textsc{Discharge} operation pushes flow along the arcs covering the vertex until either there is no excess in the vertex, or the end of the edge list is reached, in which case the vertex is relabeled. Quite obviously, the \textsc{Push} is called only on admissible arcs.

\textbf{Theorem 1.} The operation \textsc{Discharge}(v) only calls \textsc{Relabel}(v) when it is applicable.

\textbf{Proof:} Suppose we are in a call of \textsc{Discharge}(v). Let \( \{v, w\} \) be the current edge of \( v \). We make the following case distinction based on residuality of \( v, w \) and the labeling of \( v \).

\textbf{Case 1:} \( (v, w) \) is saturated

In order for \( (v, w) \) to have residual capacity once again, a call to \textsc{Push}(w, v) needs to be made. However, this would mean that \( d(w) = d(v) + 1 \), in which case \textbf{case 2} applies as \( d(v) = d(w) - 1 < d(w) + 1 \).

\textbf{Case 2:} \( (v, w) \) is residual and \( d(v) < d(w) + 1 \)

Since labels can only increase, the equality \( d(v) = d(w) + 1 \) can never hold before relabeling \( v \).
**Case 3:** \((v, w)\) is residual and \(d(v) = d(w) + 1\)

Now the call \(\text{Push}(v, w)\) can be made. If that call saturates \((v, w)\), then **case 1** applies. If it does not saturate \((v, w)\), then the current edge is not updated, and the current call to \(\text{Discharge}(v)\) is ended.

In all cases, either the current edge can never be admissible (again) before relabeling \(v\), meaning we can safely go to the next edge of the edge list, or the call ends, meaning the current edge of \(v\) is not updated, and the case distinction can be repeated in the next call of \(\text{Discharge}(v)\).

Therefore, after all edges on the list have been processed, none of them can become admissible before relabeling \(v\), meaning we can safely relabel \(v\). ■

The next step is to prove that when no \(\text{Discharge}\) operations can be applied, the first phase of the algorithm is done. We show that at this moment no flow can be pushed to \(t\) anymore.

**Theorem 2.** If no \(\text{Discharge}\) operation can be applied then no flow can be pushed to \(t\) anymore.

**Proof:** If no \(\text{Discharge}\) operation can be applied, then there are no active vertices, meaning each vertex \(v \in V \setminus \{s, t\}\) either has label \(n\) or \(e_f(v) = 0\). Now suppose that flow can still be pushed to \(t\). Then there is a vertex \(v\) with positive excess and a path from \(v\) to \(t\) in \(D_f\). Let \((v = v_0, ..., v_k = t)\) be this path. Since \(v\) has positive excess, but there are no active vertices, \(v\) must have label \(n\). However, \(d(v_i) \leq k - i\), as we will show with induction:

**Base:** \(i = k\)

Since \(v_i = v_k = t\), \(d(v_i) = d(t) = 0 = k - i\).

**Step:** \(0 \leq i < k\)

We know that \(d(v_{i+1}) \leq k - (i + 1)\). Since the path is one in the residual graph, \((v_i, v_{i+1})\) is residual, which means \(d(v_i) \leq d(v_{i+1}) + 1 \leq k - (i + 1) + 1 = k - i\).

Now we know \(d(v) \leq k\). Since paths can only have length at most \(n - 1\), \(k \leq n - 1\), which means \(d(v) \leq n - 1\), which contradicts \(d(v) = n\). Therefore no flow can be pushed to \(t\) anymore. ■

The correctness of the second phase is shown using the following theorem:

**Theorem 3.** After the second phase, \(f\) is a flow.

**Proof:** We will show that the second phase pushed back all excess to \(s\), meaning no vertices other than \(s\) and \(t\) have excess. Since all other properties of flow are kept throughout the algorithm, \(f\) is a valid flow.

Let \(h\) be the largest gap. Then all arcs from \(V^h\) to \(V_h\) are saturated, meaning there are no arcs from \(V_h\) to \(V^h\) in the flow graph. Also, since there are no active vertices, all vertices with positive excess have label \(n\), meaning they are in \(V^h\). This means that a depth-first search from the vertices with positive excess in the flow graph can never leave \(V^h\). Let \(v_0, ..., v_k\) be the topological ordering resulting from the depth-first search after cycle removal in the reversal of the flow graph.

