Reversible Graph Algorithms

Sarah Vang Nøhr

noehr@di.ku.dk

Saturday 17th January, 2015
Abstract

The field of reversible computing is still in its early development and much ground work needs to be done. In this project, the focus has been on creating a selection of common graph algorithms in reversible form, as well as their underlying data structures.

One of the main concerns in regards to reversible implementations of algorithms and other programs are garbage, i.e. output that is not in part of the irreversible version’s output yet is part of the reversible version’s output, and it has been attempted to reduce the garbage for all of the algorithms to the lowest possible amount.

The project has focused on creating reversible implementations of the algorithms Breadth-First Search, Depth-First Search, Kruskal’s Algorithm, Prim’s Algorithm and Dijkstra’s Algorithm. This has been a partial success, resulting in clean, i.e. garbage free, versions of Breadth-First Search, Depth-First Search and Kruskal’s Algorithm. As for the remaining two, the garbage stems from the use of the minimum priority queue data structure, which does not have optimal garbage size. All of the reversible algorithms covered in this report, upholds the same time complexity as their irreversible counterparts, with the exception of Prim’s algorithm.

Resume

Feltet reversibel databehandling er stadig i begyndelsen af sin udvikling, og der er meget grundarbejde, der skal klares. I dette projekt har der været fokus paa at konvertere en række almindelige grafalgoritmer til deres reversible form, saavel som deres underbyggende datastrukturer.

Der har gennem projektet været fokus paa at reducere garbage, dvs. output, der ikke er en del af en irreversibel versions output, men er en del af output i en reversibel implementation, til det lavest mulige.

1 Introduction

The world of computers changes rapidly and constantly, particularly in regards to power. The constant development is both positive and negative - on one hand it helps with innovation, making it possible to reach better living standards and solving previously unsolvable problems, on the other hand it causes a massive drain on the world’s electricity. As Moore’s law continues to hold, it is only reasonable to assume that the power problem will continue to rise, unless something is done to prevent it. In 1961, Rolf Landauer theorized that it would be possible to create a computer that requires minimal or no release of heat[4]. This led to the field of reversible computing.

For reversible computations, it holds that every computation has an inverse, or reverse, computation, resulting in both forward and backward determinism. This means that, given a reversible program $p$, we are not only able to perform a forward calculation using input $x$, returning output $y$, we are also able to use this output $y$ with the program $p$ to obtain the input $x$ once more.

While most computations are reversible, one special type of computation is not possible to do reversibly - erasure of data. This means that all variables must either be stored globally and thus be part of the output, or local variables must be able to be restored using other variables or set constants. One of the ways of restoring variables to a previously known value after a call to a procedure is using the Bennett Trick[2], where you use or store the values needed, then uncall the procedure with parameters that have identical values to the ones used in the original call to the procedure in order to cancel out the variables. However, we are not always able to restore these local variables, which means that we are instead forced to store information that could otherwise have been reverted.

Any information that is left over when running a reversible program that would not have been present in a irreversible version of the same program, is referred to as garbage. We want to reduce this garbage in order to reduce the space complexity of a given program. To describe the garbage output of a program, we use the terms faithfulness and hygiene as termed by Axelsen and Yokoyama[1]. A faithful reversible implementation of a program has the same time complexity as the irreversible program, without requiring optimal garbage, ie. minimal, whereas a hygienic reversible implementation is a faithful reversible implementation with the added requirement that the garbage size is optimal. We generally strive for a hygienic implementation when converting an irreversible program to a reversible version, however, it can be easier to first create it to a faithful implementation, maintaining the same time complexity, and then work on reducing the garbage output.

In this project, I have focused on transforming a variety of common irreversible graph algorithms to their reversible forms, with focus on reducing garbage output where possible. As will be seen, almost none of the implementations are entirely clean of garbage output, however most have successfully been converted to hygienic versions. The algorithms covered in this project are Breadth-First Search and Depth-First Search (Sections 4.1 and 4.2) used to get a feeling of the basic functionality of reversible graph algorithms, Kruskal’s and Prim’s algorithms (Sections 4.3 and 4.4), comparing two different techniques to find a minimum spanning tree reversibly and Dijkstra’s algorithm (Section 4.5), used to solve the single source shortest path problem. This selection represent a nice variety of graph algorithms, while showcasing the problems and pitfalls involved when working with reversible programming. All data
structures and algorithms have been implemented based on their irreversible versions as described in Introduction to Algorithms (CLRS)\textsuperscript{3}.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Data structures needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First Search</td>
<td>Adjacency list, FIFO queue</td>
</tr>
<tr>
<td>Depth-First Search</td>
<td>Adjacency list</td>
</tr>
<tr>
<td>Kruskal’s Algorithm</td>
<td>Adjacency matrix, Disjoint set, a sorting algorithm</td>
</tr>
<tr>
<td>Prim’s Algorithm</td>
<td>Adjacency list, Minimum-priority queue</td>
</tr>
<tr>
<td>The Bellman-Ford Algorithm</td>
<td>Adjacency list</td>
</tr>
<tr>
<td>Dijkstra’s Algorithm</td>
<td>Adjacency list, Set, Minimum-priority queue</td>
</tr>
<tr>
<td>The Floyd-Warshall Algorithm</td>
<td>Adjacency matrix</td>
</tr>
<tr>
<td>Johnson’s Algorithm</td>
<td>Adjacency matrix, uses Bellman-Ford and Dijkstra</td>
</tr>
<tr>
<td>Ford-Fulkerson’s Algorithm</td>
<td>Adjacency list, uses BFS</td>
</tr>
</tbody>
</table>

Figure 1: Overview of some common graph algorithms and their required data structures. The algorithms in bold as well as their needed data structures have been implemented and covered in this report.

As of this moment, there has been little exploration into the world of reversible algorithms. While Axelsen and Yokoyama has discussed and implemented reversible versions of comparison sorts\textsuperscript{1}, thus making the initial considerations as to what to consider when converting algorithms, it seems as though the rest of the algorithm categories remain untouched. This creates an initial challenge, as there is no other basis for comparison, as well as the lack of reversible data structures. Because of this, initial work had to be done in creating the needed data structures, and doing so in an optimal way both in regards to garbage and time complexity. We use a large section of this report to cover these data structures before moving on to the actual algorithm implementations. An overview of the algorithms and their required data structures can be seen in Figure \textsuperscript{1}.

The project was supervised by Holger Bock Axelsen, postdoc in the Algorithms and Programming Languages at DIKU.
\[ \text{program} ::= \text{(procedure } \langle \text{identifier} \rangle \\text{ ( } \langle \text{declaration} \rangle^* \text{ ) } \langle \text{declaration} \rangle^* \\text{ (statement)}^* )^+ \]

\[ \text{declaration} ::= \text{int} \text{ (variable)} \\
\quad | \text{int} \text{ (variable)} = \langle \text{expression} \rangle \\
\quad | \text{int} \text{ (variable)}[(\text{expression})] \\
\quad | \text{stack} \text{ (variable)}[(\text{expression})] = \\
\quad \{ \langle \text{expression} \rangle, (\langle \text{expression} \rangle)^* \} \\
\quad | \text{stack} \text{ (variable)}[(\text{expression})][(\text{expression})] = \\
\quad \{ \langle \text{expression} \rangle, (\langle \text{expression} \rangle)^* \} \\
\quad | \text{stack} \text{ (variable)}[(\text{expression})][(\text{expression})][(\text{expression})] = \\
\quad \{ \langle \text{expression} \rangle, (\langle \text{expression} \rangle)^* \} \]

\[ \text{statement} ::= \langle \text{variable} \rangle \text{ (function)} = \langle \text{expression} \rangle \\
\quad | \langle \text{variable} \rangle[(\text{expression})] \text{ (function)} = \langle \text{expression} \rangle \\
\quad | \langle \text{variable} \rangle[(\text{expression})][(\text{expression})] \text{ (function)} = \langle \text{expression} \rangle \\
\quad | \text{if} \langle \text{expression} \rangle \text{ then } \langle \text{statement} \rangle^* \text{ (else } \langle \text{statement} \rangle^*)? \text{ fi} \langle \text{expression} \rangle \\
\quad | \text{from} \langle \text{expression} \rangle \text{ do } \langle \text{statement} \rangle \text{ until } \langle \text{expression} \rangle \\
\quad | \text{push} \langle \langle \text{variable}, \langle \text{identifier} \rangle \rangle \rangle \\
\quad | \text{pop} \langle \langle \text{variable}, \langle \text{identifier} \rangle \rangle \rangle \\
\quad | \text{local int} \langle \text{variable} \rangle = \langle \text{expression} \rangle \\langle \text{statement} \rangle^* \text{ delocal int } \langle \text{variable} \rangle = \langle \text{expression} \rangle \\
\quad | \text{call} \langle \text{identifier} \rangle \langle (\langle \text{identifier}, \langle \text{identifier} \rangle)^* \rangle? \rangle \\
\quad | \text{uncall} \langle \text{identifier} \rangle \langle (\langle \text{identifier}, \langle \text{identifier} \rangle)^* \rangle? \rangle \\
\quad | \langle \text{identifier} \rangle \text{ <= } \langle \text{identifier} \rangle \\
\quad | \text{error} \langle \langle \text{stringliteral} \rangle \rangle \\
\quad | \text{print} \langle \langle \text{stringliteral} \rangle \rangle \\
\quad | \text{printf} \langle \langle \text{stringliteral}, \langle \text{identifier} \rangle^* \rangle \rangle \\
\quad | \text{show} \langle \langle \text{identifier} \rangle^* \rangle \\
\quad | \text{skip} \\
\quad | \text{statement} \text{ statement} \langle \text{statement} \rangle^* \]

\[ \text{expression} ::= \langle \text{constant} \rangle \mid \langle \text{variable} \rangle \mid \langle \text{variable} \rangle[(\text{expression})] \\
\quad | \langle \text{variable} \rangle[(\text{expression})][(\text{expression})] \\
\quad | \langle \text{expression} \rangle \langle \text{operator} \rangle \langle \text{expression} \rangle \\
\quad | \text{empty} \langle \langle \text{identifier} \rangle \rangle \mid \text{top} \langle \langle \text{identifier} \rangle \rangle \mid \text{size} \langle \langle \text{identifier} \rangle \rangle \\
\quad | \text{nil} \]

\[ \text{function} ::= + \mid - \mid ^ \]

\[ \text{operator} ::= \langle \text{function} \rangle \\
\quad | * \mid / \mid \% \mid ^* \mid \& \mid \mid \mid \&\& \mid || \mid ( / ) \mid = \mid != \mid (= / )= \]

Figure 2: Syntax of the Janus language.
2 Preliminaries

2.1 The Janus Language

This section contains a short introduction to the Janus language, with main focus on
the elements we will be using the most and which differ from more common languages,
as well as a short analysis of the limitations of the language in its current form and the
effects this have on the algorithms presented in this report. The syntax of the Janus
language can be seen in Figure 2, and a small example of a program implementation
can be seen in Figure 3.

2.1.1 General

The Janus language is a reversible programming language, meaning that each calcu-
lation performed in a program must have an opposite action which can nullify the
initial calculation. Because of this, some actions which are possible in irreversible
programming are impossible in reversible programming.

As an example, after the initial creation of local or non-local variables, the variables
cannot directly be set to other values. Instead, reversible actions such as addition or
the swapping of two elements can be performed.

The Janus language uses keywords to determine the structure of the program,
not relying on brackets or indentation to keep track of scope, or semicolons to end
statements, which is otherwise common in many of the most used languages today.

A Janus program can consist of several procedures, but must include one main
procedure which initializes the program. When initializing a Janus program, it is
required to not only give the input variables, but also to create variables for any
extra output that is expected, since the language only allows the creation of non-local
variables at the very start of the main procedure.

2.1.2 Types and structures

The Janus language currently only supports integers, integer arrays, including two-
dimensional arrays, and stacks, except in the print and error functions, which accepts
local int i = 0, found = 0, x = 3
for i = 0 do
  if array[i] = x then
    found += 1
  fi
  array[i] = x
  i += 1
until found = 1
delocal int i = ?, found = 1, x = 3

Figure 4: An example of a loop where we search a list for the element x, returning once we have found it. We have no way of finding out the value of i in this implementation, so we cannot delocal it.

string literals. It is not possible to create custom structures or types.

This has proven to be a bit of a challenge when creating new data structures, but in most cases it has been possible to use two-dimensional arrays as simple structures, storing the necessary values as integers. However, if this feature gets implemented, it will most likely be possible to lessen the garbage and running time of some of the algorithms, aside from increasing the general readability of the code.

2.1.3 local and delocal

local and delocal are used to store and remove local values, which is not wanted in the final output of a procedure. The delocal value of the variable is used when going backwards through a procedure, and must be obtainable either by using another variable or by using a constant. Additionally, the delocal value is checked against the value of the variable upon delocation, to ensure that the two are equal, and vice-versa when moving backwards.

Finding a correct value when it is time to delocal the variable can prove tricky. In loops, it can end up costing extra cycles, since it is not possible to simply add or subtract the missing amount from the counter value in some cases - unless there was a value inside the loop body which could be used, it would be dependent on using the value of the counter as a subtraction/addition, which would be irreversible. An example of this can be seen in Figure 4.

Reversibility of course also prevents multiplying numbers with 0, as there would be no way of knowing the initial value afterwards.

2.1.4 if and from statements

The trickiest part of the language and reversibility is the conditional statements. When creating a from loop or an if statement, one has to make sure that the until and fi conditional statements, that is, the checks that are performed when going in the reverse direction, are valid and provide reversibility. For a condition to be valid, it must hold that it takes the right branch when and only when it would have taken the same branch when going in the forward direction. The fi and to are evaluated in each forward iteration to ensure that it holds true when the loop or first branch has been entered, and false if not.

The until and fi conditions are often very different from the from and fi conditions, since the body of the statements can change the variables greatly. Sometimes,
it is impossible to make the reverse conditions without adding some sort of garbage data to keep track of when things have been changed. This will be explained more in detail when it occurs in the algorithms in the report.

2.1.5 call and uncall

call and uncall are used to call procedures - however, where call runs the procedures normally, uncall runs them in reverse. It might not be clear how useful uncall is, however, it is incredibly useful for eliminating garbage, and is used in the Bennett Trick. As explained in local and delocal, it is necessary to know the value of a variable when using delocal. Since uncall can take the output from a procedure call and return the input, it can be used to neutralize any local values that was used in the procedure. This is a trick that is used in almost all of the algorithms.

