ENLARGING DIRECTED GRAPHS TO ENSURE ALL NODES ARE CONTAINED IN CYCLES

by

JAN JOHANNES VAN DER LINDE

submitted in accordance with the requirements for the degree of
MASTER OF SCIENCE
in the subject
COMPUTING
at the
UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROF. IAN SANDERS

2015

Declaration

I declare that **"Enlarging directed graphs to ensure all nodes are contained in cycles"** is my own work and that all the sources that I have used or quoted have been indicated and acknowledged by means of complete references.

I further declare that I have not previously submitted this work, or part of it, for examination at Unisa for another qualification or at any other higher education institution.

J. J. van der Linde December 2, 2015

Abstract

Graph augmentation concerns the addition of edges to a graph to satisfy some connectivity property of a graph. Previous research in this field has been preoccupied with edge augmentation; however the research in this document focuses on the addition of vertices to a graph to satisfy a specific connectivity property: ensuring that all the nodes of the graph are contained within cycles. A distinction is made between graph augmentation (edge addition), and graph enlargement (vertex addition).

This document expands on previous research into a graph matching problem known as the "shoe matching problem" and the role of a graph enlargement algorithm in finding this solution. The aim of this research was to develop new and efficient algorithms to solve the *graph enlargement problem* as applied to the shoe matching problem and to improve on the naïve algorithm of Sanders.

Three new algorithms focusing on graph enlargement and the shoe matching problem are presented, with positive results overall. The new enlargement algorithms: cost-optimised, matrix, and subgraph, succeeded in deriving the best result (least number of total nodes required) in 37%, 53%, and 57% of cases respectively (measured across 120 cases). In contrast, Sanders's algorithm has a success rate of only 20%; thus the new algorithms have a varying success rate of approximately 2 to 3 times that of Sanders's algorithm.

Acknowledgements

My gratitude and respect to my advisor, **Prof. Ian Sanders**, for his guidance and patience during the writing of this dissertation. I am grateful for the opportunity given to me to continue his research ideas in this field. It has been a pleasure and a privilege to work with you, Professor.

Maheshini Govender for her love and support during the writing of this dissertation, her regular proofreading of it, and also for her help in the drawing of the graph structures present in this dissertation. To paraphrase Carl Sagan, one of our mutual favourite scientists: *"In the vastness of space and the immensity of time, it is my joy to share a planet and an epoch with you."*

My brother, **Ian**, a fellow computer scientist, whose ideas, knowledge, and remarks have always been a great help to me, as well as his proofreading of this dissertation.

My parents, **Jan and Zelda**, for their support throughout my life.

I also wish to thank the Directorate of Student Funding at the **University of South Africa** for the bursary awarded to me during the course of my studies.

A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine.

Alan M. Turing

Preface

Parts of this dissertation were compiled into an article and submitted to the SAICSIT 2015 conference. The paper was accepted and published in the conference proceedings. The theme of the conference was "*Knowledge Through Technology*" and ran from 28-30 September 2015. The full paper, as accepted for publication, can be found in Appendix C on p. 113.

Contents

List of Figures

List of Tables

List of Algorithms

This page intentionally left blank.

Chapter 1

Introduction

Contents

1.1 Background to the Problem

The shoe matching problem (Sanders, 2013*a*,*b*) involves people who require different sized shoes between their left and right feet. By cooperating with one another, and buying pairs of shoes as a group, they can save money and satisfy everyone's footwear needs. Instead of every person buying two pairs of shoes, he/she simply needs to find one or more persons to match up with. If, for example, one person needed a left shoe of size 8 and a right shoe of size 10, the ideal match would be someone with the opposite need. By cooperating, they need only buy a single pair of size 8's and a single pair of size 10's. However, in real life, the solution is rarely as simple as that. Matchings might need to occur within a large group of people to satisfy everyone's needs and even then not everyone may find a match. In such a case the group must buy extra pairs of shoes to ensure that everyone benefits from the arrangement.

Thus, while a naïve (and fast) solution to the problem is to furnish every participant with an extra pair of shoes, it would result in extra cost that could potentially have been avoided. For most people with differently sized feet, this costly solution is currently the norm. By forming groups within communities, participants will be able to continue cooperating to save money.

By abstracting the shoe matching problem as a graph theory problem (see

Section 2.6 on p. 23), one can utilise the techniques of graph theory, specifically cycle picking, to make sure that everyone finds their match within the group or, in the worst case scenario, make use of graph enlargement to add dummy pairs of shoes, followed by cycle picking, to satisfy the needs of all the group members.

Figure 1.1 illustrates an example of the shoe matching problem. The current cycles in the graph are:

- (1: Monde) $\rightarrow$ (5: David) $\rightarrow$ (4: Kefentse) $\rightarrow$ (3: Hendrik) $\rightarrow$ (6: Yoosuf)
- (1: Monde) $\rightarrow$ (5: David) $\rightarrow$ (6: Yoosuf)
- (3: Hendrik) $\rightarrow$ (4: Kefentse)
- (7: Kopano) $\rightarrow$ (8: Mark)

All the nodes are contained within some cycle, except for node number 2: John. This means everyone has a partner (or belongs to a group of participants) to swap shoes with, such that everyone finds a pair that fits, except for John. One therefore needs to add a dummy node (an extra pair of shoes), with the goal of enabling John to become part of one of the existing cycles.

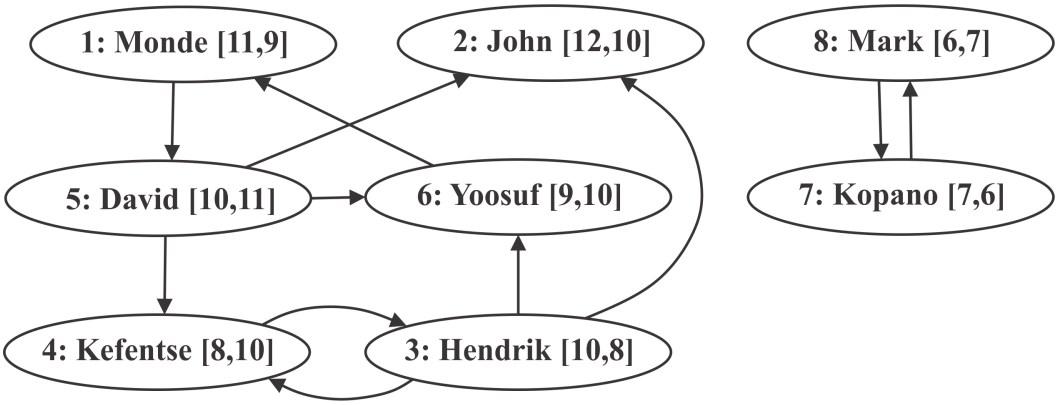


Figure 1.1: Sanders's original example of the shoe matching problem

Graph enlargement, as described in this dissertation, refers to the problem of enlarging a graph G, in which nearly all nodes are contained in cycles, to produce an enlarged graph G' in which all the nodes will be contained within cycles. Figure 1.2 illustrates the graph after enlargement has been applied. Note that John is now contained within a cycle, and the graph has been enlarged with two dummy nodes, namely nodes 9 and 10. In Chapter 4 it is shown that the problem above could have been solved by adding only a single dummy node. The cycles in the graph are now as follows:

- (1: Monde) → (5: David) → (2: John) → (9: Dummy) → (8: Mark) → (10: Dummy) → (3: Hendrik) → (6: Yoosuf)
- (1: Monde) → (5: David) → (4: Kefentse) → (3: Hendrik) → (6: Yoosuf)
- (1: Monde) → (5: David) → (6: Yoosuf)
- (2: John) → (9: Dummy) → (8: Mark) → (10: Dummy) → (3: Hendrik)
- (3: Hendrik) → (4: Kefentse)
- (7: Kopano) → (8: Mark)

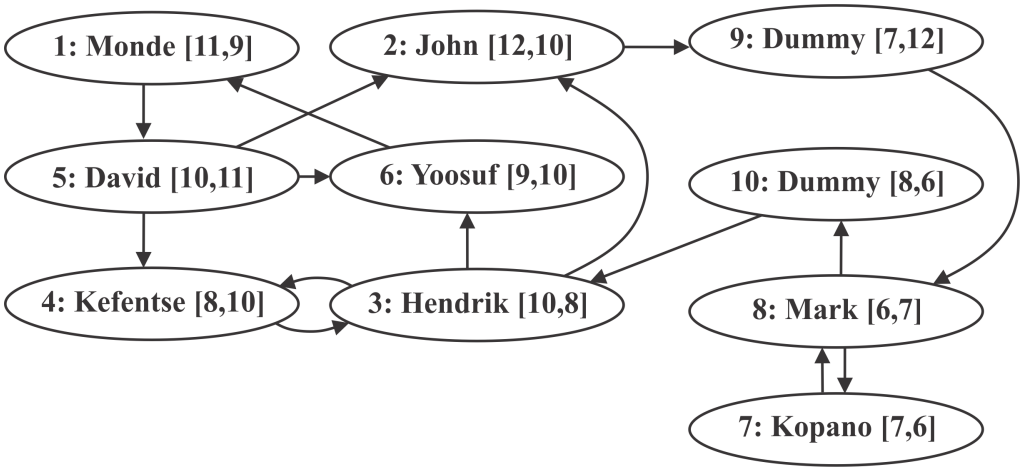


Figure 1.2: Sanders's original example of the shoe matching problem, post-enlargement

In Section 2.6 (p. 23) it is shown that graph augmentation (enlarging the graph by adding edges) is not a new concept. However, these previous papers were mostly concerned with edge augmentation and the edge connectivity of the graph. The research within this dissertation focuses on a different property: to ensure the graph is a collection of cycles and that all nodes are contained within these cycles, by adding vertices as required. **A distinction is thus made between graph augmentation (adding edges to the graph) and graph enlargement (adding vertices and specific edges to the graph)**.

It would not be possible to use the techniques of graph augmentation to solve the shoe matching problem, since the problem cannot be solved by adding only edges to the graph (edges are only created if a matching is possible between two nodes). People are represented by nodes, and matchings between people are indicated by edges. The addition of dummy nodes (and accompanying matching edges) are necessary to solve the problem.

Sanders (2013*a*) approached the shoe matching problem by generating input data to construct a representative graph. A cycle enumeration algorithm would assist in determining whether all the nodes of the graph were contained within cycles. If not, Sanders's graph enlargement algorithm would add the required dummy nodes, and a cycle picking algorithm would choose the optimal combination of cycles to minimise the total cost.

Cycle picking is an optimisation problem; to minimise the number of cycles, such that any node originally contained within a cyle(s) will be contained within the fewest number of cycles possible. Sanders (2013*a*,*b*) presented applications of both cycle picking and graph enlargement. In the example illustrated above, several nodes appear in multiple cycles. By applying cycle picking, an optimal combination of cycles can be selected. The number of nodes is minimised in this combination of cycles.

While the graph enlargement algorithm developed by Sanders (2013*a*,*b*) does enlarge the graph correctly, by adding the required dummy nodes and appropriate edges, the algorithm does not consider the properties of the graph and makes no attempt to minimise the total node cost. Nodes are more or less chosen at random, ideally any node with no incoming edges will be connected to any node with no outgoing edges.

1.2 Aim, Methodology, and Results

The shoe matching problem is an example of a simple matching problem. Solving the problem possibly requires the addition of extra dummy nodes to ensure that all nodes are contained within cycles. The **problem of efficiently adding nodes to an existing graph** necessitates the development of a new algorithm. The existing algorithm by Sanders (2013*a*) provides a good solution in most cases, but it is not always the most efficient in terms of total node cost or running time.

The primary aim of this research was to develop new and efficient algorithms to solve the *graph enlargement problem*. The main focus points for improvement for these algorithms were the total node cost to participants (in a scenario such as the shoe matching problem), and also the fastest possible running time, respectively. It was also investigated whether enlarging only a subgraph of the original graph, consisting of the nodes not contained within cycles, results in improved performance.

The new algorithms were compared to the existing one by Sanders (2013*a*) in terms of run-time and node-cost of the solution. Test data was randomly generated for comparison purposes; the data was simulated subject to reasonable assumptions (with regards to average shoe sizes and average differences between shoe sizes), since no real world data could be found.

In this dissertation, it is shown that several improvements can be made

to the original algorithm, in terms of node-cost (number of nodes) and the speed / running time of the algorithm. It is also shown that it is possible to efficiently enlarge large graphs (1000 nodes, for example) within a matter of seconds.

1.3 Expected Contribution of the Dissertation

This dissertation focuses on graph enlargement and its application to simple matching problems, specifically the shoe matching problem. Graph enlargement is an unexplored topic within graph theory with a rich potential of undiscovered applications. Three new algorithms are provided, each with its own set of advantages and disadvantages, which can be applied by the reader to other matching problems.

It also extends the research done by Sanders (2013*a*,*b*) by providing faster and more cost-effective (in terms of total node cost) solutions to the shoe matching problem.

1.4 An Overview of the Remainder of the Dissertation

Other researchers have, in previous years, focused on the problem of graph augmentation by adding edges to a graph to satisfy certain properties. The problem of enlarging a graph by adding vertices is a relatively new concept. For the purposes of this dissertation and the shoe matching problem several related concepts, such as cycle enumeration and cycle picking, are also applied. These topics are examined in greater detail in the literature review found in Chapter 2. Some basic background and terminology related to graph theory concepts are also explained, but the reader familiar with these concepts may simply gloss over these sections.

Chapter 3 covers the research methodology followed in this dissertation. The reasons for selecting specific cycle enumeration and cycle picking methods, for example, are provided and substantiated. The generation of input data due to a lack of availability of real-world data is also discussed. Finally, the chapter covers the assessment criteria to determine whether the research aim was achieved.

The experimental work involving new algorithms and their comparison to Sanders's algorithm are detailed and tabulated in Chapter 4. This chapter also provides detailed pseudocode for the new algorithms and other applicable subroutines. The different optimisation techniques employed are individually and thoroughly examined. An overview of the results and a summary of the optimisation techniques are covered in Chapter 5.

Potential future research opportunities are presented in Chapter 6. The possibility of parallelising the existing and new algorithms could greatly enhance the performance of the graph enlargement process. One could even port the existing work to a language designed for such a purpose, such as Google's new in-house developed language: Go. There is also an opportunity to implement the algorithms in a practical manner, such as a web interface allowing real people to register and participate in the shoe matching problem.

Chapter 7 restates the results and conclusions drawn in this dissertation.

Chapter 2

Background

Contents

2.1 Introduction

Sanders (2013*a*,*b*) originally abstracted the problem of people requiring different size shoes (between their left and right feet) into a graph theory problem, specifically the problem of cycle picking. Each node of the graph represents a person who cooperates with other people (nodes) to exchange a shoe of incompatible size for the correct size shoe. Cycles represent groups of people who are able to work together to satisfy their footwear needs. By employing cycle picking, an optimal combination of cycles can be found to minimise the monetary cost to the participants. However, before cycle picking can occur, the existing cycles within the graph must be enumerated.