19
Since such a search does not reach isolated vertices and \( s \) is the only vertex with positive outflow but zero inflow, \( u_0 = s \). In the topological ordering we push back all excess to neighbors. The first vertex must have zero indegree, so it can never receive excess again. For a subsequent vertex \( v \), the vertices before \( v \) in the ordering already have zero excess, meaning \( v \) can never receive excess after being processed, as those were all vertices that could have arcs into it. Furthermore, when a vertex \( v \) other than \( s \) has excess, \( v \) must have received at least as much flow, meaning the excess back can always be pushed back. So we can safely push all excess back to \( s \).

After all vertices except \( s \) have pushed back their excess, there are no vertices with excess except \( s \) and \( t \), as all these vertices have been used as initial vertices in the depth-first search. This means that \( f \) is now a flow.

Since no more flow can be pushed to \( t \) after the first phase, and after the second phase \( f \) is a flow, \( f \) is a maximum flow.

**Correctness of the Heuristics**

The global relabeling heuristic sets \( d(v) \) to the lesser of \( n \) and the distance from \( v \) to \( t \) in the residual graph, by using a backward breadth-first search from \( t \).

**Theorem 4.** \( d(v) = \begin{cases} \delta_f(v, t) & \text{if } \delta_f(v, t) < \infty \\ n & \text{if } \delta_f(v, t) = \infty \end{cases} \) is a valid labeling.

**Proof:** Obviously \( d(t) = 0 \). Let \( v \) be any vertex that can reach \( t \), but not \( t \), and let \( w \) be any vertex such that \((v, w)\) is residual. Suppose \( w' \) is the vertex such that \((v, w')\) is traversed (backwards) by BFS. Then \( d(v) = d(w') + 1 \). If \( w = w' \) then obviously \( d(v) \leq d(w) + 1 \). If \( w \neq w' \), then the shortest path from \( w \) to \( t \) is at least as long as the shortest path from \( w' \) to \( t \), thus \( d(w) \geq d(w') \), meaning \( d(v) = d(w') + 1 \leq d(w) + 1 \).

Now let \( v \) be a vertex that cannot reach \( t \). Then its distance to \( t \) is \( \infty \), but since \( d(v) \) is set to the lesser of \( n \) and the distance, it is set to \( n \). Let \((v, w)\) be any residual arc. Since \( t \) is unreachable from \( v \), \( t \) is also unreachable from \( w \), thus \( d(v) = n = d(w) \), and thus trivially \( d(v) \leq d(w) + 1 \).

Thus the acquired labeling is a valid labeling.

The Gap Relabeling heuristic is based on a gap in the labels, meaning an integer value \( h < n \) such that there is no vertex \( v \) with \( d(v) = h \), but there are vertices with \( h < d(v) < n \). Then all these vertices are relabeled to \( n \).

**Theorem 5.** Let \( h \) be a gap, then \( d'(v) = \begin{cases} n & \text{if } d(v) > h \\ d(v) & \text{if } d(v) < h \end{cases} \) is a valid labeling.

**Proof:** We consider any residual arc \((v, w)\).

- If \( v, w \in V^h \) then \( d'(v) = n = d'(w) \leq d'(w) + 1 \)
- If \( v, w \in V_h \) then \( d'(v) = d(v) \leq d(w) + 1 = d'(w) + 1 \)
- If \( v \in V_h, w \in V^h \) then \( d'(v) = d(v) < h < n + 1 = d'(w) + 1 \)

The last case, \( v \in V^h, w \in V_h \) leads to a contradiction, because then \( d(v) \geq h + 1 \geq (h - 1) + 2 \geq d(w) + 2 \), which contradicts the validity of \( d \).
Since all cases either show the validity of $d'$, or contradict the validity of $d$, $d'$ is a valid labeling, and the Gap Relabeling heuristic maintains a valid labeling.
5. Implementation Details and Encountered Problems