2.1.6 Note on pointers

The Janus language does not currently support the functionality of pointers. Throughout the report the term pointer will instead be used to refer to the index in a specific array. These “pointers” will not be able to refer to other types.

2.2 Faithfulness and Hygiene

Throughout this report, we will use the terms faithfulness and hygiene when discussing the garbage output of reversible implementations. These terms have been coined by Holger Bock Axelsen and Tetsuo Yokoyama in an article\cite{Axelsen1}, which at the time of this report has not yet been published. In simple terms, the two terms mean:

- **Faithfulness**: A faithful, reversible algorithm preserves the time complexity of the irreversible algorithm. The limit to the garbage size of the algorithm is described as a function dependent on the input size.

- **Hygiene**: A hygienic, reversible algorithm has the same qualities as a faithful, reversible algorithm, with the added condition that the function describing the garbage size must be optimal, ie. minimal.

For a formal definition, we look at the definitions from the article\cite[sec.2.1]{Axelsen1}

\begin{quote}
“Given (irreversible) program $p$ of language $\text{IR}$ and reversible simulation $q$ of $p$ in language $\text{R}$, we say that program $q$ is a faithful reversible simulation of $p$ with garbage bound $g$, where $g : \mathbb{N} \to \mathbb{N}$, if there is a constant $c$, which may depend on the programming languages $\text{IR}$ and $\text{R}$, but not on the input $x$, satisfying the following conditions:

- bounded garbage output: $|\text{snd}([q]_\text{R}(x))| \leq c \cdot g(|x|)$ for all $x$,
- no asymptotic overhead time: $\text{time}^\text{R}_q(x) \leq c \cdot \text{time}^\text{IR}_p(x)$ for all $x$,
- at most $g$ extra space: $\text{space}^\text{R}_q(x) \leq c \cdot (\text{space}^\text{IR}_p(x) + g(|x|))$ for all $x$,”
\end{quote}
where \(|z|\) is the size of data \(z\) in the binary representation. Here, \(\text{time}^{L_p}_p(d)\) represents the number of execution steps (or application of semantic rules or some other reasonable measure of time) of a program \(p\) for an input \(d\) in language \(L\), and \(\text{space}^{L_p}_p(d)\) represents the maximum space usage (e.g. the size of the heap and stack) during the execution of a program \(p\) for an input \(d\) in a language \(L\).” - Definition of the term faithfulness.

“A faithful reversible simulation \(q\) of \(p\) is called hygienic with garbage bound \(g\) if there is no \(q'\) and \(h\) such that \(q'\) is a faithful reversible simulation of \(p\) with \(h\) and \(h(n) = o(g(n))\).” - Definition of the term hygiene.

As an example, we can look at the concatenation of two arrays in a set of arrays, as we will use in the procedure \textbf{union} in Section 3.3.

In a hygienic implementation a single call to \textbf{union} takes constant time and has constant sized garbage. There are, however, other, more naive ways to implement it.

A very naive solution to this would be to simply create a new array and swap the elements of the two arrays into the new one, and if possible delocalling the two old arrays arrays. This would create minimal garbage, as we would only have to store the index of the split, however, it would cost \(O(|n| + |m|)\) time, where \(n\) and \(m\) are the two arrays, which is larger than the irreversible implementations cost of \(O(|m|)\), where \(|m| \leq |n|\). This time complexity is also attainable in a reversible implementation. Thus, this is neither a faithful nor hygienic solution.

By instead using linked list and weighted-union heuristics, as will be explained later, we can achieve the desired time complexity. In this solution, we could choose to store a copies of each of the initial lists as garbage, thus ensuring we would always be able to go back, giving us a faithful implementation. However this would be an excessive amount of garbage, size \(O(|n| + |m|)\), compared to a hygienic implementation, which stores 4 values, or \(O(1)\) size garbage.

The hygienic implementation will be explained in Section 3.3.

### 3 Data Structures

A graph \(G\) is an ordered pair of nodes and edges \((V, E)\), where an edge is a connection between two nodes in \(G\). When working with these algorithms, we assume that no node has an edge connected to itself. We also allow both edges and nodes to store additional information aside from their index and weight.

In graph theory, we generally work with 4 different kinds of graphs - undirected unweighted graphs (Figure 5a), directed unweighted graphs (Figure 5b), undirected weighted graphs (Figure 5c) and directed weighted graphs (Figure 5d). In the following sections, we will look at how these graphs can be represented in Janus, and discuss the pros and cons of each representation.

Whereas nodes in general notations are identified by characters or strings, we will be using integers to identify each node, where \(a = 1\), \(b = 2\), \(c = 3\) and so forth, with the final element being \(n\), where \(n\) is the amount of nodes in the graph.

We will also look at the reversible implementations of a number of relevant data structures used in the algorithms, as well as accompanying functions that needs to be implemented in order to get the graph algorithms to function properly. Throughout
the rest of the report, there will be references to these data structures, often in slightly altered form to fit the algorithms’ requirements.

In both the data structures and algorithms, we use some special values to represent infinity (4294967295), and we generally use -1 to represent a null value, such as when defining the non-existent parent node of the root of a tree, unless otherwise stated.

3.1 Graph Representations

We will use two different commonly used data structures for representing graphs - the adjacency matrix and the adjacency list. A common variation of the adjacency matrix is called an incidence matrix adjacency and will be covered shortly in the following section.

Due to the lack of regular pointers in the current version of Janus, all data structures are implemented using integer arrays. As explained in Section 2.1.6 the term pointer will instead refer to an array index.

3.1.1 Adjacency Matrix

The adjacency matrix describes a graph $G$ with $n$ nodes using an $n \times n$ matrix, where the entry $a_{ij}$ signifies the edge value from node $i$ to node $j$. The adjacency matrix provides representations of both undirected and directed graphs, as well as weighted and unweighted graphs.

This is simply represented with a two-dimensional integer array $a[n][n]$. This implementation is not practical in cases where the nodes or edges of the graph contains extra information, as this would have to be stored in separate arrays, however it is simple to perform lookups in and gives a quick overview of the structure of the graph, making it very usable for simple algorithms and the visual representation of a graph. In Figure 6, the 4 graphs from Figure 5 can be seen as adjacency matrices. The characters are indexes to the arrays and not part of the arrays.

An incidence matrix is a boolean adjacency matrix, meaning it contains only the boolean values 1 and 0. It is used to represent unweighted graphs. A 1 at the entry $a_{ij}$ signifies that there exists an edge from $i$ to $j$. 

Figure 5: The 4 different types of graphs.
The adjacency list describes a graph $G$ with $n$ nodes by having a list for each node in the graph containing its adjacent nodes. The adjacency list provides representations for both undirected and directed graphs. It does not support weighted graphs in its basic form, however by representing the nodes or edges as two-dimensional arrays, we can add extra information such as weight, parent nodes and similar.

Since Janus does not currently support the functionality required for regular linked lists, namely pointers to point the next node in the list, an alternative solution must be thought out using arrays. Two different solutions are presented below. Due to the limitations of the Janus language, both are slightly wasteful when it comes to space in comparison to implementations in more common irreversible languages such as Java or C that supports pointers and customized structures.

The first way is to present the adjacency list as a two-dimensional array $a$ of size $v \times v$, where $v$ represents the maximum amount of outgoing edges from a single node in the graph. Each line in $a_i$ contains a list of all the adjacent nodes, with 0 representing the end of the list. If we consider the worst case scenario, a graph where all but one node has 0 edges, the additional waste space will be of size $n \times (v - 1)$. The best case scenario, where all nodes have edges to all other nodes, there will be no waste space.

The second way is to represent an adjacency list in two arrays, one array $a$ containing all the edges, and another array $r$ of size $n$ containing the starting point of each list, such that the first adjacent node of a node $i$ is stored in the array from $a[r[i]], a[r[i + 1]], a[r[i + 2]], ..., a[r[i + |r| - 1]]$.

This will cause additional waste space of size $n$ in all cases, however, seeing as

---

(a) Undirected unweighted graph.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Directed unweighted graph.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(c) Undirected weighted graph.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

(d) Directed weighted graph.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6: The 4 graphs from Figure 5 represented as adjacency matrices.

3.1.2 Adjacency List

The adjacency list describes a graph $G$ with $n$ nodes by having a list for each node in the graph containing its adjacent nodes. The adjacency list provides representations for both undirected and directed graphs. It does not support weighted graphs in its basic form, however by representing the nodes or edges as two-dimensional arrays, we can add extra information such as weight, parent nodes and similar.

Since Janus does not currently support the functionality required for regular linked lists, namely pointers to point the next node in the list, an alternative solution must be thought out using arrays. Two different solutions are presented below. Due to the limitations of the Janus language, both are slightly wasteful when it comes to space in comparison to implementations in more common irreversible languages such as Java or C that supports pointers and customized structures.

The first way is to present the adjacency list as a two-dimensional array $a$ of size $v \times v$, where $v$ represents the maximum amount of outgoing edges from a single node in the graph. Each line in $a_i$ contains a list of all the adjacent nodes, with 0 representing the end of the list. If we consider the worst case scenario, a graph where all but one node has 0 edges, the additional waste space will be of size $n \times (v - 1)$. The best case scenario, where all nodes have edges to all other nodes, there will be no waste space.

The second way is to represent an adjacency list in two arrays, one array $a$ containing all the edges, and another array $r$ of size $n$ containing the starting point of each list, such that the first adjacent node of a node $i$ is stored in the array from $a[r[i]], a[r[i + 1]], a[r[i + 2]], ..., a[r[i + |r| - 1]]$.

This will cause additional waste space of size $n$ in all cases, however, seeing as
<table>
<thead>
<tr>
<th>Nodes</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index values</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Additional node info</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Array $r$, node information.

<table>
<thead>
<tr>
<th>Indexes</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacent nodes</td>
<td>b</td>
<td>d</td>
<td>a</td>
<td>e</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>e</td>
<td>b</td>
<td>d</td>
</tr>
</tbody>
</table>

| Additional edge info | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(b) Array $a$, edge information.

Figure 7: The undirected unweighted graph from Figure 5, represented as an adjacency list. The additional node and edge info is optional and there can be several lines of it.

A regular implementation of an adjacency list would most likely contain some sort of pointers from each node to their adjacent nodes, the additional waste space in comparison to a regular implementation is effectively none.

Figure 8: Example of an unweighted, directed graph represented as an adjacency list

The effectiveness of each solution depends on how evenly distributed edges are amongst the nodes, with the first solution being preferable if all the nodes have an equal amount of outgoing edges. However, in general cases, the second way will be preferable, and I will use this for most of the work in this project. In Figure 7 is a visual example of an adjacency list using the second implementation, and in Figure 8 is the same graph as implemented in Janus. The code also shows how extra node and edge information is stored by using a two-dimensional array. In the code, each node has room for 2 additional information values, and each edge has room for 1.

```
int G[10][2] = {{2}, {4},
               {1}, {5},
               {4},
               {1}, {3}, {5},
               {2}, {4}}

int nodes[5][3] = {0, 2, 4, 5, 8}
```

Figure 9: Shortened version of getlist.
When creating the reversible algorithms, we will often have to find the current list of adjacent nodes. In [9] we see a procedure that copies a subsection of a list to a new array. We can use this to extract the adjacent nodes of a single node by providing the index value of the node as `start`, the index value of the next node or, in case of getting the list of the final node, the size of the edge list as `end`, as well as the adjacency list and the output array to store the adjacent nodes. This version only supports single arrays, but can easily be modified into extracting the desired information from a two-dimensional array.

- `getlist(int adjacent[], int start, int end, int res[])` copies the adjacent nodes of a node `i` from the adjacency list `adjacent[]`. Takes an integer array consisting of all edges in an adjacency list, an index to the first adjacent node of `i` in the adjacency list, an index to the last adjacent node of `i` in the adjacency list and a return array of at least size `end – start`. Returns all adjacent nodes of `i` stored in `res[]`.

`getlist` produces no garbage and has a running time the size of `end – start`, or in our case, the amount of nodes adjacent to the node `i`.

### 3.2 FIFO Queue

A "First In, First Out"-queue is used for managing a standard queue, where the item added to the queue the longest time ago gets served first. Its size, `n`, determines how many items can be queued at once. When an item is added to the queue, it is added to the tail of the queue, and when an item is removed from the queue, it is removed from the head of the queue, as illustrated in Figure 10. The FIFO queue is used in Breadth-First Search, Section 4.1.

We implement this by using an array `a[n]` with all items initiated to 0 and two pointers initialized to 0, using the value 0 to represent an empty spot in the queue. The value 0 can be used to represent an empty spot, since we will be using it only to store nodes with a minimum value of 1 (ie. a). This allows us to avoid storing the size of the queue, instead using the fact that we know that we can never dequeue an empty item to avoid underflow. Each time an object is removed from the start of the queue, the pointer is incremented, thus keeping track of the current position in the queue. The adding and removal of items are done by doing a swap between the item and an empty spot (0).

The implementation features 2 procedures:

- **enqueue** enqueues an item in a queue. The implementation of `enqueue` can be seen in Figure 11. Using a queue in form of an array and a tail pointer, it adds a given item `item` to the tail of the queue if there is room. If there is no room left or the item added is empty (0 in our case), an error is thrown.

- **dequeue** dequeues the first item in a queue and returns it. Using queue represented as an array and the current head pointer, it removes an item from the queue and stores it in an empty (0) placeholder. If the queue is empty or the input item is not empty, an error will be thrown.

As can be seen in Figure 10, the head of the queue is incremented as the items are extracted from the queue. To ensure that the queue can always hold up to `n` elements,
the pointers wrap around to the first element of the array once the end of the array has been reached. This gives a dynamic queue that can enqueue and dequeue items endlessly, as long as underflow and overflow is prevented.

3.2.1 Garbage and Time Complexity

`enqueue` and `dequeue` are effectively garbage free, leaving behind only the altered state of the queue. `enqueue` leaves an empty value in the item’s place, however this cannot really be considered garbage.

`enqueue` and `dequeue` both run in constant time, which leaves us with faithful, hygienic implementations of both procedures used in the FIFO queue.

3.3 Disjoint Set

A disjoint set is a collection \( S = S_1, S_2, \ldots, S_k \) of disjoint dynamic sets, each set identified by a representative member of the set. Each set is created with a single element, and two sets in the collection can then be combined to create a larger set. This combination process is called `union` and works as shown in Figure 12. The disjoint set is used in Kruskal’s algorithm, Section 4.3.