Ideally one would hope for everyone to find a match without the need for dummy pairs of shoes, but in the real world this is an often unrealised ideal. Sanders (2013*a*,*b*) developed an algorithm to add dummy nodes as required, to ensure that all the participants find a match; in other words,

ensuring that all the nodes are contained within cycles. For detailed information on Sanders's algorithm (including pseudocode), please see Section 2.6.1 (p. 27).

Sanders's algorithm for adding nodes to an existing graph is essentially a form of graph augmentation in principle. Eswaran and Tarjan (1976) originally defined graph augmentation as the problem of determining how many **edges** must be added to a graph to satisfy a given **connectivity** property C. **The formal definition of graph augmentation is given in Section 2.6 (p. 23).** This dissertation will look at graph "augmentation" (referred to as graph *enlargement* to avoid ambiguity) differently: *how many nodes (their accompanying edges will follow automatically due to the problem definition) must be added to the graph to ensure that all the nodes within the graph are contained within cycles?*

The remainder of this chapter presents some basic definitions and terminology related to graph theory and graph algorithms, and then moves on to a more detailed study of existing graph algorithms, such as cycle enumeration and cycle picking algorithms, which are directly applicable to the research problem. The reader already familiar with graph theory may skip Sections 2.2 and 2.3.

2.2 Definitions and Terminology

Graphs are mathematical structures consisting of two sets, namely a vertex set, $V = \{v_1, v_2, ..., v_i\}$, as well as an edge set $E = \{e_1, e_2, ..., e_j\}$. An edge joins two vertices together, and two vertices are said to be *adjacent* when they are joined by an edge. An edge e can also be represented by a concatenation of the two vertices it joins, for example $e = uv$, where $u, v \in V$. Vertices are also called **nodes**, and will be referred to interchangeably throughout this dissertation. The order of a graph G is the number of vertices, $|V(G)|$, whilst the size of the graph is the number of edges, $|E(G)|$. The degree of a vertex, $\deg v$ refers to the number of vertices incident with v; the maximum degree of any vertex in a given graph G is denoted by $\Delta(G)$, while the minimum degree is denoted by $\delta(G)$. Weighted graphs are graphs in which the edges have been assigned weights. In general, pathfinding algorithms and spanning tree algorithms would utilise the edges with the least weight.

There are several notable graph structures which feature prominently in graph theory. These include, but are not limited to: cycles, paths, stars, bipartite graphs, complete graphs, trees, and planar graphs. Several textbooks are available for further reading on these families of graphs, such as Chartrand et al. (2011) and Bondy and Murty (2008). This dissertation focuses on cycles.

Paths and cycles are very closely related. Cycles, graphically speaking, are

indeed cyclical structures. Let G be a graph, with nodes $v_1, v_2, v_3, ..., v_{n-1}$, v_n and edges $v_1v_2, v_2v_3, ..., v_{n-1}v_n, v_nv_1$. Then the graph G is a cycle, and is denoted by C_n. A path is simply a cycle with a disconnect between the last and first node, that is to say it has nodes $v_1, v_2, v_3, ..., v_{n-1}, v_n$ and edges $v_1v_2, v_2v_3, ..., v_{n-1}v_n$. Paths are normally denoted by P_n where n is the number of nodes in P.

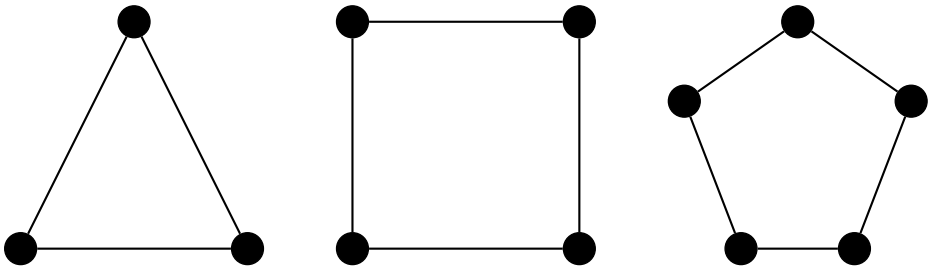


Figure 2.1: Three cycles illustrated, from left to right: C_3, C_4, C_5

2.2.1 Directed versus Undirected Graphs

Graphs can be either directed or undirected. In an undirected graph the edges do not possess direction, and one can move along any edge freely and in any direction (see Figure 2.2).

However, directed graphs consist of directed edges and are the focus of this dissertation. The edges of a directed graph **do** possess direction and one cannot move against the direction indicated (see Figure 2.3). Directed graphs are sometimes simply called digraphs, whilst directed edges can also be called arcs, arrows, or simply, edges. In this dissertation we will simply refer to digraphs and edges.

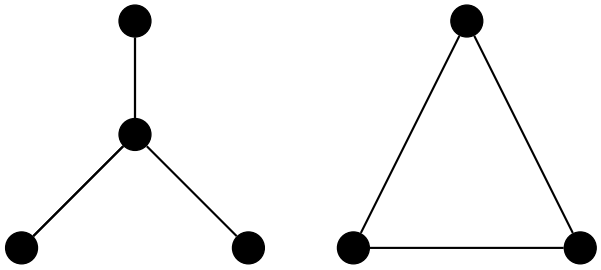


Figure 2.2: Two **undirected** *graphs are illustrated above.*

2.3 Graph Algorithms and Computer Representations of Graphs

Both graph theory and modern computer science are relatively new, but well established fields. This section will cover some classic graph algo-

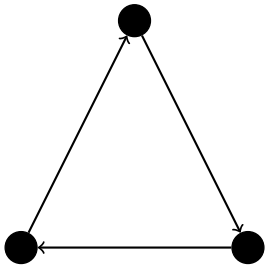


Figure 2.3: The cycle illustrated above is **directed***; the nodes must be traversed in a clockwise manner, as indicated by the arrows.*

rithms in computing, as well as the computer representation of graphs.

2.3.1 Graph Traversal Algorithms

Graph traversal algorithms play an important role in developing graph enlargement algorithms, for example, to enumerate cycles and detect components within a graph.

This section covers the following graph algorithms:

- Depth-first search
- Breadth-first search
- Connected components algorithm

Please note that **cycle enumeration algorithms** are covered in section 2.4 on p. 14.

2.3.1.1 Depth-first Search

Depth-first search (DFS) is a graph traversal algorithm that explores a branch as far as possible before backtracking. Depth-first search has several applications, including topological sorting and graph planarity testing. Note that DFS makes use of a stack data structure, and every newly discovered neighbour is pushed on top of the existing stack.

2.3.1.2 Breadth-first Search

Breadth-first search (BFS) is a graph traversal algorithm that inspects a given vertex and all of its immediate neighbours before moving on to the child nodes of neighbouring nodes. Note that BFS makes use of a queue data structure, and every newly discovered neighbour is added to the end of the queue.

Algorithm 1 Depth-first Search

```
1: procedure DFS(G, v)
2:     S ← empty stack
3:     for all u ∈ V_G do
4:         u.visited ← false
5:     end for
6:     S.push(v)                                  ▷ add to top of stack
7:     while S ≠ ∅ do
8:         u = S.pop()                            ▷ inspect first element
9:         if u.visited = false then
10:            u.visited = true
11:            for all w as neighbours of u do
12:                S.push(w)                      ▷ add to top of stack
13:            end for
14:        end if
15:    end while
16: end procedure
```

Algorithm 2 Breadth-first Search

```
1: procedure BFS(G, v)
2:     Q ← empty queue
3:     for all u ∈ V_G do
4:         u.visited ← false
5:     end for
6:     Q.enqueue(v)                               ▷ add to end of queue
7:     while Q ≠ ∅ do
8:         u = Q.dequeue()                        ▷ inspect first element
9:         if u.visited = false then
10:            u.visited = true
11:            for all w as neighbours of u do
12:                Q.enqueue(w)                   ▷ add to end of queue
13:            end for
14:        end if
15:    end while
16: end procedure
```

2.3.1.3 Connected Components Algorithm (Hopcroft-Tarjan)

Hopcroft and Tarjan (1973) provide an algorithm to discover the connected components (not necessarily strongly connected) of a graph. The logic behind the algorithm is simple and straightforward: loop through all the vertices of the graph, if the current vertex is still undiscovered use either a depth-first or breadth-first search algorithm (see Algorithms 1 and 2) to find all the nodes in the connected component containing the current undiscovered vertex. The algorithm continues until there are no undis-

covered vertices.

Algorithm 3 is a pseudocode implementation of the connected components algorithm employed for the purposes of this dissertation.

Algorithm 3 Connected Components Algorithm

```
1:  procedure COMPONENTS(G)
2:      components ← ∅
3:      for all v_i ∈ V(G) do
4:          temp ← all nodes in DFS(v_i)
5:          if V(temp) ∩ V(any existing component) ≠ ∅ then
6:              add all the vertices of temp to the existing component
7:          else
8:              create a new component based on temp
9:              add the new component to components
10:         end if
11:     end for

12:     return components
13: end procedure
```

2.3.2 Computer Representation of Graphs

It is important to understand how graphs can be represented digitally, within a computer's memory. This can either be done using matrices (i.e. arrays) or by following an object oriented approach, where classes and objects can be used to model the graph as a data structure. Even (2011) explains several methods to represent graphs by using arrays and lists; the following subsections will cover adjacency matrices, adjacency lists, and an object-oriented approach.

2.3.2.1 Adjacency Matrices

Directed and undirected graphs can be represented by an $n \times n$ matrix. The matrix, say M, can indicate an edge between two vertices v_i and v_j by letting $M_{ij} = 1$, whilst the absence of an edge can be indicated by letting $M_{ij} = 0$. In an undirected graph, the matrix is symmetric, that is to say that $M_{ij} = M_{ji}$. This matrix M is called an *adjacency matrix* and can be represented digitally by a 2-dimensional array.

An adjacency matrix requires $O(n \times n)$ memory, but allows the computer to quickly check whether an edge exists or not. In a sparse graph, an adjacency matrix contains mostly zeroes and is an inefficient use of memory.

To make provision for digraphs we alter the matrix as follows:

- The i-th row represent the outgoing edges of v_i and the sum of the 1's in the i-th row gives $\mathrm{d}_{\mathrm{out}}(v_i)$.
- The j-th column represent the incoming edges of v_j and the sum of the 1's in the j-th column gives $\mathrm{d}_{\mathrm{in}}(v_j)$.

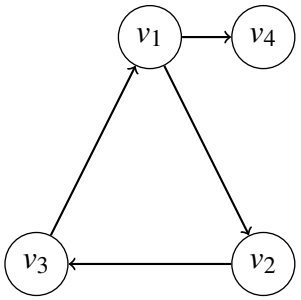


Figure 2.4: Deriving an adjacency matrix from a directed graph.

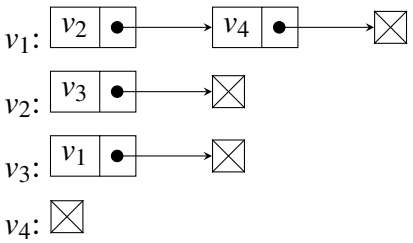
The adjacency matrix for the graph in Figure 2.4 is given by

$$M = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

2.3.2.2 Adjacency Lists

An adjacency list is an alternative to the adjacency matrix. Adjacency lists are more storage-efficient than adjacency matrices in the case of sparse graphs. Adjacency lists are slower when checking for the existence of specific edges in the graph, though.

For the directed graph in Figure 2.4, the adjacency list can be represented by linked lists as follows:



The adjacency list need not make use of linked lists; simple arrays (and dictionaries, if available) can also be used to represent the vertices and their relationships.

2.3.2.3 Object-oriented Approach

An OO (object-oriented) solution can be approached in a variety of ways. A Vertex class can be constructed which keeps lists of incoming and outgoing adjacencies. These adjacent vertices are also of the Vertex class and in turn have their own lists of adjacent vertices. An OO approach also allows the easy addition of vertex properties, such as vertex colouring. Methods can also be written to determine certain vertex properties, such as d_{in} and d_{out}, for example.

2.4 Cycle Enumeration

Cycle enumeration is the process of using a depth-first or breadth-first search (DFS / BFS) to traverse all the possible cycles within a graph. The number of cycles can increase very rapidly (even exponentially) as the number of nodes in a graph increases.

In 1970, James Tiernan published an algorithm to find the elementary circuits of a graph. Tiernan's paper focused on directed graphs, and he deemed his algorithm efficient due to the fact that every circuit in the graph was considered only once. The algorithm developed by Tiernan is iterative and favours small circuits (Tiernan, 1970).

Tiernan (1970) defines an elementary circuit to be an elementary path (each vertex is contained in the path at most once), except that the first and last vertices are the same. The algorithm requires every node to be labeled with some integer value from 1 to n where n is the order of the graph. The order of the labelling is not important.

The algorithm makes use of two arrays: the first, P is one-dimensional and is used to build the elementary path; the second, H, is a two-dimensional array filled with zeroes to begin with and will keep track of all the vertices "closed" (see below for clarification) to each vertex. The algorithm starts at the vertex labelled **1**, adding it to P, and tentatively extends to the next vertex as long as the following conditions are met:

1. The new vertex may not already be contained within P.
2. The integer label of the new vertex must be larger than the first vertex in P.
3. The new vertex cannot be closed to the last vertex in P.

Tiernan's 3^{rd} condition uses unorthodox terminology and can be clarified as follows: once a cycle is obtained, or a dead end reached, the last vertex added is removed from P and the second-to-last vertex becomes "closed" to the last vertex. The algorithm will continue to trace back through the

vertices in P until another path is possible. Essentially this means the algorithm will not repeatedly revisit the same vertices, and therefore won't continuously rediscover the same cycles. When all the possible cycles have been discovered that contain the vertex labelled 1, the algorithm moves on to the vertex labelled 2, and repeats the process; this continues until all N vertices have been considered.

The algorithm's parameters consist of G, the graph, and N, the order of the graph. A pseudocode version of Tiernan's algorithm is given below in Algorithms 4 and 5. The program starts at procedure **EC1**; the rest of the program flow is indicated by **goto** statements.