While a theoretically correct algorithm is a useful starting point, there is more to an implementation than converting these procedures to code. In this case, as the algorithm is implemented for Sage, quite some data structures and standard (di)graph algorithms have already been implemented. Examples that have been used are:

1. Graph and DiGraph data structures
2. vertex-, edge-, and neighbor-iterators
3. Breadth- and Depth-first search
4. (Directed) Cycle detection
5. DoublyLinkedList data structure

5.1 Implementation Details

Since vertex identifiers in the Graph and DiGraph structures are not restricted to being integers in the range \([0, n]\), the functions such as \(d\), \(e_f\) and \(f\) cannot be implemented using lists, which are integer indexed in Python. Luckily, Python has dictionaries as natively implemented data structures, in which any hashable object can be a key. Since vertices also need to have hashable identifiers, dictionaries can be used for representing these functions.

Furthermore, the capacity function is not given as an argument, but rather as a set of labels on the edges of the (Di)Graph structure, meaning they have to be extracted as well, first.

The flow, capacity, label and excess functions are implemented using dictionaries, while the edge lists are kept as a dictionary of lists. Another dictionary determines for each vertex the index of the current edge in the edge list. Since the label lists and active lists are identified by labels, which are integers, they can be implemented as lists of Doubly Linked Lists, which allows constant time insertion and deletion of any element.

When these data structures have been initialized, the procedures \texttt{Push}, \texttt{Relabel} and \texttt{Discharge} can almost trivially be converted to Python code. However, since the lists of active nodes and nodes for each label are kept in memory, this needs to be added as well. The \texttt{Select} procedure can have loop guard of \texttt{active_lists[k].is_empty()} and \(k > 0\), while returning \texttt{active_lists[k].head()}, assuming \texttt{active_lists} is the list of DoublyLinkedLists of active vertices, and the \texttt{head} method returns \texttt{nil} when empty, which is the case here.

For the second phase, the actual flow graph needs to be constructed. Sage has an implementation of an algorithm which detects directed cycles if there are any, or else returns a
topological ordering. By running this algorithm, directed cycles can be detected. If a cycle is found, the minimal flow over it can be calculated, say $f_{\text{min}}$, and all flow over it can be decremented by $f_{\text{min}}$, deleting the cycle from the flow graph. Afterwards, excess can be pushed back from vertices in reverse topological order. When pushing back the excess, $t$ must be skipped, because otherwise a zero flow is being constructed, as $t$ has exactly $|f|$ excess.

The global relabeling heuristic is implemented using the available breadth-first search algorithm on the Residual graph, using $t$ as the starting vertex and specifying the neighbor function to return only incoming neighbors.

The gap relabeling heuristic is implemented as a simple check while relabeling. After removing the vertex from the label list of its old label, the list is checked to be empty. If it is empty, all label lists and active vertex lists of higher labels are emptied, and vertices in it are relabeled to $n$.

5.2 Encountered Problems and Provided Bugfixes

During the implementation of the algorithm, several problems have been encountered, either in the form of bugs, or as deficiencies in implemented algorithms. The most prominent ones will be listed here.

Breadth-First Search Distance

The implemented breadth-first search algorithm has some very nice properties, such as the possibility to give multiple vertices as starting points, possibility for custom definition of a neighbor function, a maximal depth to be visited and the possibility to ignore directions in a directed graph. The custom neighbor function has been put to good use in the implementation, because by defining the neighbor function to be the implemented neighbor-in-iterator, a backwards breadth-first search can be easily done. However, the implementation only returns a breadth-first ordering of the vertices, without a calculated distance. Unfortunately, the distance was exactly what was needed to give an initial labeling, as well as for the global relabeling heuristic. Another implemented algorithm was able to give a breadth-first ordering along with the distances, but this implementation had no option to specify the neighbor function.

For the above reason, an optional argument has been added to the breadth-first search implementation, which decides on whether to report distances with the vertices as well or not. The argument was set to false by default so as not to break other implementations that use it.