This is the first of the discussed data structures that is complex to implement in Janus. A standard irreversible implementation is made using several linked lists containing each set, and this will be our basis for implementing it in Janus as well.
procedure enqueue(int queue[], int tailpointer, int item)

    // Error checks
    if item = 0 then
        error("Item added to queue is empty (0).")
    fi
    item = 0
    if queue[tailpointer] != 0 then
        error("Queue is full. Cannot enqueue item.")
    fi

    // Adds the item to the queue spot and empties the item
    local int tmp = item
    queue[tailpointer] += tmp
    item -= tmp
    delocal int tmp = queue[tailpointer]

    // Goes to next queue spot
    if tailpointer + 1 != size(queue) then
        tailpointer += 1
    else
        tailpointer -= size(queue) - 1
    fi

Figure 11: Enqueue procedure. dequeue is similar, but switches the element from the queue to the item instead.

<table>
<thead>
<tr>
<th>Combined sets</th>
<th>Disjoint sets in collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial sets</td>
<td>{a}</td>
</tr>
<tr>
<td>{b, e}</td>
<td>{a}</td>
</tr>
<tr>
<td>{a, d}</td>
<td>{a, d}</td>
</tr>
<tr>
<td>{a, b}</td>
<td>{a, d, b, e}</td>
</tr>
<tr>
<td>{a, c}</td>
<td>{a, d, b, e, c}</td>
</tr>
</tbody>
</table>

Figure 12: The combination of 5 disjoint sets in a collection into one set.

The reversible implementation is made using a two-dimensional array, see Figure 13, storing the set information, consisting of the head, length and tail of each linked list, and a corresponding array storing the node information for each node in the linked lists, consisting of a pointer to the next node in the list and a pointer to the first element in the list, the same as the set representative. It should be noted that the number of elements cannot grow larger than a set maximum $S_{max}$, used to determine the size of the set and node arrays.

Disjoint sets supports the procedures makeset, union and findset:

- makeset is used to create a set in the collection consisting of a single node with index $x$, which in our case can be used to access the node information of an input graph by letting it have the same index in the collection as it has in the graph node array. The set is then updated to point have both head pointer and tail pointer set to $x$ and its length to 1. The node is then updated to have its next node pointer to -1 and its first node pointer to $x$. 

<table>
<thead>
<tr>
<th>Sets (Linked lists)</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Length</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Tail</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

(a) Set information.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>First</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

(b) Node information.

Figure 13: Representation of a collection of sets before performing union.

- **findset** determines which set a node is a member of, and is a simple lookup in the array at the spot corresponding to the node, node[x][1].

- **union** combines two sets $S_i$ and $S_j$ in a collection $C$ into one by concatenating the two. Concatenating $S_j$ onto $S_i$ is done by changing the representative of each of the nodes in $S_j$ to the representative of $S_i$, setting the tail node $S_i$ to point to the head node of $S_j$ and changing the tail pointer of $S_i$ to the tail pointer of $S_j$. Once this is done, $S_j$ is removed from the collection.

Both makeset and findset runs in constant time.

This leaves us with union, which is where most of the actual work takes place. The implementation of union can be seen written in pseudocode in Figure [14]. This uses a helper function called searchnodes which traverses a set of nodes, changing the set membership of all nodes to another set. It does so by taking as input a node in the list that should be changed (initially the head of the list), its current set membership value, its new membership value and the disjoint sets two arrays. This helper function runs in $\Theta(s)$, where $s$ is the number of nodes in the set, and also sets the time complexity for a call to union. In a worst case scenario, a series of calls to union on a single set can in the worst case scenario have a running time of $O(n)$. Using a weighted-union heuristic, where we always pick the shortest of the two lists to run through searchnodes, we can lower the time complexity to $O(m + n \log n)$.

As described in CLRS[3, p. 566] we can lower the time complexity of sequences of union on a collection by using a weighted-union heuristic by storing the length of each set and using this information to add the smallest set to the largest each time union is called, and the Janus implementation has been implemented as such.

Since all operations in union other than searchnodes are constant, the upper bound of the procedure is determined by the helper function.

### 3.3.1 Garbage and Time Complexity

We can assume that in irreversible implementations, after each of the three operations, we have the following information stored:
procedure union(x, y, sets, nodes, garbage, counter)
    if length of set y = 0 || length of set x = 0 then
        error("Trying to perform union with an empty set")
    fi
    if length of set y = 0 || length of set x = 0
    // Calculate which set is added to the other
    local int z = 0
    if length of set x >= length of set y then
        add y to member garbage (now referred to as mset)
        z += x
    else
        add x to member garbage (now referred to as mset)
        z += y
    fi
    // Store garbage
    add tail of set z to tail garbage
    add head of set mset to head garbage
    add length of set mset to length garbage
    // Set the tail of the first set to point to the head of the second set
    zset.tag.next += yset.head + 1
    // Set the tail of the first set to the tail of the second set
    zset.tag -= current tail garbage
    zset.tag += mset.tag
    // Set all values in the second set to 0
    mset.head -= current head garbage
    mset.tag -= zset.tag
    mset.length -= current length garbage
    call searchnodes(current head garbage, mset,
        current zset, nodes, sets)
    // Add the length of the second set to the length of the first set
    zset.length += mset.length
    if mset = x then
        z -= y
    else
        z -= x
    fi
    mset = x
    delocal int z = 0
    counter += 1

Figure 14: Pseudocode for the union procedure, see Appendix C.4 for actual implementation.
(a) The two sets before the union procedure, of length 3 and 2 accordingly.  
(b) The combined set, now with length 5 (3+2).

Figure 15: The union procedure performed on two sets. The set representative is stored in each node, as indicated by the second line in each array.

- **Node information:** Pointer to next node and first node in the list for each node.

- **Set information:** Head, tail and length of each set.

- **Overall information:** List of pointers to the sets in the collection.

We see that both the set information and overall information changes size according to the amount of sets in the collection at any point. Although it is possible to create dynamic arrays in Janus to store the set information and overall information, the copying from one array to another would add to the running time.

When calling `union` we will get some output garbage, as performing `union` on two sets irreversibly removes data. In Figure 15, we can see how we lose the variables `Tail_1` and `Head_2` when performing union. As mentioned, we also overwrite the set representative in the nodes of the second set, and we are unable to split the length back into their separate lengths without knowing one of the values. We need to store these 4 variables’ information so that we can restore it accordingly when performing the reverse calculation, giving us a hygienic garbage bound of $O(1)$ for each call to `union`. None of the other two procedures add any garbage.

Now, we look at the implementation in Janus. As none of the other procedures require additional garbage, we only need to calculate the garbage output of `union` as seen in Figure 14. A single call to union saves the representative of the shortest list in either line 9 or 12. The head and length of the shortest set along with the tail of the longest set is added to the garbage in lines 17–19. No further garbage is added, giving us a garbage of size 4, or $O(1)$.

The reversible procedure maintains the same time complexity as an irreversible version; $O(m + n \log n)$, where $n$ is the number of `makeset` operations and $m$ is the total number of `makeset`, `union` and `findset` operations.

This leaves us with a faithful, hygienic implementation of the disjoint set data structure, with as low output and intermediate garbage as possible without sacrificing speed.

### 3.4 Binary Heap

A binary heap is a tree structure contained in an array. Although there might be elements filling out the entire heap, only the elements contained within $A[1..heapsize]$
is considered part of the heap, where *heapsize* is a counter maintained by the heap operations. An example of this can be seen in Figure 16.

![Figure 16: A minimum heap with heapsize = 5. Note that the array representation in Figure 16b contains elements that are not part of the heap and are marked with grey.](image)

There are two kinds of heaps; minimum heaps and maximum heaps. For minimum heaps, the value of each node is equal to or larger than its parent node’s value, with the root having no parent node. Maximum heaps have an opposite element relation, making child nodes equal to or smaller than their parents. We will be focusing on minimum heaps, as the implementation will be used for a minimum priority queue. A minimum heap supports the following procedures:

1. **start of minheapify with** $i = 1$.
2. **swap, left child < right child <** $i$.
3. **swap, left child < right child <** $i$.

![Figure 17: minheapify performed on root node.](image)

- **left**, **right** and **parent** finds the left child node, right child node or parent node of a node $i$ accordingly.

- **minheapify** swaps the elements of a subtree of a heap to ensure that it upholds the minimum heap property, by recursively switching the elements of a node starting at index $i$ with the smallest of its two child nodes until both its child notes are larger. It is assumed that the two subtrees rooted at the left and right child already uphold the minimum heap property. For an example of how minheapify works, see Figure 17.

- **buildminheap** creates a heap that upholds the minimum heap property by calling minheapify on each element in an unordered heap from the bottom up. As we will not be needing this functionality, it has not been implemented.

The implementation of **minheapify** can be seen in Figure 18.
procedure minheapify(int A[], int heapsize, int i,  
                      int heapgarbage[], int garbagecounter)
local int left = 0, int right = 0, int min = 0
call left (i, left)
call right (i, right)

// Find the smallest note between i and the children of i
min += minimum[A[i], A[right], A[left]]

// Save it in the garbage
heapgarbage[garbagecounter] += min
garbagecounter += 1

// If either of the children is smaller, switch i with the child and
// call minheapify recursively
if min != i then
call minheapify(A, heapsize, min, heapgarbage, garbagecounter)
    if left = heapgarbage[garbagecounter-1] then
        min -= left
    else
        min -= right
    fi
    left = heapgarbage[garbagecounter-1]
fi
min = 0

// Returned from recursion, set garbagecounter back
garbagecounter -= 1

// Set the current min to 0
if i = heapgarbage[garbagecounter] then
    min -= i
fi
i = heapgarbage[garbagecounter]

uncall right (i, right)
uncall left (i, left)
delocal int left = 0, int right = 0, int min = 0

Figure 18: Implementation of minheapify.
3.4.1 Garbage and Time complexity

The implementation of *minheapify* uses recursion to travel down through the heap, each time saving the minimum index of the three nodes. It will only travel at most the distance from the initial node $i$ to the bottom of the tree, giving us a garbage size of $O(\log n)$. We also have a single counter value used as an index for the garbage, which is of constant size.

Since our heap and heap procedures maintain the same structure as in CLRS [3, p.151-156], we can conclude that it has the same time complexity, $O(\log n)$. The only difference in the implementation are operations of constant time, such as saving garbage, which does not add to the time complexity.

Each of the functions **left**, **right** and **parent** are simple lookup functions, using constant time and no garbage.

This gives us a faithful, but not hygienic implementation. This is due to the garbage in *minheapify*, as it should be possible to do the recursive call and save only the initial index $i$ and the final placement of the node, as explained further in the following section. This would give a garbage size of $O(1)$, containing the index of the final position of the node.

3.4.2 Garbage reduction

It should be possible to reduce the garbage size even further to a constant size. As long as we know the initial index $i$, which is set back to its initial value before exiting the procedure, as well as the final index of the node, we know exactly which route has been taken through the tree.

Knowing that all left nodes have an even index value and all right nodes have an uneven index value, we should be able to compare this to the current minimum value, enabling us to recreate the path back, thus leaving us with only the garbage of the final position of the node.

This has not been implemented currently, but should be explored further.

3.5 Minimum Priority Queue

A minimum priority queue is a queue which is always ordered in a way such that the first element of the queue is the minimum element. We implement a minimum priority queue that uses a minimum heap, adding four new procedures to insert new elements, show and extract the minimum element and and change the node values, thus providing the full functionality needed. The minimum priority queue is used in the algorithms of Prim, Section 4.4, and Dijkstra, Section 4.5.

A minimum priority implements four extra functions to use on a minimum heap:

- **decreasekey** decreases the value of a node in the heap to a given key value. If the key given is larger than the current key of the node, an error is returned. If not, the node is continuously swapped with its current parent node until the key of its current parent node is smaller the node’s key.

- **insert** takes a node with a key value and inserts it into a minimum heap, if there is enough room in the heap. This is done by incrementing heapsize by 1, letting
the final item in the heap be the newly inserted item, then calling \texttt{decreasekey}
on the node using a given key parameter.

- \texttt{minimum} copies the content of the minimum key to a container.

- \texttt{extractmin} extracts the minimum node of a minimum priority queue, thenupdates the heap to maintain the minimum heap property. This is done byswapping last node of the heap with the first node of the heap, after storing itscontent in a container, then reducing the heap size and performing \texttt{minheapify}on the heap from the first element. See Figure 19.

3.5.1 Garbage and Time Complexity

We start by looking at the implementation of \texttt{decreasekey}, Figure 20. In the loop, ittraverses through the levels of the heap starting at the index \(i\), swapping the currentnode with its parent node, until it either reaches the root of the heap, worst case, ora parent node smaller than itself. Each time, it saves the index of the current node to the garbage. This means that both its garbage and time complexity are of size \(O(\log n)\).

When we look at the other three operations, we see that \texttt{minimum} uses a simplelookup, thus using constant time and no garbage, the only non-constant operation in\texttt{extractmin} is when it calls \texttt{minheapify}, giving it the same time and garbagecomplexity, and the only non-constant operation in \texttt{insert} is its call to \texttt{decreasekey},meaning they share garbage and time complexity.

This gives us a faithful, but not hygienic implementation. Using the same argumentas with \texttt{minheapify}, we see that we should be able to store only the final position ofthe node, meaning the optimal garbage bound is \(O(1)\).

3.5.2 Garbage reduction

As with \texttt{minheapify}, it should be enough to simply store the final position of thenode, as we know the starting index of the node.

This one is trickier, as we move up in the tree instead of down in the tree, usinga temporary value to store the current \(i\) index in order to remember which node wecome from. A good first step would be looking into a way of figuring out which subtreean index value was part of, however, this has not been solved during this project.
### graph algorithms

The main work of this bachelor's project has lied in converting irreversible algorithms to reversible algorithms while maintaining functionality and time complexity, as well as estimating and attaining an acceptable level for (additional) garbage data. All algorithms have been implemented in the Janus language. This section will cover a wide selection of algorithms, covering the following problems in graph theory:

- **Searching a graph:** When looking for a path from a source to a target node, we create a search tree, or a forest containing several search trees. A search tree consists of a selection of nodes that are connected to a root node. Each node except the root node has a pointer to its parent node. This gives a quick way to search for nodes, as we can simply start from the target node, moving up through the tree. If the node is not connected to the root node, there exists no
path from the node to the root. This problem is solved in two different ways in Section 4.1 and 4.2 and both algorithms can be used on unweighted graphs that can be either directed or undirected.