Algorithm 4 Tiernan's Algorithm (Part 1 of 2)

```
1:  procedure EC1: INITIALISATION(G,N)
2:      Read N and G
3:      P ← 0                                   ▷ 1-Dimensional array
4:      H ← 0                                   ▷ 2-Dimensional array
5:      k ← 1                                   ▷ Start at vertex 1
6:      P[1] ← 1
7:      go to EC2
8:  end procedure

9:  procedure EC2: PATH EXTENSION
10:     Search G[P[k], j] for j = 1,2,...,N such that G[P[k], j] satisfies the 3
        conditions listed on p. 14.
11:     if a value for j is found then
12:         Extend the path
13:         k ← k + 1
14:         P[k] ← G[P[k − 1], j]
15:         go to EC2
16:     else
17:         The path cannot be extended any further; either the circuit is
        complete or it's a dead end.
18:     end if
19: end procedure

20: procedure EC3: CIRCUIT CONFIRMATION
21:     if P[1] ∉ G[P[k], j], j = 1,2,...,N then
22:         No circuit has been formed.
23:         go to EC4
24:     else
25:         Circuit has been formed.
26:         Print P
27:     end if
28: end procedure
```

Algorithm 5 Tiernan's Algorithm (Part 2 of 2)

29: **procedure** EC4: VERTEX CLOSURE
30: **if** $k = 1$ **then**
31: All of the circuits containing the vertex $P[1]$ have been considered.
32: **go to** EC5
33: **else**
34: $H[P[k], m] \leftarrow 0, \forall m = 1, 2, ..., N$
35: **for** m where $H[P[k-1], m]$ is the leftmost zero in row $P[k-1]$ of H **do**
36: $H[P[k-1], m] \leftarrow P[k]$
37: $P[k] \leftarrow 0$
38: $k \leftarrow k - 1$
39: **go to** EC2
40: **end for**
41: **end if**
42: **end procedure**

43: **procedure** EC5: ADVANCE INITIAL VERTEX
44: **if** $P[1] = N$ **then**
45: **go to** EC6
46: **else**
47: $P[1] \leftarrow P[1] + 1$
48: $k \leftarrow 1$
49: $H \leftarrow 0$
50: **go to** EC2
51: **end if**
52: **end procedure**

53: **procedure** EC6: TERMINATION
54: Terminate the algorithm.
55: **end procedure**

Tarjan (1972) published a new, more efficient, algorithm to enumerate the elementary cycles of a graph. Tarjan (1972, p. 2) also states that Tiernan's algorithm "*explores many more elementary paths than are necessary*". Tarjan's research expands on Herbert Weinblatt's DFS (depth-first search) approach. (Weinblatt, 1972)

Weinblatt's algorithm will traverse each edge only once, whereas Tiernan's algorithm may traverse each edge several times. However, while Weinblatt's algorithm is generally more efficient than Tiernan's, it has an exponential run time in the worst case scenario. Tarjan improves on Weinblatt's algorithm by adding a backtracking procedure, a similar approach to that of Tiernan.

Algorithm 6 Tarjan's Algorithm

```
1: for i = 1 to n do
2:     mark(i) ← false
3: end for
4: for s = 1 to n do
5:     call Tarjan(s,f);
6:     while marked stack ≠ ∅ do
7:         u = top of marked stack
8:         mark(u) ← false
9:         delete u from marked stack
10:    end while
11: end for

12: procedure TARJAN(v, f)
13:    g ← boolean result
14:    f ← false
15:    add v to point stack
16:    v.marked ← true
17:    add v to marked stack

18:    for all w ∈ A(v) do
19:        if w < s then
20:            delete w from A(v)
21:        else if w = s then
22:            output circuit from s → v → s, given in p
23:            f ← true
24:        else if ¬ mark(w) then
25:            call Tarjan(w,g);
26:            f ← f ∨ g
27:        end if
28:    end for

29:    if f = true then
30:        while top of marked stack ≠ v do
31:            u = top of marked stack
32:            delete u from marked stack
33:            mark(u) ← false
34:        end while
35:        delete v from marked stack
36:        mark(v) ← false
37:    end if
38:    delete v from point stack
39: end procedure
```

Tarjan's algorithm requires that the graph's vertices be numbered $1, 2, ...,$ n and that the graph be represented by adjacency lists (see section 2.3.2.2 on p. 13). The adjacency list of a vertex v is represented by $A(v)$. The start vertex of each path p is denoted by s and any vertex v can only be added if $v \geq s$. The pseudocode of Tarjan's algorithm is given in Algorithm 6 on p. 17. Tarjan's algorithm is time bound by $O((V+E)(C+1))$.

Algorithm 7 Johnson's Algorithm (Part 1 of 2)

```
1: A_n ← new int array
2: B_n ← new int array
3: blocked ← new bool array
4: s ← 1

5: while s < n do
6:     A ← adjacency structure of strong component K with least vertex
   in subgraph G induced by s, s+1, ..., n
7:     if A_k ≠ ∅ then
8:         s ← least vertex in V_K
9:         for i ∈ V_K do
10:            blocked(i) ← false
11:            B(i) ← ∅
12:        end for
13:        dummy ← Circuit(s)
14:        s ← s + 1
15:    else
16:        s ← n
17:    end if
18: end while

19: procedure UNBLOCK(u)
20:    blocked(u) ← false
21:    for w ∈ B(u) do
22:        delete w from B(u)
23:        if blocked(w) then
24:            goto Unblock(w)
25:        end if
26:    end for
27: end procedure
```

Johnson (1975) provided an alternative algorithm to find the elementary circuits of a graph. The running time of Johnson's algorithm is faster in the worst case than the algorithms of Tiernan (1970) and Tarjan (1972).

Johnson's algorithm requires a graph G, with vertices labelled $1, 2, ..., n$. (The vertices are represented by these integer values.) For each vertex $v \in V$ the adjacency list $A(v)$ lists the vertices connected to v by an edge. The elementary paths are constructed from a starting vertex, s, and stored

on a stack data structure. When a vertex, v, is appended to a path, it is prevented from being used twice on the same path by "blocking" the vertex. The pseudocode of Johnson's algorithm is given in Algorithms 7 and 8.

Johnsons's algorithm is also time bound by $O((V+E)(C+1))$, but executes faster than the algorithms of Tiernan and Tarjan, due to it considering each edge at most twice per circuit.

Algorithm 8 Johnson's Algorithm (Part 2 of 2)

```
28: procedure CIRCUIT(v)
29:     f ← false
30:     push v on stack
31:     blocked(v) ← true
32:     for w ∈ A(v) do
33:         if w = s then                          ▷ back at start vertex
34:             output stack containing circuit
35:             f ← true
36:         else if ¬ blocked(w) then
37:             if Circuit(w) = true then
38:                 f ← true
39:             end if
40:         end if
41:     end for
42:     if f = true then
43:         goto Unblock(v)
44:     else
45:         for w ∈ A(v) do
46:             if v ∉ B(w) then
47:                 stack v on B(w)
48:             end if
49:         end for
50:     end if
51:     unstack v
52:     Circuit ← f
53: end procedure
```

It is worth mentioning that Hawick and James (2008) extended the circuit enumeration algorithm of Johnson (1975) to include graphs with directed arcs, multiple arcs, and self arcs.

A possible drawback of the previous algorithms is their complexity to implement. Liu and Wang (2006) provide a cycle enumeration algorithm which is easy to implement, but not as efficient as that of Johnson (1975). It is more efficient than the algorithm developed by Tiernan (1970). The adjacency structure of the graph can be either an adjacency matrix or adjacency list. The algorithm by Liu and Wang (2006, p. 2) starts off with an open path (defined as "a simple path not in a cycle"), containing a head

(v_h) and tail (v_t) node. The head and tail nodes can be the same node for self-loops. The path is then extended to vertices not already contained within the path, and with a higher index value than the head node. In terms of efficiency, Johnson's algorithm is the fastest, with Tiernan's being the slowest.

Algorithm 9 Liu & Wang's Algorithm

```
1: procedure LIUWANG(G)
2:     Q ← new queue
3:     enqueue all vertices v_1, v_2, ..., v_n to Q        ▷ each vertex identified by unique number
4:     while Q ≠ ∅ do
5:         P ← open path from Q with head v_h and tail v_t
6:         k ← length(P)
7:         if e = ⟨v_t, v_h⟩ exists then
8:             output P + e as cycle
9:         end if
10:        while not all the edges of v_i have been handled do
11:            if e = ⟨v_t, v_x⟩ exists, where x > h and v_x ∉ e_i ∀ e_i ∈ P then        ▷ h, t, x represent the unique values of the vertices v_h, v_t, v_x
12:                enqueue P + e to Q
13:            end if
14:        end while
15:    end while
16: end procedure
```

2.5 Cycle Picking

In the original paper by Sanders (2013*a*), cycle picking is defined as "an optimisation problem where cycles are chosen from a directed graph under the constraint that any node that is in a cycle in the original directed graph must be in at least one of the chosen cycles". Sanders (2013*a*) showed that while the cycle picking algorithms followed a greedy approach, their heuristic solutions corresponded very closely to the exact solutions; in fact, the heuristic solutions actually provided the exact solutions in most of the cases investigated.

It must be emphasised that cycle picking is simply the optimisation of the cycles in the graph. Redundant cycles are discarded, whilst ensuring that any node previously in a cycle still remains in at least one cycle.

Three cycle picking methods were proposed by Sanders (2013*a*) and each of these methods will be investigated in turn:

- Minimum number of cycles

- All small cycles
- Minimum total cycle length

2.5.1 Minimum Number of Cycles

The minimum number of cycles method produces the combination of cycles containing all the nodes previously contained within cycles whilst also minimising the number of cycles in the combination; that is to say that all the possible combinations of cycles are generated by a cycle enumeration algorithm, and the fewest number of cycles containing all the nodes previously contained within cycles is picked. This allows the maximum number of potential matches possible, but can be a time consuming process, depending on the number of nodes. When all the possibilities have been generated, the algorithm selects the minimum number of cycles with the restriction that any node contained within a cycle(s) in the original graph, must be contained in at least one cycle in the new graph.

Another application of graph matching to consider, which is in principle related to the shoe matching problem, might be the problem of placing dominoes perfectly such that all the dominoes are properly connected, while you are not allowed to flip the dominoes horizontally (otherwise the graph would not be *directed*). Figures 2.5 to 2.7 illustrate the minimum number of cycles method, using a given set of dominoes.

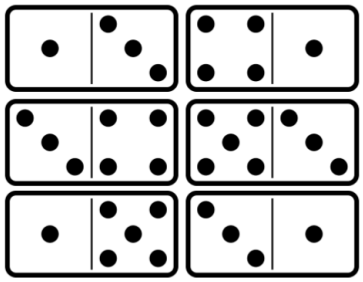


Figure 2.5: Illustrating the minimum number of cycles method: Which cycle permutations are possible?

It is possible to generate several permutations of these cycles; two possibilities are listed below in Figure 2.6.

Figure 2.7 is another permutation, which is also optimal since it makes use of only a single cycle, the fewest number of cycles possible given these dominoes.

2.5.2 All Small Cycles

An alternative method to the minimum number of cycles method, is the all small cycles method. A drawback of the former method is the possi-

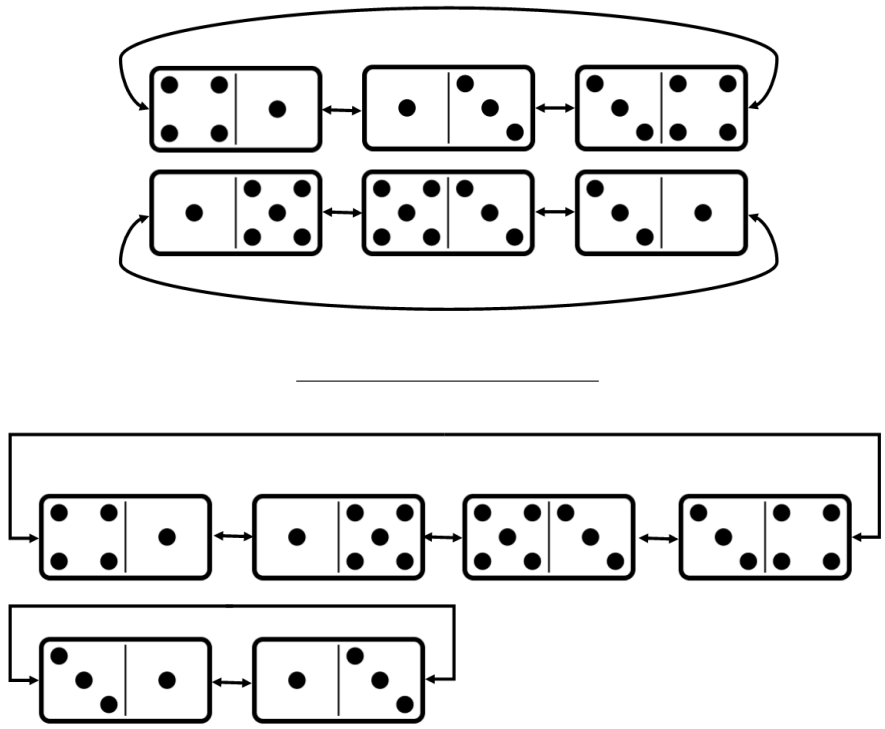


Figure 2.6: Illustrating the minimum number of cycles method: Some possible cycle permutations are given.

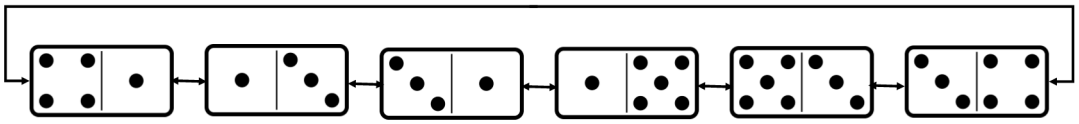


Figure 2.7: Illustrating the minimum number of cycles method: The fewest number of cycles necessary to contain every node.

bility of very long cycles. In the context of the shoe matching problem, long cycles indicate a large group of people working together; this indicates the possibility of extensive organisational overhead for the group of participants. Using the all small cycles method of cycle picking, all of the cycles should be less than some number, say L in length.

A solution to this version of the cycle picking problem is to iteratively consider all the possible cycle lengths of $2, 3, \ldots, L$ until each node is contained within a chosen cycle. In other words, all combinations of cycles of length 2 will be considered. If it is not possible to include all the nodes within cycle combinations of length 2, cycles of length 3 will also be considered, and so forth.

2.5.3 Minimum Total Cycle Length

The last alternative proposed is to minimise the total length of the cycles chosen. The smallest cycle containing the most previously unused nodes is selected and added to the solution. Suppose we have nodes $v_1, v_2, ..., v_n$. The ideal solution to the minimum total cycle length method of cycle picking would be a combination of small, disjoint cycles $(v_1, v_2)...(v_{n-1}, v_n)$.

Sanders (2013*a*) notes in his paper that this method of cycle picking is difficult to relate to the real world shoe matching problem. There is also the possibility that this method will simply favour small cycles which is, in itself, not an unreasonable approach.

2.5.4 Conjectured NP-completeness

Sanders (2013*a*) conjectured that cycle picking is an NP-complete problem, but provided no formal proof. No formal proof could be found from an alternative source either. This dissertation chooses to assume that cycle picking is NP-complete and that a combinatorial algorithm must be used to evaluate each possible combination of cycles. For the purposes of this dissertation, the r-combinations algorithm of Johnsonbaugh (2000) was used.