DoublyLinkedList bug

The Doubly Linked List (DLL) that is implemented in Sage is actually not really a DLL, but rather a list of pre-defined elements that are hidden or unhidden, that also keeps ordering. Though this is not a problem for the push-relabel algorithm, as a graph has a predefined set of vertices, it is not an actual DLL. However, there was a problem with empty lists. When all elements of a DLL were hidden, and one was unhidden, all that came after it in the original list were also unhidden.

A restructuring of the data to become an actual DLL has been proposed, which at the same time removes the bug. The new structure will not maintain the order of the predefined
elements, as the old structure did, since in principle a DLL has no concept of maintenance of order upon insertion and removal. There is the possibility that this adaptation will break some implemented algorithms in Sage which may have depended on the order that was kept. However, if this is the case the new implementation could be created besides the original implementation, possibly renaming one of them.

**Edge Cut bug**

Though this is not a bug affecting the implemented algorithm, it is a bug in an implementation that could use the Push-Relabel algorithm. The algorithm to calculate a minimal edge cut between two vertices had a bug when called on certain graphs or digraphs, in which case the correct value of the cut but a set of edges and two sets of vertices that do not represent the actual minimal cut would be returned. This algorithm is based on reachability in the Residual Graph of the calculated flow. The construction of the residual graph was faulty, which caused vertices to be on the wrong side of the cut, giving a wrong cut.

A simple fix has been proposed, which fixes the implementation for the instances that have been found to give faulty output, and should make sure that the function always returns correct cuts now.

**Gomory-Hu tree bug**

This bug is again not a bug on which the implemented algorithm depends, but one in an algorithm that uses it. The Gomory-Hu tree algorithm uses a minimal edge cut which creates a division of the graph into two smaller graphs, on which it recurses. The trees obtained from this recursion are then combined with a single edge. However, the vertices between which this edge is added were chosen arbitrarily, while there is a very specific pair of vertices to which the edge should have been added, which caused incorrect trees to be returned.

A small restructuring of the algorithm has been proposed, which uses partitions of $V$ to determine the correct edge, as defined in [1].
6. Optimizing the Implementation

The next step after creating a correctly working algorithm was to start profiling the code to find bottlenecks. Using other structures for data or other methods of computation these bottlenecks could be optimized. Profiling was done using Python’s implemented profiling tools, which state how long each function took, both including and excluding called subroutines, as well as how many times it was called. The most important improvements are listed.

6.1 Dictionary Elimination

When profiling, it quickly became apparent that a lot of time was spent getting items from dictionaries, especially from the capacity and flow dictionaries, since quite some time was spent reading flow and capacity, and calculating residual capacity, as well as setting new flow. The solution to this problem comes with initialization overhead, but gives an incredible speed up. By using a single directory that maps vertices of the graph to integer indices in the range \([0, n]\) almost all data can be converted to lists, which have a guaranteed access time of \(O(1)\), as well as a practically much faster access time. The flow and capacity now become matrices, the active vertices and label lists become lists of DoublyLinkedLists, the edge lists become lists of lists, and the excess, label and current edge functions become regular lists. This change created quite a revamp of the entire code, but established an incredible speed up.

6.2 List Length Calculation

After the dictionaries had been eliminated, the profiling data showed that quite some time was spent calculating the length of lists. This function was only used once in the entire code, to check whether the current edge was the last. Since the edge lists were constant during the entire algorithm, the length of these lists are constant as well. This led to a new list variable that saved the length of the edge list for each vertex, meaning a few extra lines of initialization code. But at the same time it meant full elimination of length calculation, by replacing those calls with a single list access.

6.3 Residual Graph Construction and Maintenance

The next profiling run showed that a significant amount of time was spent constructing the residual graph on initialization and maintaining it while pushing flow over arcs. Though attempts have been made to optimize the number of arc insertions and removals, these adaptations made little difference. This brought up the question whether the global relabeling
heuristic was actually beneficial in this implementation. Therefore optional arguments for the algorithm were created which decided whether or not to use the heuristics. Testing with all 4 possible combinations of settings showed that it was in general faster not to use global relabeling but still use gap relabeling.
7. Experimental Results

After optimizing the implementation to best knowledge, profiling was pointed towards the use of comparing the new implementation with the available implementation of the maximum flow algorithm in Sage, the Ford-Fulkerson algorithm using the Edmonds-Karp method for path determination.