- **Minimum spanning trees**: A minimum spanning tree is a tree that covers all nodes in a weighted graph with the smallest possible edge weight. In Section 4.3 and 4.4 we cover two popular algorithms used to create a minimum spanning tree. Both algorithms can use all four kinds of graphs.

- **Single-source shortest paths**: This solves the problem of finding a shortest path to every node in a graph from a given source node, based on a weighted graph. This is covered in Section 4.5. This works on all four kinds of graphs, thus expanding on the graph search algorithms.

4.1 Breadth-First Search

4.1.1 Description

Breadth-First Search is used to find a shortest path between the source node \( s \) of an unweighted graph \( G \) and each node \( v \in V \). The BFS algorithm produces a breadth-first tree with root \( s \) containing all nodes that are reachable by \( s \), where the path from \( s \) to each \( v \) represents a shortest path.

BFS, the main algorithm, takes as input a graph \( G \) and a source node \( s \) from which it builds a breadth-first tree. It also requires a first-in-first-out (FIFO) queue, initially containing only the source node, where, for each iteration, of a while loop, a node is removed from the queue, and each of its successor nodes without a set predecessor node are added to the queue, setting their predecessor node to the current node. This process is repeated until the queue is emptied. Initially, the source node is the only item in the queue. Once this process has finished, we have a breadth-first tree, which can be built using the predecessor information for each node. Figure 21 shows how a BFS traverses through a graph and the resulting breadth first tree.

![Figure 21](image_url)

(a) Shows the order of the nodes visited, assuming the adjacent nodes of each node is comes in order from lowest (a) to highest (e).

(b) The resulting breadth first tree. The tree is built using the predecessor node information stored in each node.

Figure 21: Illustration of BFS on a unweighted, undirected graph, with node \( d \) as source node.

Using a breadth-first tree, we can print a shortest path using `printpath`, which takes a breadth-first tree \( G \) with a source node \( s \), and a node \( v \) to find the shortest path to. It recursively travels through the tree, using the predecessor nodes to find a path, printing a path from \( s \) to \( v \) as it goes, if such a path exists. As an addition to `printpath`, a modified version has been made which instead returns the path in
an array. If the node \( v \) cannot be reached, a message will print this, or the array will return empty. It should be noted that this implementation is based on the description in CLRS [3, c.22.2] and might differ from other versions of the breadth-first search.

The algorithm uses a FIFO queue, initially enqueueing the root node, marking it in the process and setting its parent node to null. In each iteration, it then dequeues the node at the head of the queue and adds all its adjacent nodes to the queue, marks each of them and adds the node as their parent node. It continues this until there are no more nodes in the queue, even if all of the nodes have not been marked, thus only including nodes reachable from the root in the tree. The parent node information represents the structure of the tree. Figure 21 shows how BFS traverses through a graph and the resulting breadth first tree.

Using a breadth-first tree we can print a shortest path using `printpath` which takes a breadth-first tree \( G \) with a source node \( s \) and a node \( v \) to find the shortest path to. It recursively travels upward through the tree starting at the node \( v \), using the parent nodes to find a path, printing a path from \( s \) to \( v \) as it exits each recursive call, if such a path exists. If the node \( v \) cannot be reached, a message will print this.

The reversible implementation of BFS can be seen in Figure 22 and takes the following arguments:

- **int G[]** and **int nodes[]**: An adjacency list with extra node information used to store each node’s parent. For the implementation in Janus, we use the second version of the adjacency list described in 3.1.2 with the nodes stored in a two-dimensional array. All parent nodes in the array are initiated to 0 before running BFS.

- **int s** The source node. Is represented by an integer from 1 to \( n \), where \( n \) is the number of nodes in the graph.

- **int order[]** An extra array used to keep track of the order which the nodes have been visited.

The Janus implementation uses the FIFO queue from Section 3.2 and follows the general structure of the CLRS implementation, except using the parent node values to decide whether a node has been marked or not. The queue and other variables are initialized in lines 3–7, and the root node is marked with a parent node set to null (-1) and put in the queue in lines 10–12.

Then the loop goes through the queue until it is empty, marking and adding the unmarked adjacent nodes in the lines from 21–33.

The implementation then follows the structure of a regular irreversible implementation, with the exception of the clearing of `store` and `adj` to 0 after each node has been visited by uncalling `getlist` and subtracting the current node from the order array, thus preparing them for the next loop iteration, in lines 36–38.

### 4.1.2 Time complexity and garbage

There are three sections of the code that that require non-constant operations - the two loops. The first loop loops through each node, whereas the second loop loops through the adjacent nodes of the current node. In total, we see that we run through all nodes and edges, giving us the time complexity \( O(V + E) \), the same as an irreversible implementation.
procedure BFS(int G[], int nodes[], int s, int order[])
  // FIFO queue
  local int queue[size(G)] = {0}, int headpointer = 0,
  tailpointer = 0
  // Storage for dequeued node and adjacent nodes, loopcounter
  local int store = 0, int adjstore = 0, int i = 0
  local int adj[size(G)] = {0} // Stores the current adjacent nodes

  // We mark the current node as the source node and enqueue it
  nodes[s−1][1] = 1
  store += s
  call enqueue(queue, tailpointer, store)

  from headpointer = 0 do
    if queue[headpointer] != 0 then
      // We dequeue a node and gets its adjacency list
      call dequeue(queue, headpointer, store)
      call getlist (G, nodes, store−1, adj)
      order[headpointer] += store

      // We check the list of adjacent nodes and enqueue all unvisited nodes
      from i = 0 do
        if adj[i] is empty then
          skip
        else
          if parent of adjacent node = 0 then
            adjstore += adj[i] // store adjacent node
            set parent of adjacent node to dequeued node
            call enqueue(queue, tailpointer, adjstore)
          fi parent of adjacent node = store
          fi adj[i] = 0
          i += 1
        until i = size(adj)
      i -= size(adj)

      // Set the values of adj and store to 0
      uncall getlist (G, nodes, store−1, adj)
      store -= order[headpointer]
      fi queue[headpointer] != 1
    until headpointer = tailpointer // queue empty

  // Delocal local variables
  delocal int adj[size(G)] = {0}
  delocal int store = 0, int adjstore = 0, int i = 0
  delocal int queue[size(G)] = {0}, int headpointer = size(order)−1,
  tailpointer = size(order)−1

Figure 22: Implementation of Breadth-First Search

The only garbage required is the order garbage, i.e. an array of size $V−1$ and a single pointer to the head of the queue. Since we in can use the Bennett trick by copying the content of the nodes array, use this copy to run through the entire loop from line 14-40, copy the parent information to the original nodes array and then uncalling the loop once more, we can eliminate this garbage entirely. While the version using the
Bennett trick is not shown here, as it is identical to the shown version with the loop in another procedure, it has been implemented and can be seen in Appendix F.

Since we have no garbage in the second version of the algorithm, thus giving us a clean implementation, we are left with a hygienic, faithful implementation.

4.2 Depth-First Search

4.2.1 Description

Depth-First Search is used to create a depth-first forest, containing several depth-first trees, from an unweighted graph $G$ containing the nodes $V$. For the first created depth-first tree in a forest contains all nodes reachable from its root. Each new tree in the forest contain all nodes that are reachable from their roots, except for the nodes already present in another tree.

A depth-first tree is created by recursively visiting adjacent nodes of the nodes in a graph, setting a discovery time for each visited node and adding the parent node to each adjacent node, similar to BFS. When reaching a node with no unvisited adjacent nodes, the node is marked with a finishing time, and the algorithm returns to the parent node.

If there are no further nodes to be visited in the current tree, an unvisited node is visited and a new tree is started. Once all nodes has been visited, the breadth-first forest is complete. Figure 23 shows how DFS traverses through a graph and the resulting breadth first tree.

![Figure 23: Illustration of DFS on a unweighted, undirected graph, with node d as the first node. All nodes in the graph are connected to d, so there is only a single resulting tree in the forest.](image)

When compared to BFS, aside from the obvious difference of performing a depth search rather than a breadth search, the biggest difference is that DFS does not take a source node and creates a single tree, instead possibly creating several trees in order to cover all of the nodes in the graph, even if they are not connected or reachable by an initial root. To handle this difference, the DFS is split up in two procedures, DFS which handles the initialization of each tree by going through a list of all the nodes, checking if they are already in a tree, and DFSvisit which creates each individual tree.

The Janus implementation of DFS can be seen in Figure 24 and takes the following arguments:
```plaintext
procedure DFS(int G[], int nodes[])
    local int time = 0, int i = 0

    // We run through each individual node and check if they have been visited
    // (timestamped). If they haven’t we run DFSvisit.
    from i = 0 do
        if nodes[i][1] = 0 then
            call DFSvisit(G, nodes, i, time)
        fi
        nodes[i][2] = time
        i += 1
    until i = size(nodes)

delocal int time = size(nodes) * 2, int i = size(nodes)
```

Figure 24: Implementation of DFS

- **int G[] and int nodes[]**: An adjacency list with extra node information about each node’s two timestamps. For the implementation in Janus, we use the second version of the adjacency list described in Section 3.1.2. All timestamps in the array are initiated to 0 before running DFS.

The layout of the procedure is simple. First, the time is initiated to 0. Then, for each unvisited node (which will initially be all), we run DFSvisit using the current time, incrementing the time by 1 after each iteration.

DFSvisit can be seen in Figure 25 and takes the following arguments:

- **int G[] and int nodes[]**: An adjacency list with an extra array for information about each node’s timestamps. For the implementation in Janus, we use the second version of the adjacency list described in the data structure section. All timestamps in the array are initiated to 0 before running DFS.

- **int u**: The current node being inspected.

- **int time**: The current time.

The core of the implementation is still rather simple, as the code in lines 6–14 and again 31–37 is simply used for extracting the beginning and end of the adjacent nodes of the current node in the adjacency list. In line 2–4, we add the entry time to the current node. Then, in lines 16–25, we recursively call DFSvisit on each unvisited adjacent node. Finally, in lines 27–29, we add the exit time to the current node.

### 4.2.2 Garbage and time complexity

We first look at the time complexity of the algorithm. In DFSvisit, we recursively go through all adjacent nodes of the current node. Since we will only visit each node once, the maximum amount of times we will perform this loop, including each subsequent recursive call, is the amount of edges $E$ we have. In DFS we have a loop that goes through the list of nodes $V$ once. Adding these together gives us a time complexity of $O(E)$, the same as an irreversible implementation. This is backed up by the fact that it follows the same structure as the depth-first search as implemented in CLRS[3, p.604].
procedure DFSvisit(int G[], int nodes[], int u, int time)
   // The time is incremented and the node gets entry–timestamped
   time += 1
   nodes[u][1] += time

   // end contains a pointer to the end of adjacency list
   local int end = 0
   if u = size(nodes) - 1 then
      // If we try to get the last list we stop at the end of the element list
      end += size(G)
   else
      // Or else, we stop where the next list starts
      end += nodes[u + 1][0]
   fi

   // The adjacent nodes are checked and if they have not been visited ,
   // DFSvisit is called with the unvisited node
   local int r = nodes[u][0] // Pointer to the current node
   from r = nodes[u][0] do
      if nodes[G[r]−1][1] = 0 then // The adjacent node’s first time stamp is 0
         call DFSvisit(G, nodes, G[r]−1, time)
      fi
      nodes[G[r]−1][2] = time // The adjacent node’s second time matches the current time
      r += 1
   until r = end
   delocal int r = end

   // The time is incremented and the node gets exit–timestamped
   time += 1
   nodes[u][2] += time

   // end is delocated
   if u = size(nodes) - 1 then
      end -= size(G)
   else
      end -= nodes[u + 1][0]
      fi
   u = size(nodes) − 1
   delocal int end = 0

Figure 25: Implementation of DFSvisit.

As for garbage, interestingly enough, we can see that there is no garbage output, with only the graph with its new time stamps as output. Since obviously, we can achieve no better garbage output than no garbage, we have a clean, hygienic implementation of DFS.

4.3 Kruskal’s Algorithm

4.3.1 Description

Kruskal’s Algorithm finds the minimum spanning tree $A$, in the form of an adjacency matrix, over a weighted graph $G$ with edges $E$ and nodes $V$ which can be both directed and undirected. The algorithm works by continually unifying sets of nodes by looking at their edges and determining whether they are part of the same set or not, adding them together if they are not. This goes in order from the edges with the lowest weight.
to the highest until all nodes are in the tree, thus forming a minimum spanning tree.

We will explore two different reversible implementations of Kruskal’s Algorithm, one very naive and unfaithful, but clean, implementation, and another that follows the structure of the algorithm in CLR [3, p.631], yet has more garbage.

4.3.2 Naive implementation

For this implementation, we use as input the graph and an output graph in form of two adjacency matrices. It can be seen in Figure 26 written in pseudocode. The algorithm uses the following helper functions:

- **findset** which searches the output graph $A$ for a connection between two nodes $x$ and $y$. by recursively checking $A$ for adjacent edges, i.e. an edge that is not 0, by looking at the array inside the two-dimensional array representing $x$, $A[x]$. If it finds an adjacent edge, it calls findset on this new element, unless the adjacent edge is a connection to $y$. In each iteration, a value is added to an array representing the nodes to mark that this node has been visited and checked to ensure that it does not check a node twice.

- **addedge** adds an edge with a weight to the graph $A$. This runs in constant time.

The algorithm itself first creates a list of edges sorted by weight, then runs through these edges one by one, using findset to check if the two nodes connected by an edge is in the same set. If they are not, the edge is added to $A$.

As explained, it is a clean implementation, meaning there is no output garbage. Instead, we look at the time complexity.

There are three non-constant sections of the code: The sorting of the edges, the loop going through each edge and the calls to findset. The sorting can be done using a standard reversible sorting function, and can thus be run in $O(n \log n)$ time. Such sorting functions has been implemented and described by Axelsen and Yokoyama [1], and we simply use one of these implementations.

In the worst case, where the graph is connected in a single line, from smallest edge to largest edge, a single call to findset will take $O(|V|^2)$ time. The best case scenario, where the set is empty, takes $O(|V|)$ time. In the loop of mstkruskal, the main section of the algorithm, we loop through all of the edges, each time calling findset, giving us a worst-case time complexity of $O(|V|^2 \cdot |E|)$. In most cases, however, it should be lower.

This is not a very satisfying time complexity, and it should be taken into consideration in spite of the clean and simple implementation. Now we turn to the recommended implementation using disjoint sets, which has an estimated time complexity of $O(|E| \log |V|)$ in an irreversible implementation.