2.6 Graph Augmentation and Enlargement

The foundations of graph augmentation are already established; common problems in graph theory often involve determining how many edges or vertices must be **removed** from a graph to satisfy some connectivity property, for example, the property of biconnectedness. However, Eswaran and Tarjan (1976) investigated the opposite: how many vertices or edges should be **added** to a certain graph to satisfy a particular connectivity property, say C. It is assumed that C is monotone increasing, but not necessarily monotone decreasing (this is indeed the case for some common connectivity and planarity properties). Eswaran and Tarjan (1976) applied a cost (weight) to all the edges of a graph, say G, by defining a real-valued function $f(v,w)$ for any edge $(v,w) \in E(G)$. Furthermore, $\varepsilon_0 = \{(v,w) | v,w \in G, f(v,w) = 0\}$ and the initial state of the graph to be augmented was given by $G = (V, \varepsilon_0)$. The augmentation problem, as defined by Eswaran and Tarjan (1976) is thus about finding a set of edges, ε, such that $G = (V, \varepsilon_0 \cup \varepsilon)$ satisfies C, the connectivity property, whilst also minimising the cost function, $\sum_{(v,w) \in E(G)} f(v,w)$.

Eswaran (1973) investigated the problem of adding a minimal-cost set of edges to a digraph, G, such that there is some cycle $C \in G$, which contains

all the edges of G; an efficient solution to the problem is also provided. Goodman and Hedetniemi (1974); Goodman et al. (1975) investigated the minimum number of additional edges necessary to transform a graph into a Hamiltonian graph. They first provided an $O(V^2)$ algorithm to solve the problem, but later improved it to $O(V)$. Goodman and Hedetniemi (1974) defined $hc(G)$, the *Hamiltonian completion number* of a connected graph, G, as the minimum number of edges necessary to be added to G to ensure that G is Hamiltonian. A Hamiltonian graph is a graph which contains a Hamiltonian cycle: a cycle which visits each of the nodes in the graph exactly once. It was proved that $hc(G) = \min_{T_i \in S} hc(T_i)$ where S is the set of spanning trees of G, and T is an arbitrary tree in S.

Frank and Chou (1970) presented a solution to a more generalised augmentation problem: given a $V \times V$ symmetric matrix $[r_{ij}]$ with $r_{ii} = 0 \ \forall i$, they ask for an undirected graph, G, with a minimal number of edges on the vertex set $V(G)$, such that there are at least r_{ij} edge-disjoint paths between vertices i and j. A set of paths is defined to be edge-disjoint if no edge is common between any two paths. Frank and Chou (1970) focuses on the survivability of networks against enemy attacks by modelling the network as an undirected graph with a minimum number of edges such that the graph contains no parallel branches. Parallel or multiple edges are two or more edges incident to the same vertices. In network construction, this means that multiple edges (network paths) will fail if one of the incident nodes happens to fail. The algorithms of Frank and Chou (1970) are complex but can be applied to graphs with hundreds of nodes. In a nutshell, the algorithms focus on the rearrangement of edges (network paths) to avoid cascading network failures. This is achieved by providing redundant parallel edges between network nodes.

Graph augmentation algorithms have been used extensively to increase the edge connectivity of graphs. Rosenthal and Goldner (1977) published a linear-time algorithm to augment graphs in order to increase their connectivity to biconnected. However, Sheng Hsu and Ramachandran (1993) exposed a flaw in the original algorithm and published an updated algorithm, running in $O(\log^2 n)$ time. Naor et al. (1997) published an algorithm to increase the edge connectivity of an undirected, unweighted graph from its existing λ-connectedness to being δ-connected. The aforementioned results can be practically implemented in the fields of network synthesis (Frank, 1990), for example. Networks are generally represented by undirected graphs due to the full duplex (simultaneous bi-directional) flow of information.

The addition of nodes to a graph model is common in network infrastructure to enhance redundancy. In Figure 2.8, the "Fully Connected" network is the most redundant. The failure of a single node will not affect the other nodes in the network. However, this level of redundancy is only available at the cost of additional cabling between all the different nodes. The "Mesh" network model is also a fairly redundant model, while also being

more affordable than the "Fully Connected" model. The "Line", "Ring", and some other topological models are very poor examples of redundancy. The failure of a single node will at best split the network into different sections, unable to communicate with one another across this virtual gap.

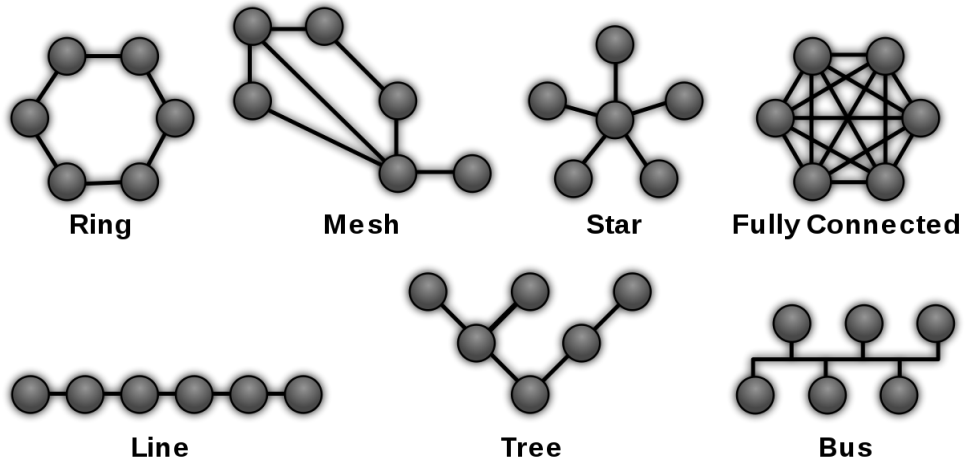


Figure 2.8: Illustrating some common network topologies
(Image source: http://commons.wikimedia.org/wiki/File:NetworkTopologies.png)

By representing networks as undirected graphs, it is possible to increase the redundancy (and thus the reliability) of a network by adding extra nodes (access points) to the network. These redundant nodes do not serve any purpose other than providing failover links when the primary nodes fail. Egeland and Engelstad (2009) focused on increasing the redundancy of a **wireless mesh** network whilst also taking the economic (monetary) constraints of adding nodes into account. Mesh networks are considered to be *ad hoc* networks and each node participates in the distribution and routing of data. In addition, they note that the additional cost of adding redundant nodes is easy to forecast, while the additional reliability provided by the redundant nodes is not; this is mainly due to a lack of literature on the subject. By considering a mesh network G, with distribution nodes $d_i \in D = (d_1, d_2, ..., d_{k-1})$ and a single root node r (giving a total of k nodes), the k-terminal reliability is given by the probability formula:

$$P_c^{r,d_1,d_2,...,d_{k-1}}(G) = 1 - \sum_{i=\beta}^{\varepsilon} C_i^{r,d_1,d_2,...,d_{k-1}}(p)^i(1-p)^{\varepsilon-i}$$

The k-terminal reliability formula is defined as the probability that a path exists and connect k nodes in a given network. $C_i^{r,d_1,d_2,...,d_{k-1}}$ represents the number of edge cutsets of cardinality i, whilst p denotes the probability of a link being offline. p is later given as $p = \frac{\lambda}{\mu+\lambda}$ where λ is the failure rate parameter and μ the repair rate parameter. Egeland and Engelstad (2009) advise network planners to use the k-terminal reliability formula

to analyse the reliability of their networks. The graphs for wireless mesh networks like these are undirected, and the placement of redundant distribution nodes occur foremostly at the vulnerable points in the network. While being a rare example of graph enlargement via vertex addition, the network redundancy scenario is very different from the matching problem faced in this dissertation.

Xulvi-Brunet and Sokolov (2007) explore a scenario very similar to that of Egeland and Engelstad (2009): the enlargement of a network while taking geographical constraints into account. These networks are generally referred to as spatial or geographical networks. Networks are often embedded in physical space. Transport infrastructure, such as railways, or the electricity grid are part of these types of networks in physical space. Even computer networks, and especially wide area networks such as the internet, are also subject to these constraints. The cost of establishing long distance connections between two distant locations is generally higher than establishing shorter connections between closer connections. Long distance connections also tend to exist between the most highly connected nodes in a network, for example long distance railways are normally built between the largest cities. Large cities also serve as central transport hubs for smaller cities and towns. Xulvi-Brunet and Sokolov (2007) go on to simulate the growth of geographical networks with a probabilistic model, mainly focusing on the unique and sometimes peculiar **structures** of real world spatial networks, such as railway lines. This is again a rare example of graph enlargement via vertex addition, but in a different scenario and for a different purpose.

Due to the fact that previous research on graph augmentation mainly focuses on **edge augmentation**, most of the results are not directly applicable to this dissertation, but instead demonstrate an opportunity to extend the field of graph augmentation / enlargement; this dissertation therefore focuses on **graph enlargement via vertex addition**.

Graph enlargement in this dissertation shall be defined as:

Instance: Given a digraph, $G(V,E)$ such that each node, $v \in V$ is associated with a pair of values $(v_1, v_2) \in (\mathbb{Z}^+ \times \mathbb{Z}^+)$. There is an edge uv in E if and only if $u_1 = v_2$.

Problem: Find a set of nodes V' and associated edges E', such that $G_E(V_E, E_E)$ satisfies the property $P \leftarrow$ *all nodes in* V_E *are contained within cycles*. Here $V_E = (V \cup V')$ and $E_E = (E \cup E')$; any node, $w \in V_E$ has a pair of values associated with it $(w_1, w_2) \in (\mathbb{Z}^+ \times \mathbb{Z}^+)$; and for any two nodes $w, z \in V_E$ an edge wz is in E_E when $w_1 = z_2$).

The cost of the graph enlargement can be minimised by minimising the total number of nodes required (all nodes have equal cost/weight).

In Chapter 4, the number of total nodes required has been reduced by reducing the unique number of dummy nodes required, and utilising inno-

vative techniques such as cycle compression, to compress repeated nodes within multiple cycles.

2.6.1 Sanders's Graph Enlargement Algorithm

With regards to the shoe matching problem, Sanders (2013*a*) makes provision for an alternative solution in the event that a person cannot find a partner (or a group) to cooperate and exchange shoes with. Abstracting this situation, into one from a graph theory perspective, means that the node (representing the aforementioned person) is not in a cycle. There are four reasons for this:

- The node is completely isolated
- The node has only incoming edges
- The node has only outgoing edges
- The node only serves as a bridge between two other nodes

The graph can be enlarged with dummy nodes, which represent extra pairs of shoes bought by the group to fulfill everyone's needs. This allows everyone to still save money overall, whilst also satisfying the footwear needs of every member of the group. Sanders (2013*a*, see pp. 10-12) provides a pair of algorithms to enlarge the graph. The first algorithm handles cases 1, 2, and 3, while the second algorithm handles the last case. See Algorithms 10 (p. 28) and 11 (p. 29) for the original pair of algorithms.

After the algorithms *firstPass* and *secondPass* have been applied to the graph, every node (representing both real people and dummy pairs) will be contained within a cycle.

Sanders's cost distribution model places the cost burden of the dummy nodes on the participants **by cycle**. In other words, a cycle of 5 participants and a single pair of dummy shoes will cost each participant of that cycle the value of $1 + \frac{1}{5}$ pairs of shoes. If there is a second cycle in the graph with 2 participants and 1 pair of dummy shoes, then each participant will pay for $1 + \frac{1}{2}$ pairs of shoes. In Section 4.2.1, the cost distribution definition was revisited to produce a cost distribution method which is fairer to smaller cycles, without affecting larger cycles too negatively. In essence, the cost is evenly distributed across all the participants in the graph, and not per cycle.

2.6.2 Sanders's Algorithm - A Worked Example

Suppose we have the graph G as illustrated in Figure 2.9.

The only cycle currently present in the graph is:

Algorithm 10 Sanders's Algorithm (Part 1 of 2)

```
1: procedure FIRSTPASS(G, cycles)
2:     Copy G to newGraph
3:     dis ← set of all isolated nodes
4:     no_in ← set of all nodes with no incoming edges
5:     no_out ← set of all nodes with no outgoing edges
6:     use ← set of all other nodes
7:     while dis ≠ ∅ ∨ no_in ≠ ∅ ∨ no_out ≠ ∅ do
8:         first ← a node from no_out (undefined if none available)
9:         last ← a node from no_in (undefined if none available)
10:        mid ← a node from dis (undefined if none available)
11:        if mid is defined then
12:            if first is undefined then
13:                first ← any node ∉ no_out, no_in, dis
14:            end if
15:            if last is undefined then
16:                last ← any node ∉ no_out, no_in, dis
17:            end if
18:            create two new nodes: dummy1, dummy2
19:            create path first → dummy1 → mid → dummy2 → last in
    newGraph
20:        else
21:            if first is undefined then
22:                first ← any node ∉ no_out, no_in, dis
23:            end if
24:            if last is undefined then
25:                last ← any node ∉ no_out, no_in, dis
26:            end if
27:            create a new node: dummy3
28:            create path first → dummy3 → last in newGraph
29:        end if
30:    end while
31:    return newGraph
32: end procedure
```

- (1: Adam) → (4: David) → (2: Bob)

The nodes with no incoming edges are:

- (7: George)
- (8: Harry)

All the nodes of the graph have outgoing edges. It follows, but can be easily confirmed, that the graph contains no isolated nodes. In terms of

Algorithm 11 Sanders's Algorithm (Part 2 of 2)

```
33: procedure SECONDPASS(G, cycles)
34:     Copy G to newGraph
35:     cycles ← all nodes which are in cycles
36:     nocycles ← all nodes which are not in cycles
37:     while nocycles ∉ ∅ do
38:         x ← any node ∈ nocycles
39:         B ← x
40:         while B ∈ nocycles do
41:             new B ← node at the other end of an edge originating from
    B
42:         end while
43:         E ← a node with an edge to x
44:         while E ∈ nocycles do
45:             new E ← node at the other end of an edge terminating at E
46:         end while
47:         create a new dummy node: D
48:         create path B → D → E in newGraph
49:         update cycles, nocycles appropriately
50:     end while
51:     return newGraph
52: end procedure
```

the pseudocode for Sanders's Algorithm we therefore have that $no\_in \leftarrow$ {(7: George), (8: Harry)}, whilst both $no\_out \leftarrow \emptyset$ and $dis \leftarrow \emptyset$.

After applying Sanders's algorithm (there were no bridge nodes after the completion of the *firstPass* process), we have the result illustrated in Figure 2.10.