7.1 Test Environment and Performed Tests

The test environment is a Lenovo Thinkpad W520, with 2.0 GHz Quad-core processor and 4GB of RAM. The tests were run on Ubuntu 14.04 running Sage 6.2.

Initially all graphs and digraphs on at most 5 vertices were used for testing the correctness of the implementation. This test was run because it eliminates small edge cases that might pose problems. However, they also gave an insight in the performance of the implementation.

Furthermore, randomly generated digraphs on 10, 20, 50, 100, 200, 500, 1000 and 2000 vertices with density probabilities of 0.2, 0.4, 0.6 and 0.8 have been tested. These were generated using the sage command `DiGraphs.RandomDirectedGNP`. When testing randomly generated digraphs of size $n \leq 10000$ instances were tested to keep running times tractable. Furthermore, tests involving only flow value calculation were done as well as tests involving flow value and graph calculation.

7.2 Performance Results

The results of the graphs and digraphs on at most 5 vertices are presented in Table 7.1. The results of the randomly generated digraphs are shown in Figure 7.1 through Figure 7.8 in Log-Log scale. This configuration is chosen because for larger graphs, the Push-Relabel algorithm is actually a number of times faster.

The figures show that, though not always faster, there is definitely a trend towards the Push-Relabel implementation being faster, both practically and asymptotically. As graphs grow bigger, both in number of vertices and number of edges, the Push-Relabel implementation becomes several times faster.
### Table 7.1: Small (Di)Graph Calculation Timings

<table>
<thead>
<tr>
<th>Instance</th>
<th>Value Only</th>
<th>Value and Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FF</td>
<td>PR</td>
</tr>
<tr>
<td>Up to 4 vertices</td>
<td>0.181</td>
<td>0.156</td>
</tr>
<tr>
<td>On 5 vertices</td>
<td>20.709</td>
<td>18.436</td>
</tr>
</tbody>
</table>

- Push–Relabel
- Ford–Fulkerson

**Figure 7.1: Value Only Calculation with Density of 0.2**

**Figure 7.2: Value Only Calculation with Density of 0.4**

28
Figure 7.3: Value Only Calculation with Density of 0.6

Figure 7.4: Value Only Calculation with Density of 0.8
Figure 7.5: Value and Flow Graph Calculation with Density of 0.2

Figure 7.6: Value and Flow Graph Calculation with Density of 0.4
Figure 7.7: Value and Flow Graph Calculation with Density of 0.6

Figure 7.8: Value and Flow Graph Calculation with Density of 0.8
8. Conclusion

The investigated problems all have an efficient algorithm to solve them. Most of them already have an implementation in Sage, though not an optimal one. Of these problems, the one chosen to be improved is the Gomory-Hu tree algorithm, including its subproblems, the Minimal $s - t$ Edge-Cut problem, and the Maximum Flow problem.

Sage already had an implementation to solve the Maximum Flow problem based on the Ford-Fulkerson algorithm. However, the Push-Relabel algorithm is a theoretical improvement over the Ford-Fulkerson algorithm in terms of time complexity, and its implementation has been shown to be a practical improvement in general in terms of running time as well. This improvement propagates to the Minimum $s - t$ Edge-Cut implementation and the Gomory-Hu tree implementation. This creates an improved running time for all three implementations.

Furthermore, bug fixes have been provided for the implementations of the Minimum $s - t$ Edge-Cut and the Gomory-Hu tree algorithms, improving their correctness.

From a personal viewpoint, the project has been a success. I have learnt about some algorithmic problems and got a general idea of how they are solved efficiently. I did some deepening research into one particular algorithm, the Push-Relabel algorithm, for solving the Maximum Flow problem. Though the problem was known to me, the algorithm was unknown, and quite intrigued me, as it takes a different approach than algorithms I knew before. In retrospect, the results are better than I could have hoped for. For these reasons, the learning experience and the positive results, I think the goal of the project has been achieved.
Bibliography