4.3.3 Faithful implementation

The second implementation follows the structure of CLRS[3, p.631] and can be seen implemented in Figure 27. It uses a disjoint set data structure to keep track of the collection $A$, taking the following arguments:

- **G[] and edgecount** represents the graph $G$ in the form of an adjacency list.
procedure mstkruskal(int G[], int edgecount, int A[])
    // Initialize variables for loop and findset
    local int i = 0, int ret = 0, int visit = 0
    local int visited [size (G)] = {0} // Used to keep track of which nodes have been visited

    Gather all of the edges from G into a list *edges*, each edge (u, v) containing the weight of the edge and pointers to u and v, sort list using.

    from i = 0 do // Go through all edges
        // Findset checks two nodes combined by an edge are in the same subset
        call findset (edges[i][1]−1, edges[i][2]−1, A, visited, visit, ret)
        if ret = 0 then
            // If they aren’t, the two nodes are connected
            uncall findset (edges[i][1]−1, edges[i][2]−1, A, visited, visit, ret)
            call addedge (edges[i][1]−1, edges[i][2]−1, edges[i][0], A)
        else
            uncall findset (edges[i][1]−1, edges[i][2]−1, A, visited, visit, ret)
        fi
        A[edges[i][1]−1][edges[i][2]−1] != 0
        i += 1
    until i = edgecount

    // Delocal values.
    uncall msort (edges, edgecount, p)
    from r = edgecount do
        p[r−1] = r−1
        r -= 1
    until r = 0

    delocal int r = 0
    delocal int p[edgecount] = {0}
    uncall extractedges (G, edges)
    delocal int edges[edgecount][3] = {{0}, {0}, {0}}
    delocal int visited [size (G)] = {0}
    delocal int i = edgecount, int ret = 0, int visit = 0

Figure 26: Naive implementation of Kruskal’s algorithm.

- A[] represents the output graph A in the form of an adjacency matrix.

- nodes[] and set[] represents the disjoint set data structure once the algorithm has been run, containing the output disjoint set information ie. the information contained in the nodes as well as the set information of the final set.

- splitgarbage[] will end up containing the garbage from the disjoint sent procedures.

In the Janus implementation, we initially create a list edges to store a list of all edges in G, each containing an edge index and the weight of the edge. This list is then sorted according to edge weight. Then sets of edges are created, one for each node, stored in the variable sets, in which all set information is initially 0.

Using checksets, we check whether the nodes in the currently lightest edge are in the same subset - if they are not, we add the edge to the A and call union on the two nodes.

Once we have iterated through all of the edges, we swap the set information of the final set to the output set. This reduces the set output garbage from \(O(|V| + |E|)\) to
procedure mstkruskal(int G[], int edgecount, int A[],
    int nodes[], int set[], int splitgarbage[])
// Initialize variables for loop, checksets and split
local int i = 0, int found = 0, int splitcounter = 0,
    int sets[size(G)][3] = {{0}, {0}, {0}}
local int edges[edgecount][3] = {{0}}
Gather all of the edges into the list 'edges' and sort

from i = 0 do // Create the edgset
call makeset(i, nodes, sets)
i += 1
until i = edgecount

i -= edgecount

from i = 0 do // Go through all edges
// checksets checks if two nodes u (edges[i][1]−1) and v (edges[i][2]−1) are in the same subset
    call checksets(u, v, nodes, found)
if found = 0 then
    // If they aren't, the two nodes are connected
    uncall checksets(u, v, nodes, found)
    Add edge between u and v to A
    local int x = u, int y = v
    call union(x,y,
        nodes, sets, splitgarbage, splitcounter)
    delocal int x = u, int y = v
else
    uncall checksets(u, v, nodes, found)
fi A[u][v] != 0
i += 1
until i = edgecount

Swap final set information into set
Unsort and ungather the list 'edges'

delocal int edges[edgecount][3] = {{0}}
delocal int i = edgecount, int found = 0, int splitcounter = size(G) − 1,
    int sets[size(G)][3] = {{0}, {0}, {0}}

Figure 27: Faithful implementation of Kruskal's algorithm

\[ O(|E| + 1), \text{ or } O(|E|). \]

4.3.4 Garbage and time complexity

As the structure follows the irreversible implementation in CLRS \footnote{3} p.631, and the used data structure is implemented faithfully, the time complexity is the same as the irreversible version.

As for garbage, it will grow each time we call the \texttt{union} procedure, which belongs to the disjoint set data structure. We know from earlier that it will always return garbage of constant size. Since we call it \( V - 1 \) times, the garbage output will be of size \( O(|V|) \).
4.4 Prim’s Algorithm

4.4.1 Description

Prim’s algorithm provides another way of finding the minimum spanning tree $A$ over a weighted graph $G$. It uses a minimum priority queue containing the nodes sorted in accordance to a key value. The root node has its key initialized to 0 and all other nodes to infinity, meaning the root note will initially be the first node in the minimum priority queue. In the algorithm, we extract the minimum node of the minimum priority queue until it is empty. Each time a node is extracted, the key values of the node’s adjacent nodes is updated such that, if the key of the current node $u$ is larger than the weight between the nodes $u$ and adjacent node $v$, it inserts the new weight value as the new key value of $v$, thus relaxing the edge.

The reversible implementation of Prim’s algorithm can be found in Figure 28 and takes the following arguments:

- $G$ and nodes represents the graph as an adjacency list.
- $r$ is an index to the root of the tree.
- queue and extractgarbage is for the garbage output of the minimum priority queue. extractgarbage is used each time we extract the minimum element of the queue, as explained in Section 3.5.
- relaxgarbage and relaxcounters is garbage introduced when relaxing due to the call to decreasekey.
- count is used as a counter for the garbage.
- adjacentgarbage is used to store at which point in the queue the current adjacent node is stored. Since we need to check if an adjacent node is still part of the queue, and we later need to access it to change its key value if it is not, we save its queue index to an array. It is primarily used in the procedure isinqueue.

4.4.2 Garbage and time complexity

Due to the isinqueue procedure, which takes $O(|V|)$ time for each call, we have a situation where it adds $O(|V|^2)$ time to the algorithm, giving us a unfaithful implementation. This should be possible to improve by adding an index value to the node that keeps track of the node’s position in the queue, however, due to time pressure this has not been implemented. This would add to the garbage, but keep the implementation faithful, as the procedure would instead be able to take constant time.

As for garbage, it comes from the minimum priority queue. If this can be improved, we can reduce garbage and possibly get a hygienic implementation.

4.5 Dijkstra’s Algorithm

4.5.1 Description

Dijkstra’s Algorithm is used to solve the single-source shortest-paths problem, finding the shortest path from a source node to all its connected nodes in a weighted, possibly
procedure MSTPrim(int G[], int nodes[], int r, int queue[]),
    int extractgarbage [], int decreasegarbage [],
    int decreasecounters [], int count, int adjacentgarbage[])
local int i = 0, int queue size = size(queue)−1, int extractcount = 0
call initialize (nodes, queue, r)

from queue size = size(queue)−1 do
    local int u = 0, int adjsize = 0
call extractmin(queue, queue size, u, extractgarbage, extractcount)

call getadjsize(u, nodes, G, adjsize)
local int adjlist [adjsize] = {0}
call getlist(G, nodes, u−1, adjlist)
local int j = 0

from j = 0 do
    local int k = 0, int inqueue = 0
call isinqueue(inqueue, k, queue size, queue, j,
    adjlist, adjacentgarbage, count)
if inqueue = 1 &&
G[nodes[u−1][0] + j][1] < queue[k−1][1] then
    adjacentgarbage[count][1] <=> nodes[adjlist[j−1][1]]
    nodes[adjlist[j−1][1]] += u
    call decreasekey(queue, adjacentgarbage[count−1][0],
    G[nodes[u−1][0] + j][1],
    decreasegarbage, decreasecounters, count−1)
fi inqueue = 1 && nodes[adjlist[j−1][1]] = u
j += 1

if queue size != 0 then
    from k = queue size + 1 do
    k -= 1
    if queue[k][0] = adjlist[j−1] && inqueue = 1 && k != 0 then
    inqueue -= 1
    fi queue[k][0] = adjlist[j−1] && inqueue = 0 && k != 0
    until k = 0
fi queue size != 0

delocal int k = 0, int inqueue = 0
until j = size(adjlist)
delocal int j = size(adjlist)
uncall getlist(G, nodes, u−1, adjlist)
delocal int adjlist [adjsize] = {0}
uncall getadjsize(u, nodes, G, adjsize)
delocal int u = queue[queue size+1][0], int adjsize = 0
until queue size = 0
delocal int i = 0, int queue size = 0, int extractcount = size(nodes)

Figure 28: Janus implementation of Prim’s algorithm.

directed graph G. Using a minimum priority queue containing the nodes sorted in accordance to a key value, with the root node initialized to 0 and thus the first element and all other nodes to infinity, it goes through the list, extracting the minimum value each time. It then updates the key value of the currently extracted node’s adjacent
nodes in a way such that, if the key of the current node $u$ plus the weight of an edge from $u$ to $v$ is smaller than the current key of $v$, the key of $v$ is updated to contain the value of the key of $u$ added to the weight of the edge from $u$ to $v$.

In other words, we in each iteration we add the node that is currently the closest to the source node and is not in the set already. We can see that both the implementation and general thought behind Dijkstra’s algorithm is similar to that of Prim’s algorithm in the way it uses a minimum priority queue to update nodes’ key values.

The reversible implementation of Dijkstra’s algorithm can be found in Figure 29 as pseudo code and takes the following arguments:

- **G and nodes** represents the graph $G$ as an adjacency list with two added node values representing the node’s parent and its distance from the source node and an added edge value containing the weight of the edge.

- **s** is an index to the source node.

- **queue and extractgarbage** is for the garbage output of the minimum priority queue. extractgarbage is used each time we extract the minimum element of the queue, as explained in Section 3.5.

- **relaxgarbage, relaxcounters and adjacentgarbage** is garbage introduced when relaxing due to the call to decreasekey.

- **count** is used as a counter for the garbage.

The algorithm starts by initializing the queue and nodes by calling the procedure initialize. initialize sets the parent of the source node to null and adds it to the minimum queue with a key value of 0. It then updates the rest of the nodes’ parents to 0 and key values to infinity. From here, it follows the structure of the algorithm as implemented in CLRS[p.658][3]. It loops through the queue until it’s empty, extracting the minimum element of the queue in each iteration. Then the adjacent nodes are found using getlist.

It then goes through the list of adjacent nodes, relaxing each of the edges by comparing the current key value of an adjacent node, representing the shortest distance to the source node found so far, to the weight of the edge added to the distance between the extracted node and the source node, setting the key value of the adjacent node to this last distance and its parent node to the extracted node if it is smaller.

### 4.5.2 Time complexity and garbage

The Janus implementation follows the structure of the implementation in CLRS, and will have the same time complexity, considering the minimum queue it uses has been implemented faithfully.

When it comes to garbage, the only garbage required is the garbage for the minimum priority queue. If the garbage of the minimum priority queue can be reduced, so can the garbage of Dijkstra’s algorithm.

Since the minimum priority queue is not currently hygienic, neither is the implementation of Dijkstra’s algorithm. It is still, however, faithful.
procedure dijkstra(int G[], int nodes[], int s, int queue[],
    int relaxgarbage[], int relaxcounters[], int extractgarbage[],
    int count, int adjacentgarbage[]))
    local int queuesize = size(queue) − 1, int extractcounter = 0
    call initialize (nodes, queue, s)
    from queuesize = size(queue) − 1 do
        local int u = 0, int adjsize = 0
        call extractmin(queue, queuesize, u, nodes,
            extractgarbage, extractcounter)
        call getadjsize(u, nodes, G, adjsize)
        local int adjlist [adjsize] = {0}
        call getlist(G, nodes, u−1, adjlist)
        local int i = 0
        from i = 0 do
            if queuesize != 0 then
                local int weight = G[nodes[u−1][0]+i][1]
                call relax(u, adjlist [i], nodes, weight,
                    relaxgarbage, relaxcounters, queue, queuesize,
                    count, adjacentgarbage)
                delocal int weight = G[nodes[u−1][0]+i][1]
            fi
            queuesize != 0
            i += 1
        until i = size(adjlist)
        delocal int i = size(adjlist)
        uncall getlist(G, nodes, u−1, adjlist)
        delocal int adjlist [adjsize] = {0}
        uncall getadjsize(u, nodes, G, adjsize)
        delocal int u = queue[queuesize+1], int adjsize = 0
        until queuesize = 0
    delocal int queuesize = 0, int extractcounter = size(nodes)

Figure 29: Implementation of Dijkstra’s algorithm

5 Testing

For testing, two tools have been created to help test the implementations thoroughly.

The test programs uses randomly created graphs, each with 50 nodes, created using a
Python program, along with graphs with 0 and 1 nodes. The program can create
randomly generated graphs of both directed and undirected, weighted and unweighted
graphs of any size, printing the resulting graphs to a .txt file. This program is included
in Appendix K.

The second program is a Janus program used to convert an adjacency matrix to an
adjacency list and has been created to ease testing, as well as to provide for an easier
visual representation of graphs in algorithms that use the adjacency list as input. This
has been included in Appendix K.L.

Currently, only a few tests have been implemented due to lack of time.

There will not be any tests of execution time. Instead, the algorithms have their
time complexity explained throughout the report by analysing the operations made.
As the code would run through the Jana interpreter, the tests could suffer from inter-
pretive overhead, thus giving slower and less stable times and if it had been compiled.

6 Implementation

The Janus implementations run on the Jana/Janus interpreter. The interpreter is maintained mainly by Michael Kirkedal Thomsen, receiving frequent updates, and the algorithms function on the version updated on 11/25/14 (Commit 6b3a5c1). There are two versions of the interpreter available; an online interpreter as well as the official interpreter available at Michael Kirkedal Thomsen’s private Git repository.

For getting a feel of how the Janus language works or for testing the given algorithm implementations, I recommend using the online interpreter to avoid having to gain access to the private repository. The online interpreter can be found at [http://topps.diku.dk/pirc/janus-playground/](http://topps.diku.dk/pirc/janus-playground/) along with a number of example programs, and as per all of the algorithms given in this report works as intended in it.

The code from this report has been added to a .zip file and uploaded with the report. It contains two folders, one with the basic implementations plus a few examples of running the code called Implementations, and one with the full test suites called Test suites.