The cycles now present in the graph are:

- (1: Adam) → (4: David) → (2: Bob)
- (1: Adam) → (4: David) → (2: Bob) → (10: Dummy) → (8: Harry) → (6: Frank) → (3: Carol)
- (1: Adam) → (9: Dummy) → (7: George) → (5: Eddie) → (2: Bob)
- (1: Adam) → (9: Dummy) → (7: George) → (5: Eddie) → (2: Bob) → (10: Dummy) → (8: Harry) → (6: Frank) → (3: Carol)
- (1: Adam) → (9: Dummy) → (7: George) → (6: Frank) → (3: Carol)
- (2: Bob) → (10: Dummy) → (8: Harry) → (5: Eddie)

Note that every node within the graph now has both an incoming and outgoing edge. After applying the minimum number of cycles method of

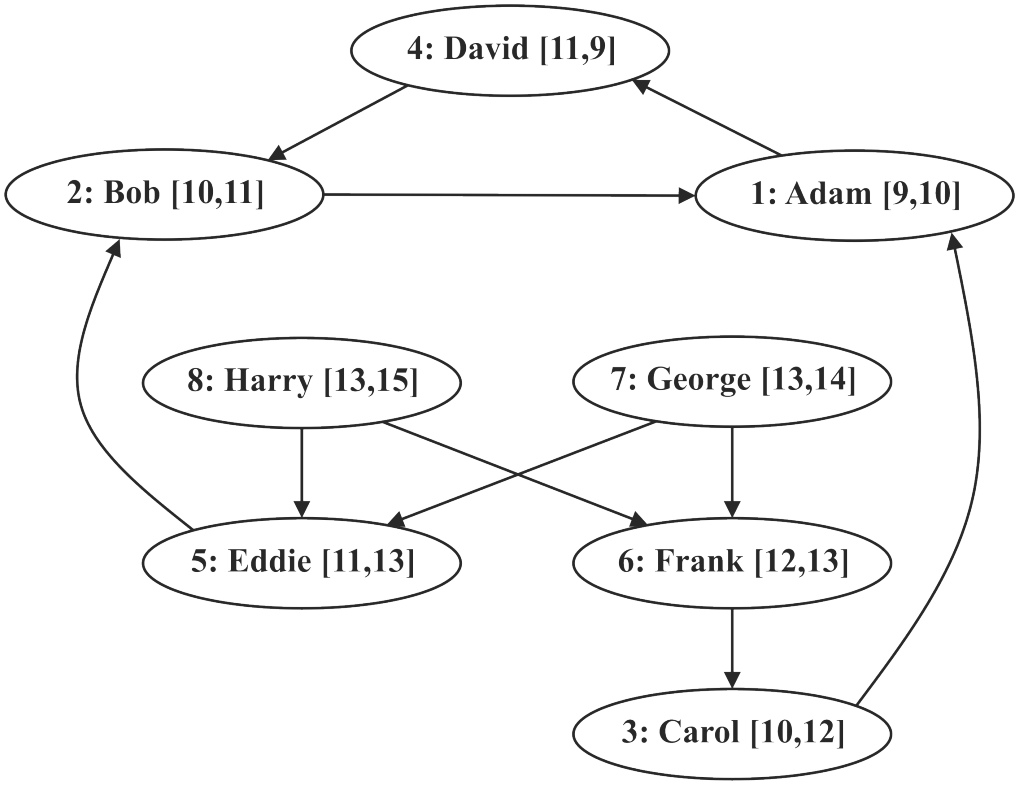


Figure 2.9: The graph G to which Sanders's algorithm will be applied

cycle picking, we have the following solution to this specific instance of the shoe matching problem:

- (1: Adam) → (4: David) → (2: Bob) → (10: Dummy) → (8: Harry) → (6: Frank) → (3: Carol)
- (1: Adam) → (9: Dummy) → (7: George) → (5: Eddie) → (2: Bob)

Cost distribution, as per Sanders's definition, would be as follows: for the first cycle each participant pays for $1 + \frac{1}{6}$ pairs of shoes, whilst each participant in the second cycle pays for $1 + \frac{1}{4}$ pairs of shoes.

2.6.3 Theoretical Runtime of Sanders's Algorithm

For a given algorithm, it is possible to calculate the asymptotic upper bound of the running time, depending on the size of the input. This upper bound is expressed in O-notation and refers to the worst case running time of the algorithm. (Cormen et al., 2009)

In the *firstPass* algorithm (p. 28) lines 7 to 30 are executed n times, given n number of nodes in $dis \cup no\_in \cup no\_out$. The firstPass algorithm theoretically runs in $O(n)$ time.

The *secondPass* algorithm runs in $O(2n^2)$, derived as follows:

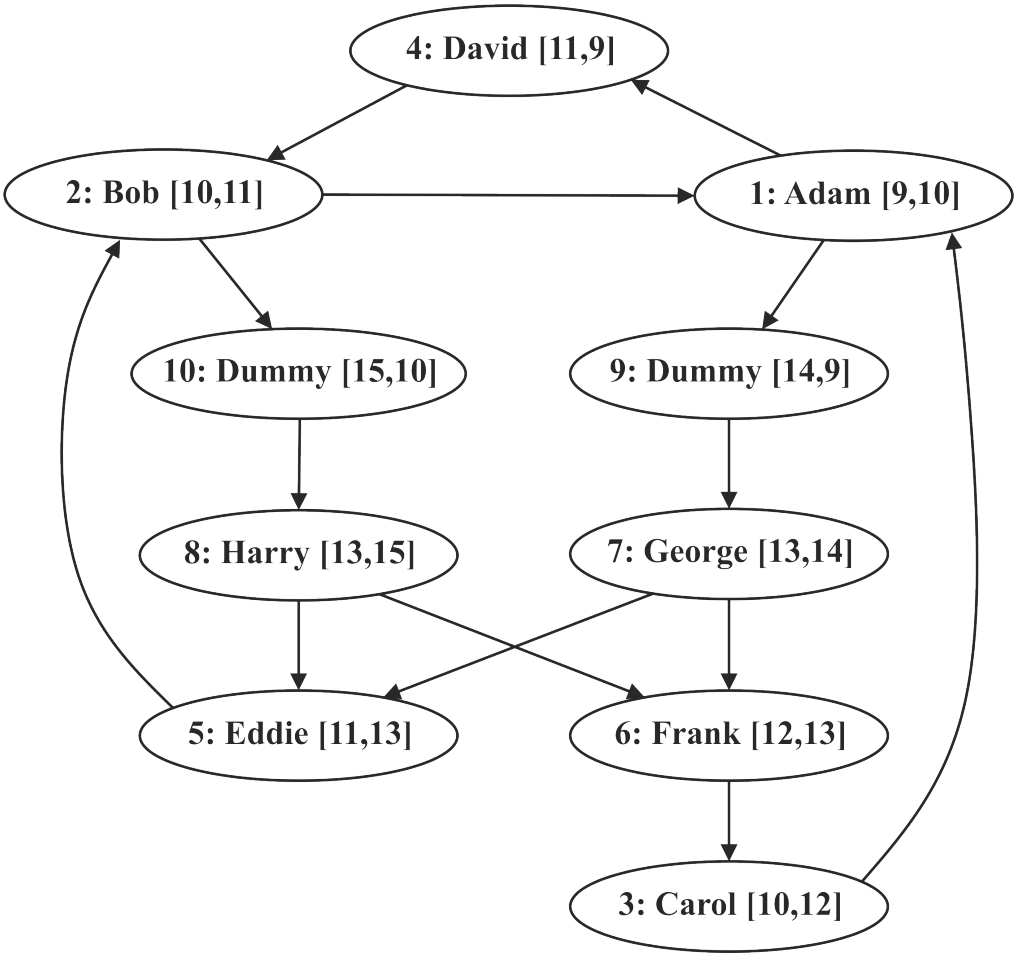


Figure 2.10: The graph G after Sanders's algorithm has been applied

- line 36: *nocycles* contains n nodes not contained within cycles
- line 37: outer while loop is executed n times in the worst case (only one node is attached to a cycle per iteration)
- lines 40 - 42: inner while loop is executed n times in the worst case (B iterates through all of the nodes in *nocycles* before finding a suitable node)
- lines 44 - 46: inner while loop is executed n times in the worst case (E iterates through all of the nodes in *nocycles* before finding a suitable node)

The dominant terms for the running time of Sanders's entire algorithm is therefore $T(n) \in O(n^2) + O(n^2) + O(n) \implies T(n) \in O(2n^2)$.

2.7 Summary

In this chapter basic graph theory concepts were introduced, as well as the more advanced topics of cycle enumeration, cycle picking, and graph augmentation / enlargement.

The purpose and usage of these topics are clarified in Chapter 3, which expands on the research methodology and overall strategy employed for the purposes of this dissertation. Cycle enumeration and cycle picking form the backbone of the cost-optimised algorithm, which also builds on the Hopcroft-Tarjan connected components algorithm and Sanders's graph enlargement algorithms. In the next chapter the specifics of these algorithms, as utilised in this dissertation, are examined in greater detail.

Research papers on graph augmentation have been preoccupied with edge augmentation, introducing an opportunity for this dissertation to focus on vertex addition to graphs. Chapter 4 introduces some new enlargement techniques to improve upon previous work done by Sanders (2013*a*), by utilising (or improving) existing procedures, as covered in this chapter.

Chapter 3

Methodology

Contents

3.1 Introduction

This chapter presents the research aim and examines the auxiliary functions necessary to proceed with graph enlargement. This includes the generation of synthetic data, cycle enumeration algorithms, and cycle picking algorithms. Pseudocode implementations of important auxiliary functions are also presented, including that of cycle picking.

The abovementioned algorithms are all necessary to achieve the research aim of developing an improved graph enlargement algorithm to the shoe matching problem. The cycle enumeration algorithms determine which nodes are not contained within cycles (and which are). The graph enlargement algorithms include these orphaned nodes as part of the existing cycles, and the cycle picking algorithm chooses an optimal combination of cycles with regards to node-cost.

Furthermore, the overall research strategy, research aim, and justification for important decisions made regarding the dissertation (such as the programming language of choice and choice of cycle enumeration algorithm, for example) are presented. Finally, the chapter examines the assessment criteria necessary for determining whether or not the research aim was achieved.

3.2 Research Aim

In section 2.6 (p. 23) Algorithms 10 & 11 were presented; originally developed by Sanders (2013*a*), for the purpose of graph enlargement. As mentioned in Chapter 1 (p. 1), this algorithm is naïve, but it does enlarge the graph correctly. The research aim is therefore:

> **To develop an improved algorithm(s) for the graph enlargement problem**; in terms of:
>
> - Faster run time and the capability to handle larger graphs
> - Fewer total nodes required to enlarge the graph such that all nodes are contained within cycles

The primary aim of this research, in essence, is an improvement to the algorithm of Sanders (2013*a*).

3.3 Overall Strategy

It was vital to first understand the existing enlargement algorithms and other ideas developed by Sanders (2013*a*), as described in Section 2.6.1 (p. 27). This also includes Sanders's ideas on cycle picking, among others. Sanders's original algorithm was then implemented from the pseudocode provided, in order to run comparative tests between the original algorithm and the new algorithms.

It was not possible to source academic literature or data on shoe sizes, due to a lack of literature on the subject. Sanders (2013*a*) encountered the same difficulty and decided to generate synthetic data. The same decision was taken for this dissertation, and more information on the generation of the synthetic data is available in Section 3.5 (p. 35).

Experimental work and ideas were implemented after the previous steps were completed. Experiments to test different optimisations, each with its own advantages and disadvantages, were carried out. This was an iterative process; new algorithms and subprocesses were devised and implemented, and then compared to Sanders's original algorithm. The com-

parisons were done using the exact same data sets to ensure scientific integrity and rigour. If the comparison between the original and new algorithm was found to be in favour of the new algorithm, the new ideas were documented and definitively implemented as part of the algorithm. If the comparison was **not** favourable the ideas were either discarded or, if possible and worthwhile, negative effects were negated by implementing processes to counter their disadvantages (see Section 4.2.5, p. 54 on "Cycle Compression", for example). The steps described in this paragraph were repeated as necessary to develop the new algorithms.

The results were assessed (see Section 3.10, p. 39 for more information on the assessment criteria) according to predetermined criteria and reported in detailed tabular format (see Sections 4.2, 4.3, and 4.4; Chapter 5; and Appendix A).

3.4 Programming Language

The first major operational decision of the research involved choosing an appropriate programming language to proceed with. **Python 2.7** was chosen for the following reasons:

- Python is easy to learn and use.
- Python code is robust, flexible, and easily legible.
- Python is open source and free.
- Python has built-in support to not only handle lists, but also mathematical sets.
- The source code in this dissertation can be easily transferred to different operating systems and platforms, e.g. Linux on the Raspberry Pi (powered by an ARM-architecture processor).

3.5 Finding Appropriate Data

Once the programming language was decided upon, data on shoe sizes had to be researched, collected and generated. Due to a lack of academic literature, synthetic data had to be generated from certain assumptions to proceed with the research.

The following assumptions were made:

- Gender and style differences are not considered. A different graph could be constructed for each gender and style combination.

- Average shoe sizes range from 9 to 11, with a standard deviation of 2. Shoe sizes therefore range between 7 to 13.
- Once the left shoe size is generated the right shoe size is derived from it by adding a random value between -2 to 2 to the left shoe size.

Due to the lack of real-world data, synthetic data was generated on the abovementioned assumptions.

3.6 Generating Synthetic Data

As mentioned in Section 3.5, no academic literature or real-world examples were available for use, and synthetic data had to be generated. Algorithm 12 lists a pseudocode implementation of the synthetic data generator.

The **synthetic data** was randomly generated; however, a randomisation seed could be set to keep the generated numbers constant. A list of shoe size pairs were generated and then converted into an adjacency list (for more on adjacency lists, see Section 2.3.2.2, p. 13). Nodes were considered adjacent (connected via an edge) if the left shoe size of the first pair equaled the right shoe size of the next pair. That is to say, for any vertices $u, v \in G$ where any vertex $v = (x, y)$ where $x, y \in \mathbb{Z}^+$, the edge uv is automatically created when $u_x = v_y$. Graph enlargement, in this dissertation, occurs via adding **vertices**, instead of edge augmentation. (See Section 2.6 on p. 23 for further background.)

3.7 Cycle Enumeration

Cycle enumeration, or cycle detection as it is also known, was first attempted using the algorithm of Tiernan (1970). However, Tiernan's algorithm was not efficient enough for the purposes of this research. Recursion depth errors were common when the number of nodes in the graph exceeded approximately 12 nodes. It was possible to increase the stack recursion depth in the Python code, but this led to system instability in return for virtually no improvement: the number of nodes in the graph could be increased to about 14 before recursion depth errors occurred once more. The algorithm of Tarjan (1972) was decided upon for use in this dissertation; the recursion depth errors were no longer encountered and performance was greatly improved. It would have been possible to further improve the performance of the cycle detection function by using the algorithm of Johnson (1975), but Tarjan (1972) already proved efficient enough for the purposes of this dissertation and struck a good balance between complexity and efficiency. Ultimately, the aim of this dissertation was to

Algorithm 12 Generating Synthetic Data and the Adjacency List

```
1: procedure SYNTHETICDATA(N)
2:     seed ← randomisation seed of your choice
3:     shoes ← ∅                                   ▷ Array of all shoe pairs
4:     for i = 1 to N do
5:         left ← random value between [9, 11]
6:         left ← left + random deviation between [-2, 2]
7:         right ← left + random deviation between [-2, 2]
8:         shoes.add([left, right])
9:     end for
10: end procedure

11: procedure ADJACENCYLIST(seq)
12:     seq ← shoes          ▷ The sequence consists of the shoe pairs array
13:     N ← length(seq)
14:     G ← ∅
15:     for i = 1 to N do
16:         for j = 1 to N do
17:             if i ≠ j and seq[i][1] = seq[j][2] then     ▷ left shoe matches a
    different node's right shoe
18:                 G[i].add(j)                             ▷ j is an adacency to i
19:             end if
20:         end for
21:     end for
22: end procedure
```

find an efficient graph enlargement algorithm and the cycle detection algorithm is simply a means to an end. For more on Tiernan and Tarjan's cycle detection algorithms, see Section 2.4, p. 14.