6.1 Running a program in the online interpreter

The online interpreter at [http://topps.diku.dk/pirc/janus-playground/](http://topps.diku.dk/pirc/janus-playground/) seen in Figure 30 contains three areas - A menu at the top, an editor in the middle and the output area in the bottom.

In the top menu, there are several options, the most important for starting in Janus being Run, Invert and Examples.

- **Run** runs the program currently in the editor, returning the output or any errors in the output area.

- **Invert** gives the inverse version of the current program - that is, the program as it is run when run in reverse or how uncalling works for individual functions. This can be useful both for understanding how the reversibility works, but also as a tool for finding proper conditions for **fi** and **until**.

- **Examples** has a list of programs that gives a good insight into the workings and syntax of the Janus language.

To run one of the programs containing the algorithms from this report, simply copy-paste the code of the desired algorithm from one of the folders into the editor and click run. This will run a test suite and return the results in the editor below, as well as all non-local variables, ie. the variables declared in **main**, with their new output values.

6.2 Installing the Janus interpreter on a personal computer

To install the Janus interpreter on a personal computer, it is necessary to obtain the interpreter code from Michael Kirkedal Thomsen’s Git repository. This can be found at [https://github.com/kirkedal](https://github.com/kirkedal). To install the interpreter, follow the steps below:

1. Download the interpreter code from the Git repository.
2. Download and install Cabal (https://www.haskell.org/cabal/).

3. Run the commands `cabal configure`, `cabal build` and `cabal install` in the interpreter directory.

In order to run a Janus program, use the command `jana filepath` in the console window, replacing `filepath` with the path of the file. This will run the file in the Jana interpreter and return the results of a test suite as well as final output values if one of the test suites are run. An example of running a test suite program in the console can be seen in Figure 31.

7 Conclusion

This report has presented faithful, reversible implementations of a variety of common graph algorithms and their data structures, of which most have also been hygienic. This includes hygienic implementations of Breadth-First Search, Depth-First Search and Kruskal’s Algorithm, as well as faithful implementations of Prim’s Algorithm and Dijkstra’s Algorithm, both which uses a heap data structure that cause extra garbage. Although the algorithms cover several different types of algorithmic problems, there are still plenty to study further.

1If you have the Haskell Platform installed already, you should already have Cabal and can skip this step.
It is clear that there is a strong link between the amount of garbage of an algorithm and the complexity of the data structures they use. The more complex data structures, such as the minimum priority queues and disjoint set, both implement procedures that requires erasure of data, and, as such, storage of data in their reversible versions. This means that the erased data must be stored as garbage, as the output cannot be neutralized using the Bennett trick\cite{bennett89}.

It is impossible to create clean versions of all of the algorithms. A significant part of the analysis work has gone into figuring out how much garbage is optimal for each algorithm, and such what should be required of the hygienic implementations. Although not all of the algorithms live up to these requirements, suggestions have been made as to which steps to take to reach them.

### 7.1 Future Work

As described in the introduction, little work has been focused on reversible algorithms in general. This leaves a large gap when it comes to making reversibility easily accessible to the masses. With the two subjects of comparison sorts and graph algorithms at least partly covered, it seems only logical to continue the work on subjects such as search algorithms and dynamic programming.

Although this report have covered the implementation of several graph algorithms, there are still some, that have not been explored fully, in particular the entire problems of All-Pair Shortest-Path and Max Flow. The algorithms left unexplored should be implemented, and hopefully, the work done in this project will help ease the implementations, in particular in regards to the data structures provided.

It would also be worth looking into improving the implementations that do not currently have hygienic implementations.

Finally, due to the complications that have occurred due to the current limitations of the Janus language, it could be worth looking into expanding it further. This could include custom structures and types, which could both ease the implementation of data structures and make finished code easier to read. It could also include implementations of more data types, such as pointers, characters and strings for general use.

### References


Appendices

A  Janus: Adjacency List

A.1  Basic data structure

```c
// Graph with 5 nodes represented as an adjacency list
int G[] = {2, 4, 1, 5, 4, 1, 3, 5, 2, 4};
int nodes[size(G)] = {0, 2, 4, 5, 8};
// Below is the same nodes with an extra field for node information
int nodes2[size(G)] = {{0, 0},
    {2, 0},
    {4, 0},
    {5, 0},
    {8, 0}};
```

A.2  Procedure: getlist v1

```c
/*
 * getlist gets the contents of a single list in the adjacency list.
 */

procedure getlist(int G[], int nodes[], int i, int res[])
    // end contains a pointer to the end of the wanted list
    local int end = 0
    if i = (size(nodes) - 1) then
        // If we try to get the last list we stop at the end of the element list
        end += size(G)
    else
        // Or else, we stop where the next list starts
        end += nodes[i + 1]
    fi
    // All elements of the wanted list are loaded into res
    local int r = nodes[i]
    from r = nodes[i] do
        res[r - nodes[i]] += G[r]
        r += 1
    until r = end
```
A.3 Procedure: getlist v2

```plaintext
/*
* getlist gets the contents of a single list in the adjacency list.
*
* Parameters:
* int G[] A list of adjacency list elements
* int start A pointer to the first element to be extracted
* int end A pointer to the last element to be extracted
* int res[] A list to store the returned elements in
*
* Postcondition:
* A copy of a the items in a list in the adjacency list.
*/

procedure getlist(int G[], int start, int end, int res[])
  // Elements of the wanted list are loaded into res
  local int r = start
  from r = start do
    res[r - start] += G[r]
    r += 1
  until r = end
  delocal int r = end
```

A.4 Procedure: getadjsize

```plaintext
/* getadjsize finds the size of the adjacency list belonging to a single node
*
* int u Index of the node
* int nodes[] Array containing all nodes
* int G[] List of adjacency nodes
* int return Return value (initially 0)
*
* Post condition:
* return contains the amount of nodes adjacent to u
*/

procedure getadjsize(int u, int nodes[], int G[], int return)
```
if u < size(nodes) then
    return += nodes[u][0] - nodes[u-1][0]
else
    return += size(G) - nodes[u-1][0]
fi u < size(nodes)

B Janus: FIFO Queue

B.1 Basic data structure

// Queue of size 3 with pointers initialized to 0
int queue[3]
int headpointer
int tailpointer

B.2 Procedure: enqueue

 procedure enqueue(int queue[], int tailpointer, int item)
    // Error checks (input)
    if item = 0 then
        error("Item added to queue is empty (0).")
    fi item = 0
    if queue[tailpointer] != 0 then
        error("Queue is full. Cannot enqueue item.")
    fi queue[tailpointer] != 0
    // Adds the item to the queue spot and empties the item
    local int tmp = item
    queue[tailpointer] += tmp
    item -= tmp
dealloc int tmp = queue[tailpointer]
```c
// Goes to next queue spot
if tailpointer + 1 != size(queue) then
tailpointer += 1
else
tailpointer -= size(queue) - 1
fi
```

### B.3 Procedure: dequeue

```c
/*
 * dequeue removes an item from a queue.
 *
 * Parameters:
 * int queue[] Queue to add item to.
 * int headpointer Pointer to the first item in the queue
 * int placeholder Empty item placeholder.
 *
 * Postcondition:
 * A queue with the first item removed and the dequeued item.
 *
 * Errors:
 * Placeholder item not empty (0).
 * Queue is empty.
 */

procedure dequeue(int queue[], int headpointer, int placeholder)

    // Error checks (input)
    if placeholder != 0 then
        error("Placeholder not empty (0).")
    fi
    if queue[headpointer] = 0 then
        error("Queue is empty. Tried to remove item from empty queue.")
    fi

    // Loads the queue item into the placeholder and empties the queue spot
    local int tmp = queue[headpointer]
    placeholder += tmp
    queue[headpointer] -= tmp
delocal int tmp = placeholder

    // Goes to next queue spot
    if headpointer + 1 != size(queue) then
        headpointer += 1
    else
        headpointer -= size(queue) - 1
    fi
```
C  Janus: Disjoint Set

C.1  Basic data structure

```c
// Node information (empty). Can contain up to 5 nodes.
int nodes[5][2] = {{0}}
// Set information (empty). Contains the sets expected at output, here 1.
int set[3]
// Garbage output
int garbage[size(nodes)-1][4] = {{0}}
// Intermediate garbage (all set information and garbage counter)
local int sets[size(nodes)][3] = {{0}}, int counter = 0

// Initialize the a full collection
local int i = 0
from i = 0 do
    call makeset(i, nodes, sets)
    i += 1
until i = size(nodes)
delocal int i = size(nodes)
```

C.2  Procedure: makeset

```c
/*
* makeset creates a node for a single element and adds it to a new set.
*
* Parameters:
* int x Numbered element to be added - Should NOT be in the list currently.
* int nodes[] A list of all current nodes added to the data structure.
* int sets[] A list of all current sets added to the data structure.
*
* Postcondition:
* A disjoint set data structure with the added element
*/
procedure makeset(int x, int nodes[], int sets[])
    nodes[x][0] -= 1 // Next element, initially points to no element
    nodes[x][1] += x // Part of its own set
    sets[x][0] += x // Head
    sets[x][1] += x // Tail
    sets[x][2] += 1 // Length
```
C.3 Procedure: checkset

```c
/*
 * checksets checks if two nodes are in the same set.
 */

procedure checksets(int x, int y, int nodes[], int found)
```

C.4 Procedure: union

```c
/*
 * union combines two sets.
 */

procedure union(int x, int y, int nodes[], int sets[],
```

45
C.5 Procedure: searchnodes (Helper function)

/*
 * searchnodes traverses through a linked list and changes the set membership
 * Parameters:
 *  int i Current node
 *  int newset The new set
 *  int oldset The current set
 *  int nodes[] A list of all current nodes added to the data structure.
 *  int sets[] A list of all current sets added to the data structure.
 * Postcondition:
 *  A disjoint set data structure with the added element
 */
procedure searchnodes(int i, int newset, int oldset, int nodes[], int sets[])

nodes[i][1] -= oldset
nodes[i][1] += newset
if nodes[i][0] != -1 then
call searchnodes(nodes[i][0], newset, oldset, nodes, sets)
fi nodes[i][0] != -1

D  Janus: Binary Heap

D.1 Basic data structure

// Initial minimum heap.
int A[] = {0, 1, 2, 3, 7, 4, 8, 9}

int heapsize = size(A)-1
// Heap garbage for each time we perform minheapify (used in extractmin).
// The size can differ in accordance to the heap’s size.
int heapgarbage[(heapsize+1)/2]

// Garbage for each time we perform decreasekey (used in insert).
int insertgarbage[(heapsize+1)/2]

D.2 Procedures: parent, left and right

/*
 * parent finds the index of the parent of the node at index i
 * Parameters:
 * int i       Node index
 * int parent  Container for the index of the parent node. Must be 0.
 * Postcondition:
 * parent contains an index to the parent node of node #i
 */
procedure parent(int i, int parent)
    parent += i / 2

/*
 * left finds the index of the left child of the node at index i
 * Parameters:
 * int i       Node index
 * int left    Container for the index of the child node. Must be 0.
 * Postcondition:
 * left contains an index to the left child node of node #i
 */
procedure left(int i, int left)
left += 2 * i

/*
 * right finds the index of the right child of the node at index i
 * Parameters:
 * int i Node index
 * int right Container for the index of the child node. Must be 0.
 * Postcondition:
 * right contains an index to the right child node of node #i
 */

procedure right(int i, int right)
    right += 2 * i + 1

D.3 Procedure: minheapify

/*
 * minheapify ensure the minimum heap property of a tree, where both
 * subtrees of
 * the node at index i has the minimum heap property, yet the node at index
 * i
 * might be larger
 * Parameters:
 * int A[] A minimum heap array with an inconsistency at index i
 * int heapsize The heaps current size
 * int i An index for a node in the heap
 * int heapgarbage[] An array used for garbage
 * Postcondition:
 * A minimum heap which has no inconsistency at index i and its two subtrees
 */

procedure minheapify(int A[], int heapsize, int i,
                     int heapgarbage[], int garbagecounter)
    local int left = 0, int right = 0, int min = 0
    call left(i, left)
    call right(i, right)

    if left <= heapsize && A[left] < A[i] then
        min += left
        if right <= heapsize && A[right] < A[min] then
            min -= left
            min += right
        fi
    else
        min += i
        if right <= heapsize && A[right] < A[min] then
            min += right
        fi
    fi
E Janus: Minimum Priority Queue

Note: The minimum priority queue uses the same data structure as the binary heap.

E.1 Procedure: decreasekey

```c
/*
 * decreasekey decreases the key of a node in the queue
 *
 * Parameters:
 * int A[] A minimum heap array with an inconsistency at index i
 * int i Index to the node which key should be decreased
 * int key The new key value of the item
 * int garbage[] An array used for garbage
 * int garbagecounter A counter used when traversing through the heap
 *
 * Postcondition:
 * The minimum item of the minimum priority queue has been transferred to min
 */
```
### E.2 Procedure: insert

/*
 * insert insert a new item into the minimum priority queue
 *
 * Parameters:
 * int A[] A minimum heap array with an inconsistency at index i
 * int heapsize The heap's current size
 * int key The key value of the new item
 * int garbage[] An array used for garbage
 * int garbagecounter[] A counter used when traversing through the heap
 *
 * Postcondition:
 * The item has been inserted in the right place in the queue
 */

procedure insert(int A[], int heapsize, int key, int garbage[], int garbagecounter)
E.3 Procedure: minimum and extractminimum

```c
/*
* minimum finds the index of the right child of the node at index i
*
* Parameters:
* int i Node index
* int right Container for the index of the child node. Must be 0.
*
* Postcondition:
* right contains an index to the right child node of node #i
*/
procedure minimum(int A[], int min)
    min += A[1]

/*
* extractmin extracts the minimum item in a minimum priority queue, then
* rearranges the underlying heap to ensure the minimum heap property.
*
* Parameters:
* int A[] A minimum heap array with an inconsistency at index i
* int heapsize The heap's current size
* int min A placeholder for the minimum item in the heap
* int heapgarbage[] An array used for garbage
*
* Postcondition:
* The minimum item of the minimum priority queue has been transferred to
* min
*/
procedure extractmin(int A[], int heapsize, int min,
    int heapgarbage[])
    if heapsize < 1 then
        error("Heap underflow")
    fi
    heapsize < 1

    // Extract minimum element
    min += A[1]
    A[1] -= min
    A[1] += 4294967295

    // Swap minimum element with the final element of the heap

    // Decrease size of heap
    heapsize -= 1

    // Call minheapify to preserve minimum heap property
    local int garbagecounter = 0
    call minheapify(A, heapsize, 1, heapgarbage, garbagecounter)
    delocal int garbagecounter = 0
```
/ * minheapify ensure the minimum heap property of a tree, where both
 * subtrees of
 * the node at index i has the minimum heap property, yet the node at index
 * i
 * might be larger
 *
 * Parameters:
 * int A[] A minimum heap array with an inconsistency at index i
 * int heapsize The heaps current size
 * int i An index for a node in the heap
 * int heapgarbage[] An array used for garbage
 * int garbagecounter A counter signifying the current level of recursion i
 * the heap
 *
 * Postcondition:
 * A minimum heap which has no inconsistency at index i and its two subtrees
 */

procedure minheapify(int A[], int heapsize, int i,
                      int heapgarbage[], int garbagecounter)

local int left = 0,
     right = 0,
     min = 0

// Find the smallest note between i and the children of i
if left <= heapsize && A[left] < A[i] then
  min += left
  if right <= heapsize && A[right] < A[min] then
    min -= left
    min += right
  else
    min += i
  fi
else
  min += i
  if right <= heapsize && A[right] < A[min] then
    min -= i
    min += right
  fi
fi

// Save it in the garbage
heapgarbage[garbagecounter] += min
garbagecounter += 1

// If either of the children is smaller, switch i with the child and
// call minheapify recursively
if min != i then
  call minheapify(A, heapsize, min, heapgarbage, garbagecounter)
  if left = heapgarbage[garbagecounter-1] then
F Janus: Breadth-First Search

F.1 BFS v1

```
/*
 * BFS searches a graph from a single source node creating a breadth-first tree.
 */
procedure BFS(int G[], int nodes[], int s, int headpointer, int order[])
  // FIFO queue
  local int queue[size(G)] = {0}
  local int tailpointer = headpointer
  // Storage for dequeued node and adjacent nodes, loopcounter
  local int store = 0, int adjstore = 0, int i = 0
  local int adj[size(G)] = {0}  // Stores the current adjacent nodes

  // We mark the current node as the source node and enqueue it
  nodes[s-1][1] -= 1
  store += s
  call enqueue(queue, tailpointer, store)

  // We dequeue a node and gets its adjacency list
```

call dequeue(queue, headpointer, store)
call getlist(G, nodes, store-1, adj)
order[headpointer] += store

// We check the list and enqueue all unvisited nodes
from i = 0 do
  if adj[i] = 0 then
    skip
  else
    if nodes[adj[i]-1][1] = 0 then
      adjstore += adj[i]
      nodes[adj[i]-1][1] += store
      call enqueue(queue, tailpointer, adjstore)
    fi
    nodes[adj[i]-1][1] = store
  fi
  adj[i] = 0
  i += 1
until i = size(G)
i -= size(G)

// Remove the content from adj and store
uncall getlist(G, nodes, store-1, adj)
store -= order[headpointer]
fi
queue[headpointer] != 1
until headpointer = tailpointer

// Delocal local variables

delocal int adj[size(G)] = {0}
delocal int store = 0, int adjstore = 0, int i = 0
delocal int tailpointer = headpointer
delocal int queue[size(G)] = {0}

F.2  BFS v2

procedure copy(int inputarray[], int outputarray[], int index)
  local int i = 0
  from i = 0 do
    outputarray[i][index] += inputarray[i][index]
    i += 1
  until i = size(inputarray)
delocal int i = size(inputarray)

procedure BFS2(int G[], int nodes[], int headpointer, int tailpointer,
  int queue[], int store, int adj[], int adjstore,
  int order[], int i)
  from headpointer = 0 do
    if queue[headpointer] != 0 then
      // We dequeue a node and gets its adjacency list
      call dequeue(queue, headpointer, store)
call getlist(G, nodes, store-1, adj)
order[headpointer] += store

// We check the list and enqueue all unvisited nodes
from i = 0 do
    if adj[i] = 0 then
        skip
    else
        if nodes[adj[i]-1][1] = 0 then
            adjstore += adj[i]
            nodes[adj[i]-1][1] += store
            call enqueue(queue, tailpointer, adjstore)
            fi nodes[adj[i]-1][1] = store
        fi adj[i] = 0
    i += 1
until i = size(adj)
i -= size(adj)

// Remove the content from adj and store
uncall getlist(G, nodes, store-1, adj)
store -= order[headpointer]
fi queue[headpointer] != 1
until headpointer = tailpointer

/*
 * BFS searches a graph from a single source node creating a breadth-first tree.
 *
 * Parameters:
 * int G[] A list of adjacency list elements
 * int nodes[] A list of pointers to the start of each individual list
 * int s The source node
 * int headpointer A pointer to the head of a queue
 * int order[] An array with the order taken in the BFS
 */
procedure BFS(int G[], int nodes[], int s)
    local int order[size(nodes)-1] = {0}, int headpointer = 0
    // FIFO queue
local int queue[size(G)] = {0}, int tailpointer = headpointer
    // Storage for dequeued node and adjacent nodes, loopcounter
local int store = 0, int adjstore = 0, int i = 0
local int adj[size(G)] = {0} // Stores the current adjacent nodes
local int tmp[size(nodes)][2] = {{0}}
    call copy(nodes, tmp, 0)
    // We mark the current node as the source node and enqueue it
    tmp[s-1][1] -= 1
    store += s
    call enqueue(queue, tailpointer, store)
call BFS2(G, tmp, headpointer, tailpointer, queue, store, adj, adjstore, order, i)
call copy(tmp, nodes, 1)
show(nodes)

uncall BFS2(G, tmp, headpointer, tailpointer, queue, store, adj, adjstore, order, i)
uncall copy(nodes, tmp, 0)
uncall enqueue(queue, tailpointer, store)
store -= s
tmp[s-1][1] += 1

// Delocal local variables
delocal int tmp[size(nodes)][2] = {{0}}
delocal int adj[size(G)] = {0}
delocal int store = 0, int adjstore = 0, int i = 0
delocal int queue[size(G)] = {0}, int tailpointer = headpointer
delocal int order[size(nodes)-1] = {0}, int headpointer = 0

F.3 Procedure: printpath

/*
 * printpath prints a path from the source of a breadth-first tree to a node
 * if such a path exists
 * Parameters:
 * int G[] A list of adjacency list elements
 * int v The destination node
 */
procedure printpath(int G[], int v)
local int u = G[v-1][1]
if G[v-1][1] = 0 then
    printf("No path exists from the source of G to node %d", v)
else
    if G[v-1][1] != -1 then
        call printpath(G, u)
        fi
    G[v-1][1] = -1
    printf("%d", v)
fi
G[v-1][1] = 0
delocal int u = G[v-1][1]

G Janus: Depth-First Search

Note: This uses a slightly different adjacency list structure.
procedure DFS(int G[], int nodes[])
    local int time = 0, int i = 0
    // We run through each individual node and check if they have been visited
    // (timestamped). If they haven’t we run DFSvisit.
    from i = 0 do
        if nodes[i][1] = 0 then
            call DFSvisit(G, nodes, i, time)
        fi
        nodes[i][2] = time
        i += 1
    until i = size(nodes)
    delocal int time = size(nodes) * 2, int i = size(nodes)
    
    procedure DFSvisit(int G[], int nodes[], int u, int time)
    // The time is incremented and the node gets entry-timestamped
    time += 1
    nodes[u][1] += time
    
    // end contains a pointer to the end of the wanted list
    local int end = 0
    if u = size(nodes) - 1 then
        // If we try to get the last list we stop at the end of the element
        
        */
```plaintext
list
end += size(G)
else
    // Or else, we stop where the next list starts
    end += nodes[u + 1][0]
fi
u = size(nodes) - 1

// The adjacent nodes are checked and if they have not been visited,
// DFSvisit is called with the unvisited node
local int r = nodes[u][0]
from r = nodes[u][0] do
    if nodes[G[r]-1][1] = 0 then // The adjacent node's first time stamp
        is 0
        call DFSvisit(G, nodes, G[r]-1, time)
    fi
    nodes[G[r]-1][2] = time // The adjacent node's second time
        matches the current time
    r += 1
until r = end
delocal int r = end

// The time is incremented and the node gets exit-timestamped
time += 1
nodes[u][2] += time

// end is delocated
if u = size(nodes) - 1 then
    end -= size(G)
else
    end -= nodes[u + 1][0]
fi
u = size(nodes) - 1
delocal int end = 0
```

H Janus: Kruskal’s Algorithm

H.1 Version 1

```plaintext
/*
* mstkruskal is the main algorithm, creating a minimum spanning tree from a
* undirected, weighted graph.
* Parameters:
* int G[]        An adjacency matrix representing G
* int edgecount  The number of edges in the graph G
* int A[]        An empty adjacency matrix for storing the result
* Returns:
* An adjacency matrix containing a minimum spanning tree of G.
*/
procedure mstkruskal(int G[], int edgecount, int A[])
```
14 // Initialize variables for loop and findset
15 local int i = 0, int ret = 0, int visit = 0
16 local int visited[size(G)] = {0}
17
18 // Gather all of the edges into a list.
19 local int edges[3][edgecount] = {{0}, {0}, {0}}
20 call extractedges(G, edges)
21 // Sort the edge list using quicksort
22 local int p[edgecount] = {0}
23 call id_perm(p, edgecount)
24 call qsort2(edges, 0, edgecount - 1, p)
25
26 from i = 0 do // Go through all edges
27 // Findset checks two nodes combined by an edge are in the same
28 // subset
29 call findset(edges[1][i]-1, edges[2][i]-1, A, visited, visit, ret)
30 if ret = 0 then
31 // If they aren’t, the two nodes are connected
32 uncall findset(edges[1][i]-1, edges[2][i]-1, A, visited, visit, ret)
33 call addedge(edges[1][i]-1, edges[2][i]-1, edges[0][i], A)
34 else
35 uncall findset(edges[1][i]-1, edges[2][i]-1, A, visited, visit, ret)
36 fi A[edges[1][i]-1][edges[2][i]-1] != 0
37 i += 1
38 until i = edgecount
39
40 uncall qsort2(edges, 0, edgecount - 1, p)
41 uncall id_perm(p, edgecount)
42 delocal int p[edgecount] = {0}
43 uncall extractedges(G, edges)
44 delocal int edges[3][edgecount] = {{0}, {0}, {0}}
45 delocal int visited[size(G)] = {0}
46 delocal int i = edgecount, int ret = 0, int visit = 0

H.1.1 Procedure: extractedges

1 /*
2 * extractedges extract the edge information (starting node, ending node,
3 * weight) of a graph G and returns it in a list.
4 */
5 * Parameters:
6 * int array[] An adjacency matrix.
7 * int edgelist[] An empty array of size |E|
8 *
9 * Returns:
10 * A list of all edges in G.
11 */
procedure extractedges(int G[], int edgelist[])
    local int i = 0, int j = 0, int s = 0
    from i = 0 do
        from j = i do
            if G[i][j] != 0 then
                edgelist[0][s] += G[i][j]
                edgelist[1][s] += i+1
                edgelist[2][s] += j+1
                s += 1
            fi
        j += 1
    until j = size(G)
    i += 1
    j -= size(G) - i
    until i = size(G)

delocal int i = size(G), int j = size(G), int s = size(G)

H.1.2 Procedure: findset

procedure findset(int a, int b, int array[], int visited[], int visit, int retval)
    // Visited and visit are used to keep track of which nodes have been visited
    // Once all nodes have been visited, we stop the recursion
    visited[a] += visit
    visit += 1

    if array[a][b] != 0 then
        retval += 1 // The element has been found, we return 1
    else
        // findset recursively searches an adjacency matrix for a connection between a and b.
        * Parameters:
        * int a First node of edge.
        * int b Second node of edge.
        * int array[] An adjacency matrix.
        * int visited[] An empty array of size |V|
        * int visit A counter value initialized to 0
        * int retval The return value
        *
        * Returns:
        * A returnvalue retval, 1 if a and b are connected by edges and 0 if they are not.
        */
local int i = 0
from i = 0 do
  // Check for edges of current node a, call function recursively
  if array[a][i] != 0 && a != i && visited[i] = 0 && retval != 1
    then
call findset(i, b, array, visited, visit, retval)
fi array[a][i] != 0 && a != i && visited[i] = visit
i += 1
until i = size(array)
delocal int i = size(array)
fi array[a][b] != 0
visit -= 1

H.1.3 Procedure: addedge

procedure addedge(int a, int b, int weight, int array[])
  array[a][b] += weight
  if a != b then // Symmetric edge only added when not in the diagonal
    array[b][a] += weight
fi a != b

H.2 Version 2

/*
* mstkruskal is the main algorithm, creating a minimum spanning tree from a
* undirected, weighted graph.
*
* Parameters:
* int G[] An adjacency matrix representing G
* int edgecount The number of edges in the graph G
* int A[] An empty adjacency matrix for storing the result
*/
procedure mstkruskal(int G[], int edgecount, int A[],
    int nodes[], int set[], int splitgarbage[])
    // Initialize variables for loop, checksets and split
    local int i = 0, int found = 0, int splitcounter = 0,
        int sets[size(G)][3] = {{0}, {0}, {0}}
    // Gather all of the edges into a list.
    local int edges[edgecount][3] = {{0}}
    call extractedges(G, edges)
    // Sort the edge list using mergesort
    local int p[edgecount] = {0}
    local int r = 0
    from r = 0 do
        p[r] += r
        r += 1
    until r = edgecount
    call msort(edges, edgecount, p)
    from i = 0 do
        // Create the edgeset
        call makeset(i, nodes, sets)
        i += 1
    until i = edgecount
    i -= edgecount
    from i = 0 do
        // Go through all edges
        // checksets checks two nodes combined by an edge are in the same subset
        call checksets(edges[i][1]-1, edges[i][2]-1, nodes, found)
        if found = 0 then
            // If they aren’t, the two nodes are connected
            uncall checksets(edges[i][1]-1, edges[i][2]-1, nodes, found)
            A[edges[i][1]-1][edges[i][2]-1] += edges[i][0]
            A[edges[i][2]-1][edges[i][1]-1] += edges[i][0]
            local int x = edges[i][1]-1, int y = edges[i][2]-1
            call union(x,y, nodes, sets, splitgarbage, splitcounter)
        end
        delocal int x = edges[i][1]-1, int y = edges[i][2]-1
    until i = edgecount
    sets[nodes[0][1]]0] <=> set[0]
    sets[nodes[0][1]]1] <=> set[1]
    sets[nodes[0][1]]2] <=> set[2]
// Delocal values.
uncall msort(edges, edgecount, p)
from r = edgecount do
  p[r-1] -= r-1
  r -= 1
until r = 0
delocal int r = 0
delocal int p[edgecount] = {0}
uncall extractedges(G, edges)
delocal int edges[edgecount][3] = {{0}}
delocal int i = edgecount, int found = 0, int splitcounter = size(G) - 1,
  int sets[size(G)][3] = {{0}, {0}, {0}}