3.8 Cycle Picking

Cycle picking (see Section 2.5 on p. 20) is an optimisation problem. A cycle enumeration algorithm may return several cycles in which some node appears. The number of cycles can be optimised by selectively picking cycles with the constraint that any node which was in a cycle (or cycles) before, must remain in at least one cycle.

The cycle picking algorithms were derived from the original examples of Sanders (2013*a*). The minimum number of cycles method of cycle picking was implemented for this dissertation. The process of cycle picking is conjectured to be NP-complete (Sanders, 2013*a*) and therefore a combinatorial approach was followed to find the optimal arrangement of cycles whilst keeping the number of cycles to a minimum. While this approach

does hamper performance, it ensures the efficacy of the cycle picking algorithm.

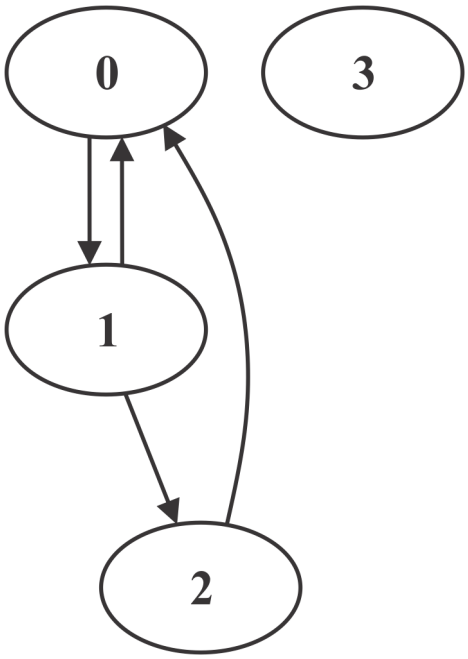


Figure 3.1: A simple directed graph to illustrate cycle picking

In Figure 3.1 the following cycles can be found:

- Cycle 1: $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$
- Cycle 2: $0 \rightarrow 1 \rightarrow 0$

For example, if one applies the minimum number of cycles picking method to the cycles in Figure 3.1, one can safely discard Cycle 2, since nodes 0 and 1 are already contained within Cycle 1.

3.8.1 Minimum Number of Cycles Implementation

The minimum number of cycles picking method produces an optimal combination of cycles by selecting a minimal number of cycles, while also preserving nodes already contained within cycles. The only input the algorithm requires is a list of all the cycles detected by a cycle enumeration algorithm, such as Tiernan (1970), Tarjan (1972), Johnson (1975), or Liu and Wang (2006). The pseudocode for the minimum number of cycles implementation is given in Algorithm 13.

Please see Section 2.5.1 (p. 21) for a background of the minimum number of cycles method of cycle picking.

Algorithm 13 Cycle Picking: Minimum Number of Cycles

```
1: procedure MINIMUMNUMBEROFCYCLES(AllCycles)
2:     r-combs ← ∅
3:     for r ≤ len(AllCycles) do
4:         Evaluate all r-combinations of AllCycles
5:         if any r-combination contains all nodes currently in cycles then
6:             add current r-combination to r-combs array
7:         end if
8:     end for

9:     for all r-combinations ∈ r-combs do
10:        select combination with least number of nodes    ▷ to lower the
   total node cost
11:    end for

12:    return combination with least number of nodes
13: end procedure
```

3.9 Enlarging Cycles

The graph enlargement algorithms were first replicated from the pseudocode of Sanders (2013*a*) for extra insight into the problem and the previous solution. This also assisted in identifying possible shortcomings of the original algorithm, as well as opportunities to recognise improvements which could be applied the algorithm. Background information on cycle enlargment is given in Section 2.6 (p. 23).

Experimentation with certain optimisations were undertaken, namely:

- Cost - reducing the number of dummy nodes. (see Section 4.2)
- Speed - reducing the running time necessary. (see Section 4.3)
- Input size - by enlarging a subgraph consisting of only nodes not contained within cycles. (see Section 4.4)

3.10 Assessment Criteria

The assessment criteria for the efficiency of the algorithm could be considered subjective, depending on the needs of the person implementing the algorithm. The algorithm could optimise either one of the following:

- **Cost** - fewer dummy nodes would mean less monetary cost to the participants. Compare the solutions in Figures 3.2 and 3.3: the original solution contained 2 dummy nodes, and 10 nodes in total, whilst

the cost-optimised solution contained 1 dummy node and 9 nodes in total. The problem of repeated nodes, due to fewer unique dummy nodes, was also encountered, but cycle compression proved to be an efficient counter to this problem (see Section 4.2.5 on p. 54).

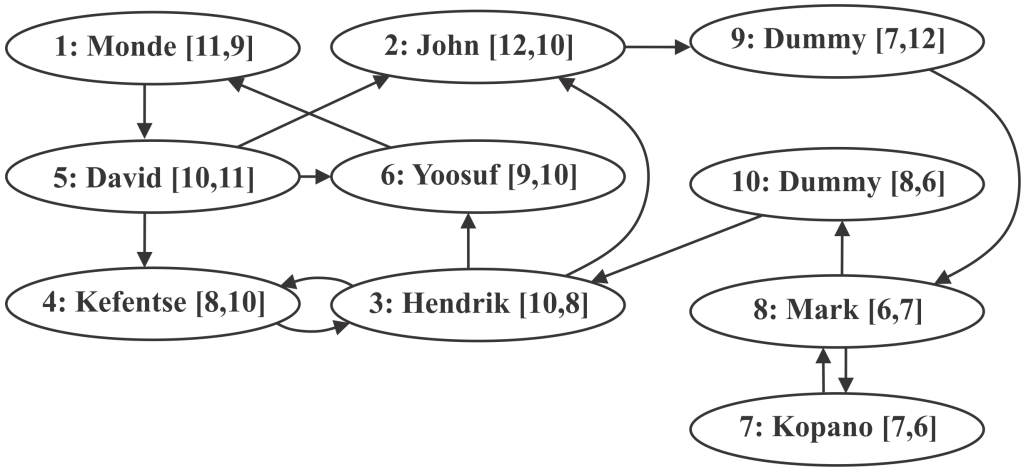


Figure 3.2: Sanders's original solution to the shoe matching problem (2 dummy nodes, 10 nodes in total)

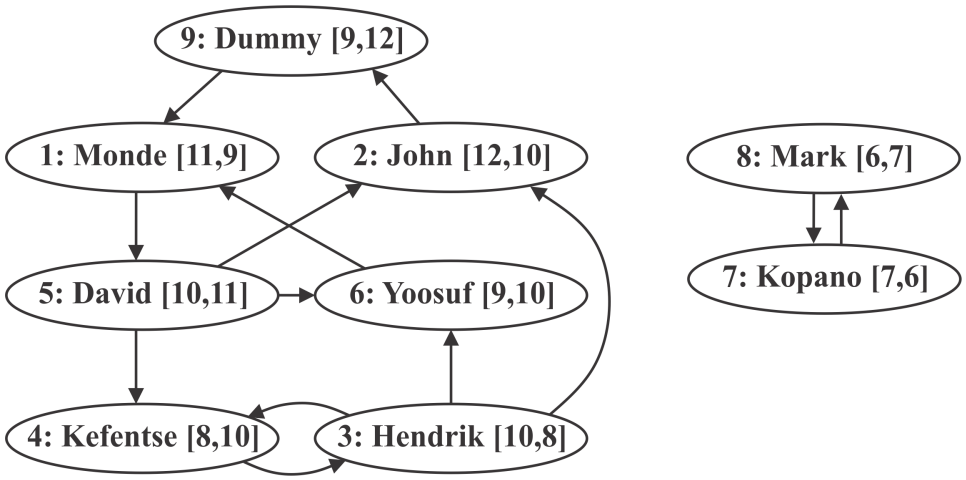


Figure 3.3: A cost-optimised solution (only 1 dummy node, 9 nodes in total)

- **Speed** - from a technical or academic perspective, one might be interested in the fastest possible solution. A matrix-based graph enlargement method is presented in Section 4.3 on p. 67, with greatly improved running times over Sanders's original algorithm.

- **Input size** - by enlarging only a subgraph of the original graph. If most of the nodes within the graph are contained within cycles, can we simply create and enlarge a subgraph of the nodes not contained within cycles? (see Section 4.4 on p. 81).

In the case of the shoe matching problem, minimising the cost per participant would be ideal, but in a situation where cost is an irrelevant or abstract notion, the speed-optimised algorithm will be a better fit.

3.11 Summary

This chapter provided an overview of the research methodology and strategy, with the aim of developing a new, efficient algorithm for the purposes of graph enlargement.

The concepts regarding the implementation of several auxiliary functions were also covered, including that of cycle enumeration and cycle picking.

The next chapter presents these strategies and concepts implemented experimentally; the algorithms are compared to Sanders's original algorithm, and detailed results are tabulated for comparison purposes.

Chapter 4

Experimental Work

Contents

4.1 Introduction

The primary aim of this dissertation is an improvement on the original graph enlargement algorithm of Sanders (2013*a*) to solve a specific graph matching problem, namely the shoe matching problem.

Experimental research ideas and the results thereof are presented in this chapter. The following optimisation methods are investigated, as previously detailed in Section 3.10:

- Cost - fewer dummy nodes would mean less monetary cost to the participants.
- Speed - from a technical or academic perspective, one might be interested in the fastest possible solution.
- Input size - by enlarging only a subgraph of the original graph. If most of the nodes within the graph are contained within cycles, can we simply create and enlarge a subgraph of the nodes not contained within cycles?

4.2 Cost Optimisation

For the original shoe matching problem, the ideal solution would be to provide everyone with a pair of shoes at the least possible cost. If dummy pairs of shoes are necessary, the group would ideally want to purchase as few extra pairs as possible. The following strategies were proposed to help save on monetary cost:

- Redefining cost distribution
- Handling graphs with only strictly isolated nodes differently
- Avoiding bridge nodes
- Compressing repeated nodes, i.e. nodes that occur in multiple cycles

4.2.1 Redefining Cost Distribution

In the original graph enlargement example (Sanders, 2013*a*), the monetary costs for the shoes were distributed per cycle, that is to say the cost of adding dummy nodes to a particular cycle would be carried by the people (nodes) who were already part of the cycle. This leads to the following consequence:

Smaller cycles carry higher costs per node (person) than the average node, if dummy nodes need to be added. Isolated nodes and small cycles are due to certain people not being able to find matches, or simply finding fewer matches. This could be due to a large difference in the size of their feet (≥ 3 sizes), or simply because their shoe size is generally uncommon (e.g. size 14 shoes). The original cost distribution method penalised people for something outside of their control; a better method of cost distribution would be if everyone involved split the cost of all the necessary pairs, equally.

Sections 4.2.2 and 4.2.3 are **dependent** on the acceptance of this new definition for cost distribution.

4.2.2 Graphs Containing Only Strictly Isolated Nodes

In case the graph contains **only** strictly isolated nodes, but **does not** contain any nodes that have only incoming or outgoing edges (see Figure 4.1), it is unnecessarily expensive to add two dummy nodes (see Figure 4.2) for the sake of integrating a few (or even only one) isolated nodes into an already existing cycle. It would be more cost effective to simply furnish the isolated node (see Figure 4.3) with one extra pair of shoes.

It is important to note that the alternative presented in Figure 4.3 is only applicable to situations where the graph contains only strictly isolated nodes. If the graph **does** contain nodes that simply have no incoming or outgoing edges, it would be better to use the isolated node as a midpoint between a node with no incoming edge and a node with no outgoing edge (Sanders's original solution for all isolated nodes). This method also relies on the proposed change to the distribution of costs (see Section 4.2.1), otherwise one person would be paying for two pairs of shoes, which is precisely the situation we are trying to avoid.

Sanders's algorithm could therefore be modified as follows: if **first** and **last** are undefined, then define **mid** as the isolated node and connect it to a single dummy node. Since everyone involved shares the cost of all the extra shoes purchased, the previously isolated person does not pay for two pairs of shoes, instead the group as a whole carries the cost of buying one extra pair of shoes. The average cost per person would still be less for the solution in Figure 4.3 than for the solution in Figure 4.2, due to the fact that only one extra pair of shoes is necessary.