I Janus: Prim’s Algorithm

/* MSTPrim creates a minimum spanning tree from a graph.
  * Uses an adjacency list with an added key to represent the graph.
  * int G[] An adjacency matrix representing G
  * int nodes[] Nodes of the graph G
  * int r Index for the root of the tree
  * int queue[] Empty garbage arrays for each call to extractmin
  * int extractgarbage[] Empty garbage arrays for each call to extractmin
  * int decreasegarbage[] Empty garbage arrays for each call to decreasekey
  * int decreasecounters[] Empty counter arrays for each call to decreasekey
  * int count Pointer for decreasekey, initialized to 0
  * int adjacentgarbage[] Empty garbage arrays for each call to decreasekey
  *
  * Post conditions:
  * nodes[] contains a minimum spanning tree for the graph G
  */
procedure MSTPrim(int G[], int nodes[], int r, int queue[],
  int extractgarbage[], int decreasegarbage[],
  int decreasecounters[], int count, int adjacentgarbage[])
  local int i = 0, int queuesize = size(queue)-1, int extractcount = 0
call initialize(nodes, queue, r)
from queuesize = size(queue)-1 do
  local int u = 0, int adjsize = 0
call extractmin(queue, queuesize, u, extractgarbage, extractcount)
call getadjsize(u, nodes, G, adjsize)
call getlist(G, nodes, u-1, adjsize)
  local int j = 0
from j = 0 do
  local int k = 0, int inqueue = 0
call isinqueue(inqueue, k, queuesize, queue, j, 
    adjlist, adjacentgarbage, count)
if inqueue = 1 &&
    G[nodes[u-1][0] + j][1] < queue[k-1][1] then
    adjacentgarbage[count][1] <= nodes[adjlist[j]-1][1]
    nodes[adjlist[j]-1][1] += u
    call decreasekey(queue, adjacentgarbage[count-1][0], 
        G[nodes[u-1][0] + j][1],
        decreasegarbage, decreasecounters, count-1)
fi inqueue = 1 && nodes[adjlist[j]-1][1] = u
j += 1
if queuesize != 0 then
    from k = queuesize + 1 do
        k -= 1
        if queue[k][0] = adjlist[j-1] && inqueue = 1 && k != 0 then
            inqueue -= 1
        fi queue[k][0] = adjlist[j-1] && inqueue = 0 && k != 0
        until k = 0
    fi queuesize != 0
until j = size(adjlist)
delocal int k = 0, int inqueue = 0
delocal int j = size(adjlist)
uncall getlist(G, nodes, u-1, adjlist)
delocal int adjlist[adjsize] = {0}
uncall getadjsize(u, nodes, G, adjsize)
delocal int u = queue[queuesize+1][0], int adjsize = 0
until queuesize = 0
delocal int i = 0, int queuesize = 0, int extractcount = size(nodes)

I.1 Procedure: isinqueue

/* isinqueue checks if a value is in a minimum priority queue. 
* int inqueue Return value indicating 1 true, 0 if false 
* int k The value to check for 
* int queuesize The current size of the queue 
* int queue[] The queue to search 
* int j Indicates the placement of the value in a given 
* adjacency list 
* int adjlist An adjacency list 
* int adjacentgarbage Garbage for storing the element, if it is found 
* int count A garbage counter value 
* 
* Post conditions: 
* If the element is found, j indicates where it has been found, and the 
* garbage has been added and its counter incremented */
procedure isinqueue(int inqueue, int k, int queuesize, int queue[], int j, int adjlist[], int adjacentgarbage[], int count)

    if queuesize != 0 then
        k += 1
        from k = 1 do
            if queue[k][0] = adjlist[j] then
                inqueue += 1
                adjacentgarbage[count][0] += k
                count += 1
                queue[k][0] = adjlist[j]
                k += 1
            until k = queuesize + 1
        fi queuesize != 0
    fi

I.2 Procedure: initialize

    procedure initialize(int nodes[], int queue[], int s)
        local int i = 2, int n = 1
        queue[1][0] += s
        nodes[0][1] -= 1
        from i = 2 do
            if i = s + 1 then
                n += 1
            fi i = s + 1
            queue[i][0] += n
            queue[i][1] += 4294967295
            nodes[i-1][1] -= 1
            n += 1
            i += 1
        until i = size(nodes) + 1
        delocal int i = size(nodes) + 1, int n = size(nodes) + 1

J Janus: Dijkstra’s Algorithm

    procedure MSTPrim creates a minimum spanning tree from a graph.
    * Uses an adjacency list with an added key to represent the graph.
procedure dijkstra(int G[], int nodes[], int s, int set[], int queue[],
    int relaxgarbage[], int relaxcounters[], int
    extractgarbage[],
    int count, int adjacentgarbage[])
local int queuesize = size(queue) - 1, int extractcounter = 0
call initialize(nodes, queue, s)

from queuesize = size(queue)-1 do
    local int u = 0, int adjsize = 0
call extractmin(queue, queuesize, u, nodes,
    extractgarbage, extractcounter)
call getadjsize(u, nodes, G, adjsize)
local int adjlist[adjsize] = {0}
call getlist(G, nodes, u-1, adjlist)
call joinset(set, u, nodes)
local int i = 0
from i = 0 do
    if queuesize != 0 then
        local int weight = G[nodes[u-1][0]+i][1]
call relax(u, adjlist[i], nodes, weight,
    relaxgarbage, relaxcounters, queue, queuesize,
    count, adjacentgarbage)
delocal int weight = G[nodes[u-1][0]+i][1]
    fi queuesize != 0
    i += 1
until i = size(adjlist)
delocal int i = size(adjlist)
uncall getlist(G, nodes, u-1, adjlist)
delocal int adjlist[adjsize] = {0}
uncall getadjsize(u, nodes, G, adjsize)
delocal int u = queue[queuesize+1], int adjsize = 0
until queuesize = 0
delocal int queuesize = 0, int extractcounter = size(nodes)
J.1 Procedure: joinset

/* joinset adds the current value of node u to its corresponding place in a set. */

procedure joinset(int set[], int u, int nodes[])

    set[u][0] += nodes[u][0]
    set[u][1] += nodes[u][1]
    set[u][2] += nodes[u][2]

J.2 Procedure: relax

/* relax relaxes an edge between the nodes u and v. */

procedure relax(int u, int v, int nodes[], int weight,
    int relaxgarbage[], int relaxcounters[],
    int queue[], int queuesize,
    int count, int adjacentgarbage[])

    local int i = 0
    from i = 0 do
        if queue[i+1] = v then
            adjacentgarbage[count][0] += i + 1
            count += 1
        queue[i+1] = v
        i += 1
    until i = queuesize
    delocal int i = queuesize
if nodes[v-1][2] > nodes[u-1][2] + weight then
    call decreasekey(queue, adjacentgarbage[count-1][0], nodes[u-1][2] + weight,
        relaxgarbage, relaxcounters, count-1, nodes)
    nodes[v-1][1] <=> adjacentgarbage[count][1]
    nodes[v-1][1] += u
fi
nodes[v-1][2] = nodes[u-1][2] + weight

K Graph generator

The graph generator is made in Python and can create custom sized random graphs of all 4 types. It prints the graph to the file graph.txt.

# Program used to generate randomized graphs.
# Will save a randomly generated graph of the type choses to graph.txt.
#
# The 4 different types of graphs are:
# 1. Unweighted, undirected graph
# 2. Unweighted, directed graph
# 3. Weighted, undirected graph
# 4. Weighted, directed graph
#
# The amount of nodes and type of graph is decided by keyboard input
# when running the program

import random

def uwud():
    file = open("graph.txt", "w")
    print "An unweighted, undirected graph has been printed to graph.txt"
    graph = [[0 for x in range(size)] for x in range(size)]
    for x in range(size):
        for y in range(size):
            if x < y:
                graph[x][y] = random.randint(0, 1)
                graph[y][x] = graph[x][y]
    file.write("{{
        for y in range(size):
            if x < y:
                graph[x][y] = random.randint(0, 1)
                graph[y][x] = graph[x][y]
    file.write("{{}
        for row in graph}}
    file.write("})
    file.close()

def uwd():
    file = open("graph.txt", "w")
    print "An unweighted, directed graph has been printed to graph.txt"
    graph = [[0 for x in range(size)] for x in range(size)]
    for x in range(size):
        for y in range(size):
            if x != y:
                graph[x][y] = random.randint(0, 1)
                graph[y][x] = graph[x][y]
    file.write("{{
        for y in range(size):
            if x != y:
                graph[x][y] = random.randint(0, 1)
                graph[y][x] = graph[x][y]
    file.write("{{}
        for row in graph}}
    file.write("})
    file.close()
graph[x][y] = random.randint(0, 1)
file.write("{"
file.write('},
{'.join([','.join(['{:1}'.format(item) for item in row]) for row in graph]))
file.write("}"
file.close()

def wud():
    maxweight = input("Max edge weight: ")
    file = open("graph.txt", "w")
    print "A weighted, undirected graph has been printed to graph.txt"
    graph = [[0 for x in range(size)] for x in range(size)]
    for x in range(size):
        for y in range(size):
            if x < y:
                graph[x][y] = random.randint(0, maxweight)
                graph[y][x] = graph[x][y]
    file.write("{"
    file.write('},
{'.join([','.join(['{:1}'.format(item) for item in row]) for row in graph]))
    file.write("}"
    file.close()

def wd():
    maxweight = input("Max edge weight: ")
    file = open("graph.txt", "w")
    print "A weighted, directed graph has been printed to graph.txt"
    graph = [[random.randint(0, maxweight) for x in range(size)] for x in range(size)]
    for x in range(size):
        for y in range(size):
            if x != y:
                graph[x][y] = random.randint(0, maxweight)
    file.write("{"
    file.write('},
{'.join([','.join(['{:1}'.format(item) for item in row]) for row in graph]))
    file.write("}"
    file.close()

# map the inputs to the function blocks
options = {1: uwud,
           2: uwd,
           3: wud,
           4: wd,
           }

while True:
    size = input("Size of graph: ")
if size > 0 and isinstance(size, int):
    break;
else:
    print "Please insert a positive integer as size"

print "Choose graph type by entering the number corresponding to the graph type"
print "(1: Unweighted, undirected, 2: Unweighted, directed,"
print "3: Weighted, undirected, 4: Weighted, directed)"

type = input("Choice: ")
while True:
    try:
        options[type]()
        break
    except KeyError:
        print "Incorrect graph type. Please choose a number between 1 and 4"
        type = input("Choice:")

K.1 Graph converter

The graph converter is a Janus program that can be used to convert adjacency matrices to adjacency lists. It supports 3 different kinds of conversions (to an adjacency list without any node information, to an adjacency with node information and to an adjacency list with both weight and node information).

The converter is very simple, focusing only on the task needed.

procedure listtomatrix()
    skip

procedure wmatrixtolist(int matrix[], int G[], int nodes[], int counter)
    local int i = 0
    local int j = 0
    from i = 0 do
        nodes[i] += counter
        from j = 0 do
            if matrix[i][j] != 0 then
                G[counter][0] += j + 1
                G[counter][1] += matrix[i][j]
                counter += 1
            fi
            matrix[i][j] != 0
            j += 1
        until j = size(matrix)
        j -= size(matrix)
i += 1
until i = size(matrix)
delocal int j = 0
delocal int i = size(matrix)

/* Converts a matrix into an adjacency list that can contain extra node info */
procedure matrixtolist(int matrix[], int G[], int nodes[], int counter)
local int i = 0
local int j = 0
from i = 0 do
    nodes[i] += counter
    from j = 0 do
        if matrix[i][j] != 0 then
            G[counter][0] += j + 1
            counter += 1
        fi
        matrix[i][j] != 0
        j += 1
until j = size(matrix)
i += 1
until i = size(matrix)
delocal int j = 0
delocal int i = size(matrix)

/* Converts a matrix into an adjacency list without any extra node info */
procedure simplematrixtolist(int matrix[], int G[], int nodes[], int counter)
local int i = 0
local int j = 0
from i = 0 do
    nodes[i] += counter
    from j = 0 do
        if matrix[i][j] != 0 then
            G[counter] += j + 1
            counter += 1
        fi
        matrix[i][j] != 0
        j += 1
until j = size(matrix)
i += 1
until i = size(matrix)
delocal int j = 0
delocal int i = size(matrix)

procedure wtransfer(int input[], int output[], int counter)
local int i = 0
from i = 0 do
    output[i][0] += input[i][0]
    output[i][1] += input[i][1]
i += 1
until i = counter

delocal int i = counter

procedure transfer(int input[], int output[], int counter)
    local int i = 0
    from i = 0 do
        output[i][0] += input[i][0]
        i += 1
    until i = counter

delocal int i = counter

procedure simpletransfer(int input[], int output[], int counter)
    local int i = 0
    from i = 0 do
        output[i] += input[i]
        output[i] += input[i]
        i += 1
    until i = counter

delocal int i = counter

procedure main()
    int matrix[][] = {{2,4,2,5,0,0,2,4,4,1},
                     {4,0,3,1,4,3,1,1,4,5},
                     {4,5,4,4,0,2,2,2,3,1},
                     {2,3,5,1,1,3,5,4,5,1},
                     {5,0,0,1,4,5,2,2,0,1},
                     {1,3,3,2,4,1,0,2,0,5},
                     {5,4,2,5,3,2,1,1,5,2},
                     {1,4,4,2,4,1,3,1,3,5},
                     {1,0,1,3,4,5,3,3,3,0},
                     {4,3,1,4,5,4,1,5,0,3}}
    int G[size(matrix)*size(matrix)][2]
    int nodes[size(matrix)]
    int counter

    call wmatrixtolist(matrix, G, nodes, counter)
    local int shortG[counter][2] = {{0}}
    call transfer(G, shortG, counter)
    show(shortG)
    uncall transfer(G, shortG, counter)
    delocal int shortG[counter][2] = {{0}}