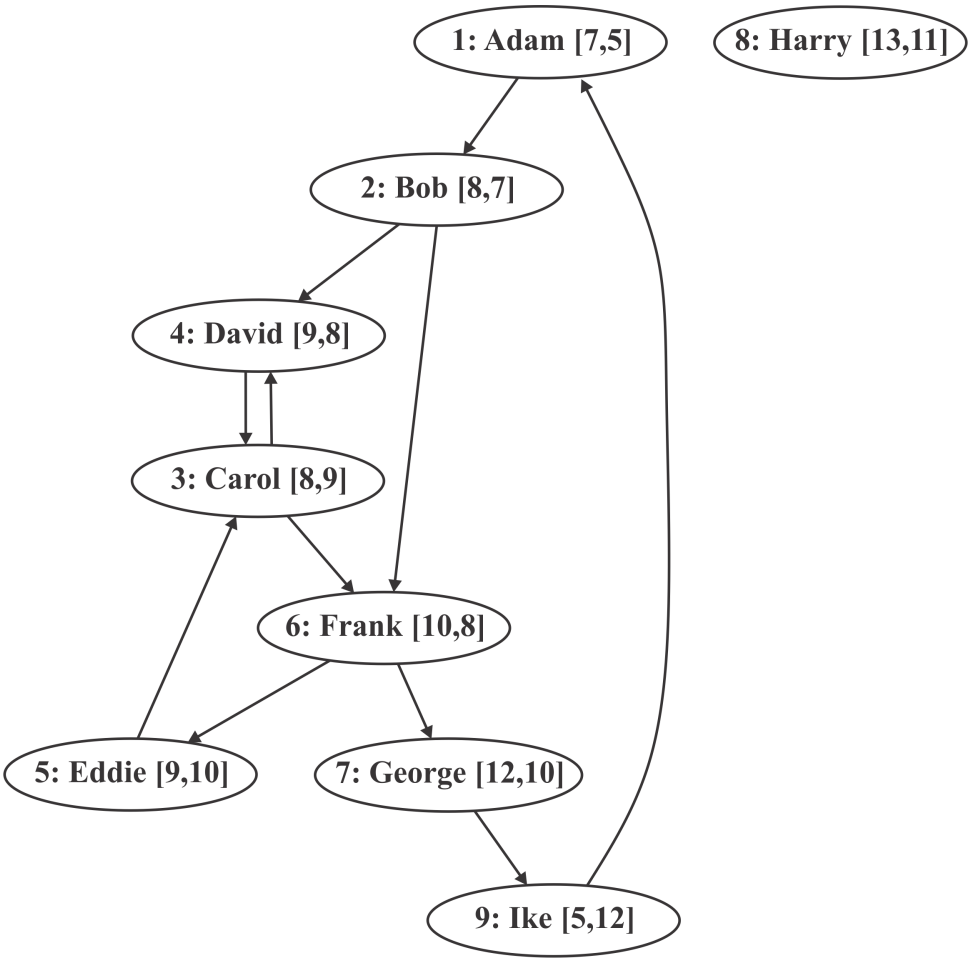


Figure 4.1: Revising cost distribution - only one isolated node

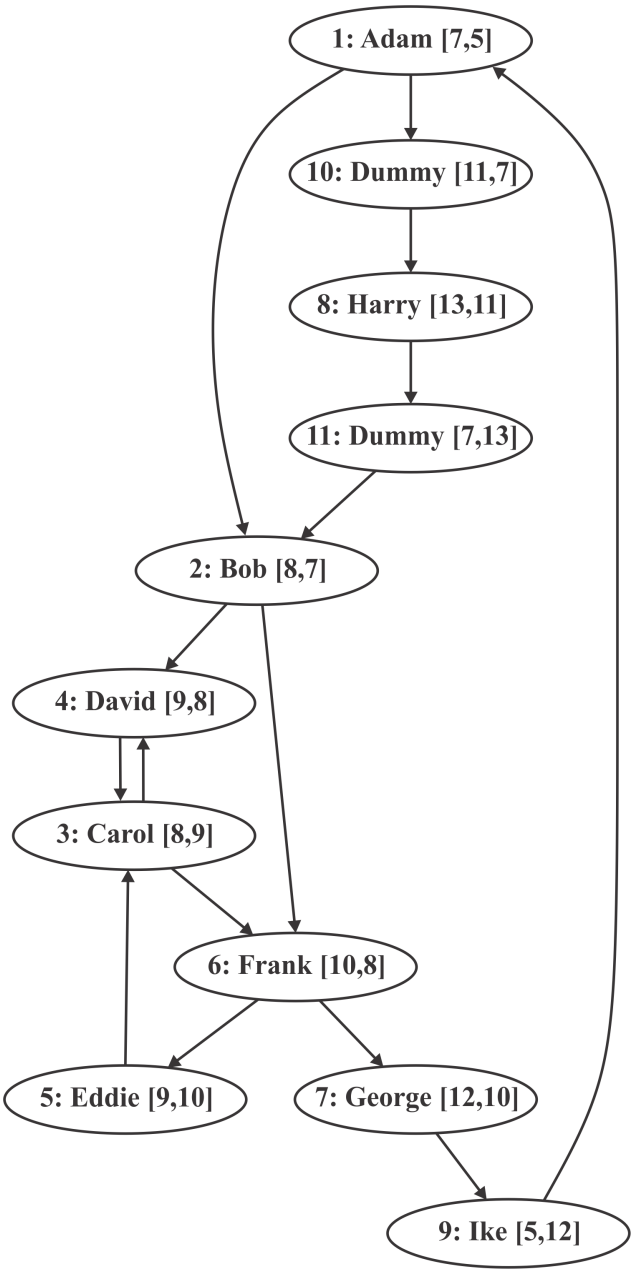


Figure 4.2: Revising cost distribution - two dummy nodes added

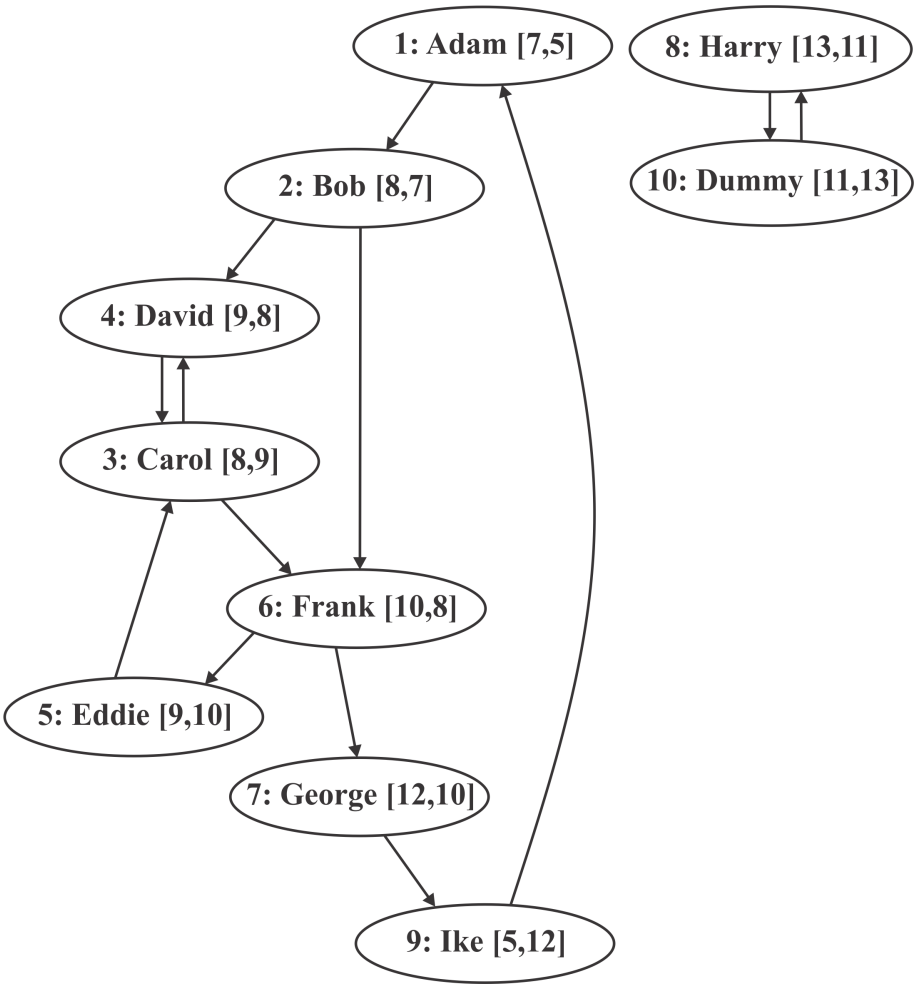


Figure 4.3: Revising cost distribution - an alternative

4.2.3 Avoiding Bridge Nodes

In the original graph enlargement example (Sanders, 2013*a*), the algorithm added a dummy node which created a bridge between two separate graph components. (This situation is not guaranteed to occur for every implementation of Sanders's algorithm, but it is the case with the original example.) The addition of dummy node 9 (see Figure 4.5) connected the two graph components, but caused both nodes 2 and 9 to not be contained within cycles, requiring the addition of yet another dummy node (see Figure 4.5).

To counter this problem, one could consider enlarging the graph by only using nodes that belong to the same component. By avoiding the creation of bridge nodes, the original graph could be enlarged using only one unique dummy node (see Figure 4.6).

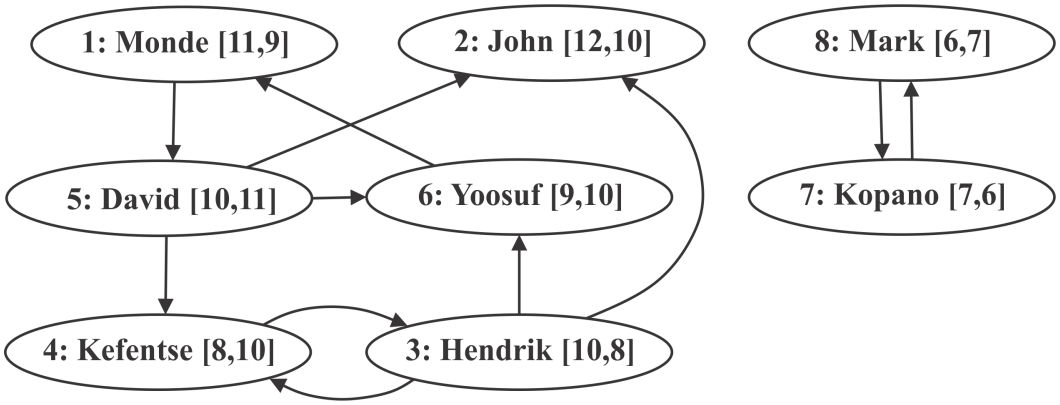


Figure 4.4: Sanders's original example before graph enlargement; note that node 2: John is not contained within a cycle

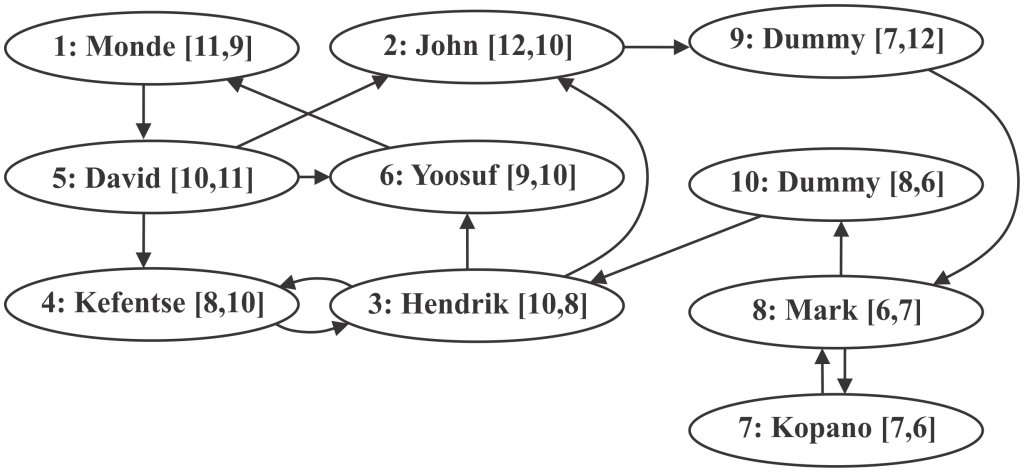


Figure 4.5: Sanders's original example after graph enlargement; note the addition of dummy nodes 9 and 10

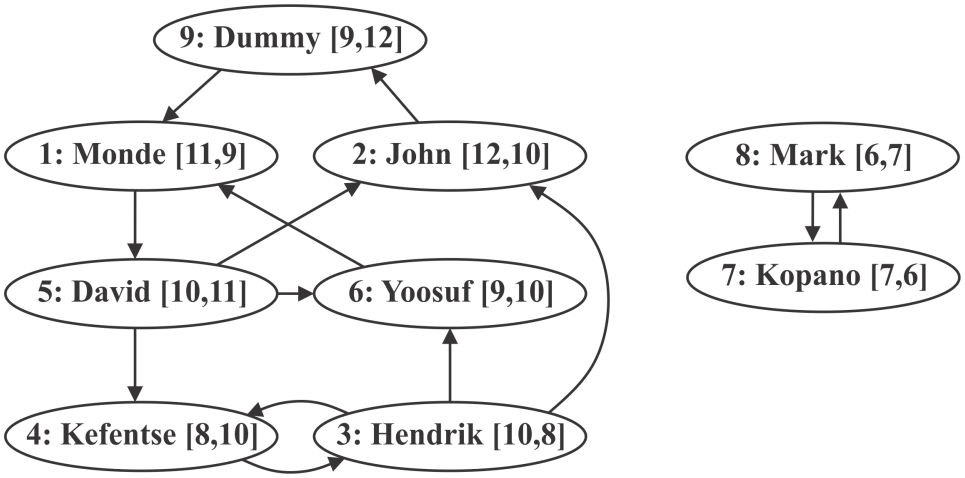


Figure 4.6: An alternative to Sanders's original example; only a single dummy node was added (node 9)

To find the separate components of a graph, the Hopcroft-Tarjan algorithm (Hopcroft and Tarjan, 1973) can be incorporated into the graph enlargement algorithm. Background information on the Hopcroft-Tarjan algorithm is given in Section 2.3.1.3.

4.2.4 Rule-based Decision Making

A new idea is presented in this subsection: an update to the original algorithm in which several cases of graphs and the optimal solution for each case will be explored. The new part of the algorithm will first assess the graph and the components within the graph, and then generate a "situation matrix" on the state of the graph. The algorithm will then use this matrix to follow predefined rules in order to enlarge the graph successfully.

Let the situation matrix be given by the following two rows for any graph G:

$$SM = \begin{array}{c} \\ \text{Graph} \\ \text{Component} \end{array} \begin{array}{c} \begin{array}{cccc} C & G & I & B \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

The C, G, I, B rows indicates the presence of charity, greedy, isolated, and bridge nodes in both the graph as a whole and the component the algorithm is currently evaluating. Charity nodes have no incoming edges, only outgoing. Greedy nodes are the opposite, with only incoming edges. Isolated nodes can belong to any component. Bridge nodes have both incoming and outgoing edges, but do not belong to a specific cycle. **Any value in the matrix can be either 0 or 1, i.e. a bit flip to indicate the presence of specific cases within the graph.**

As with Sanders's algorithm, bridge nodes can be handled separately, and only an exhaustive set of rules for the submatrix $[C, G, I]$ needs to be constructed. The following 8 cases are possible within the graph and/or currently inspected component:

- Case 0, $[C, G, I] = [0, 0, 0]$
  Possible handling of bridge nodes necessary within the graph, but no charity, greedy, or isolated nodes within the component. Sanders's *secondPass* algorithm will be applied to handle the bridge nodes.

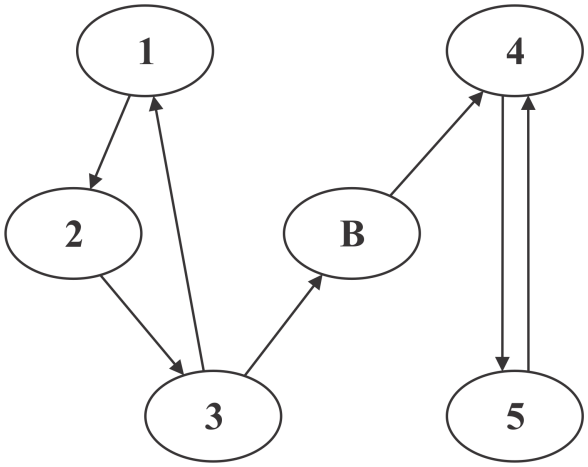


Figure 4.7: Illustrating case 0 of the rule-based algorithm

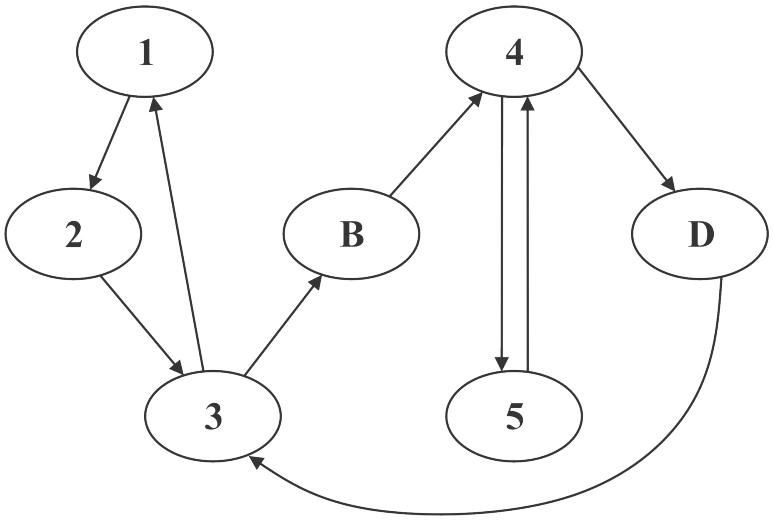


Figure 4.8: Illustrating case 0 of the rule-based algorithm (solution)

- Case 1, $[C, G, I] = [0, 0, 1]$
  There exists an isolated node only.
  Add a single dummy node as described in Section 4.2.2 (p. 43).

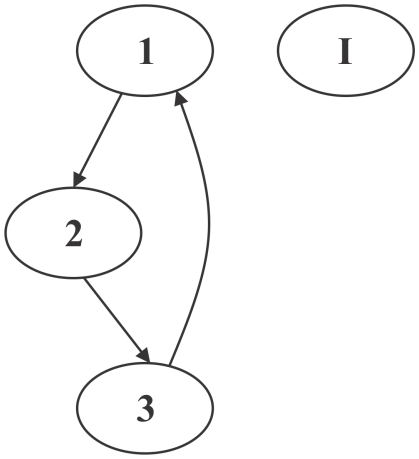


Figure 4.9: Illustrating case 1 of the rule-based algorithm

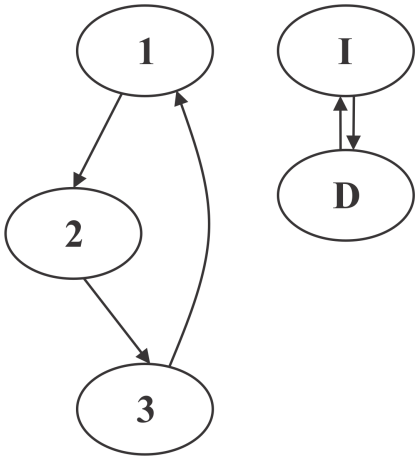


Figure 4.10: Illustrating case 1 of the rule-based algorithm (solution)

- Case 2, $[C, G, I] = [0, 1, 0]$
  There exists a greedy node only.
  Add a dummy node and link to greedy node to any different node in the component via the dummy node.

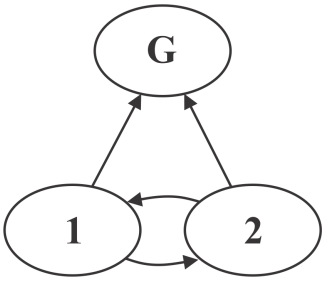


Figure 4.11: Illustrating case 2 of the rule-based algorithm

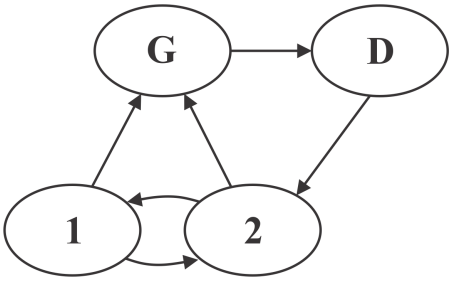


Figure 4.12: Illustrating case 2 of the rule-based algorithm (solution)

- Case 3, $[C,G,I] = [1,0,0]$
  There exists a charity node only.
  Add a dummy node and link to charity node to any different node in the component via the dummy node.

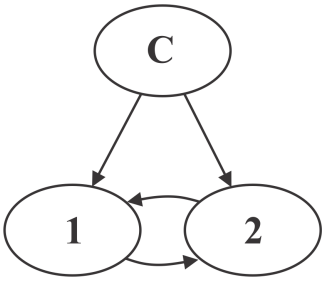


Figure 4.13: Illustrating case 3 of the rule-based algorithm

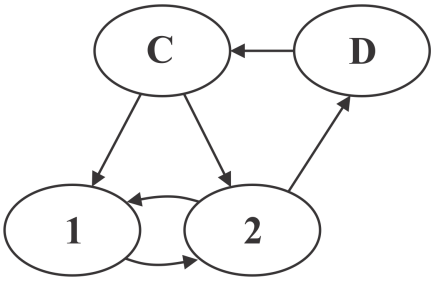


Figure 4.14: Illustrating case 3 of the rule-based algorithm (solution)

- Case 4, $[C,G,I] = [0,1,1]$
  There exists both an isolated and a greedy node.
  Add two dummy nodes and create the link: greedy $\rightarrow$ dummy 1 $\rightarrow$ isolated $\rightarrow$ dummy 2 $\rightarrow$ some node in the component containing the greedy node.

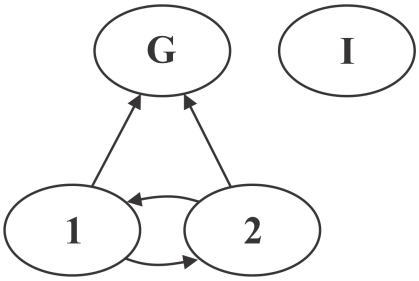


Figure 4.15: Illustrating case 4 of the rule-based algorithm

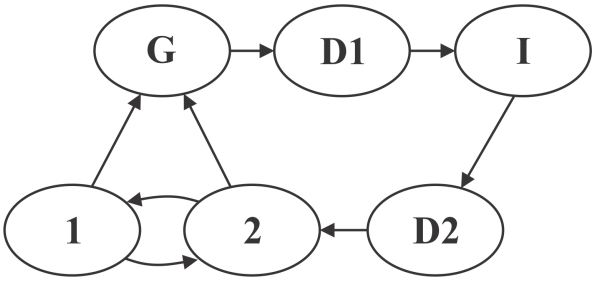


Figure 4.16: Illustrating case 4 of the rule-based algorithm (solution)

- Case 5, $[C,G,I] = [1,0,1]$
  There exists both an isolated and a charity node.
  Add two dummy nodes and create the link: some node in the component containing the charity node $\rightarrow$ dummy 1 $\rightarrow$ isolated $\rightarrow$ dummy 2 $\rightarrow$ charity.

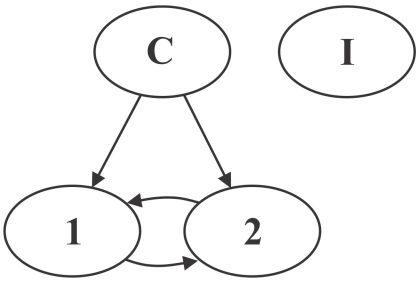


Figure 4.17: Illustrating case 5 of the rule-based algorithm

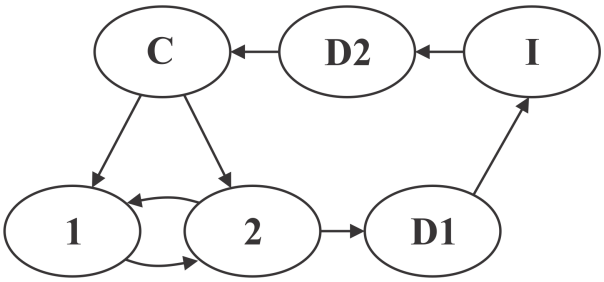


Figure 4.18: Illustrating case 5 of the rule-based algorithm (solution)

- Case 6, $[C, G, I] = [1, 1, 0]$
  There exists both a greedy and a charity node.
  Add a single dummy node via which to link the greedy and charity nodes.

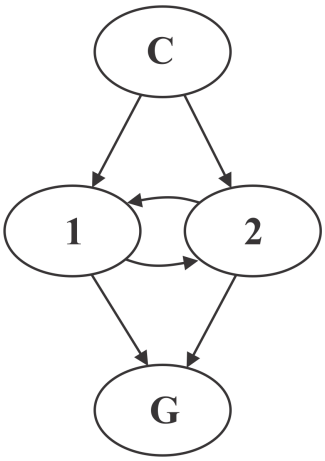


Figure 4.19: Illustrating case 6 of the rule-based algorithm

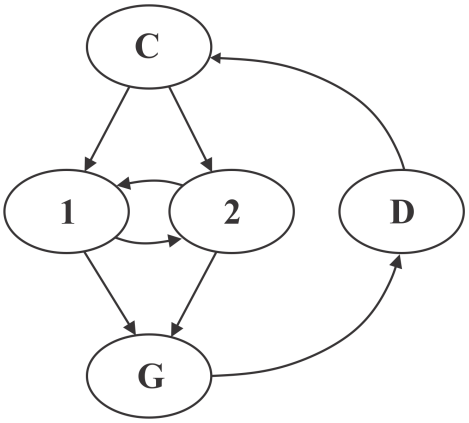


Figure 4.20: Illustrating case 6 of the rule-based algorithm (solution)

- Case 7, $[C, G, I] = [1, 1, 1]$
  The component contains greedy, charity, and isolated nodes.
  Add two dummy nodes and create the link: greedy $\rightarrow$ dummy 1 $\rightarrow$ isolated $\rightarrow$ dummy 2 $\rightarrow$ charity.

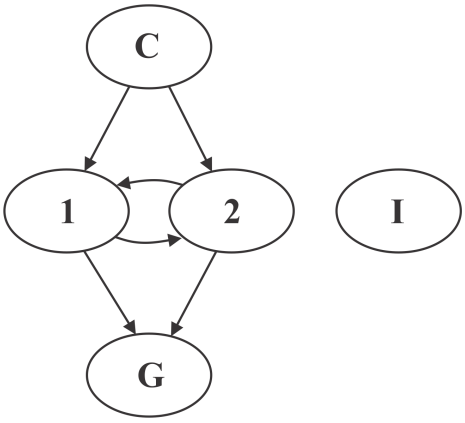


Figure 4.21: Illustrating case 7 of the rule-based algorithm

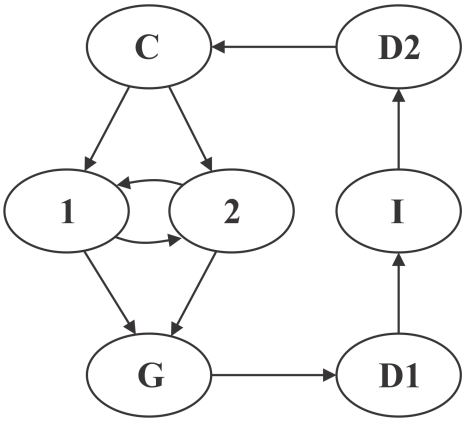


Figure 4.22: Illustrating case 7 of the rule-based algorithm (solution)

4.2.5 Cycle Compression

The proposed cost-optimised algorithm generally requires fewer unique dummy nodes than the solution provided by Sanders's algorithm. However, it is necessary to note that fewer unique dummy nodes does not necessarily imply fewer total nodes required; in fact, quite the opposite. The fact that fewer dummy nodes are required means that there will be more repeats of existing cycles and nodes.

To expand on this point, we illustrate using Figure 4.23. Note that even though only a single unique dummy node is required, four cycles are created (this would only happen if A, B, C, and D all have the same requirements):

- A $\rightarrow$ Dummy
- B $\rightarrow$ Dummy
- C $\rightarrow$ Dummy

- D $\rightarrow$ Dummy

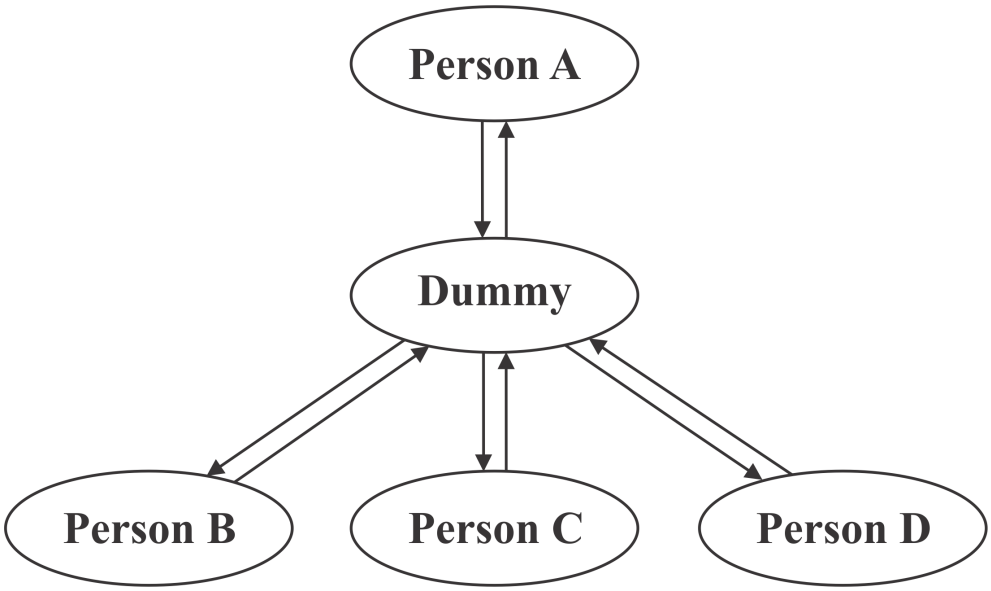


Figure 4.23: Illustrating repeated nodes (1)

All four of these cycles are required to ensure that all nodes are contained within cycles. This means that the dummy node, or dummy pair of shoes, needs to be purchased 4 times. In larger graphs, these chains become even larger and even more repeats are necessary, including repeats of individual, non-dummy nodes. For example, the graph illustrated in Figure 4.24 has the following cycles:

- A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D
- A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ E

Effectively, to accommodate nodes D and E, duplicates of nodes A, B, and C must be purchased. To accommodate the needs of 5 nodes (people), a total of 8 nodes (pairs of shoes) is required. However, we can observe that we need not buy the shoes for nodes A, B, C again. We can compress these 3 nodes into a single node by adding a dummy node consisting of the left shoe of C and the right shoe of A. The graph illustrated in Figure 4.25 has the following cycles and requires only 6 nodes in total:

- A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D
- Dummy$(C_{left}, A_{right}) \rightarrow$ E

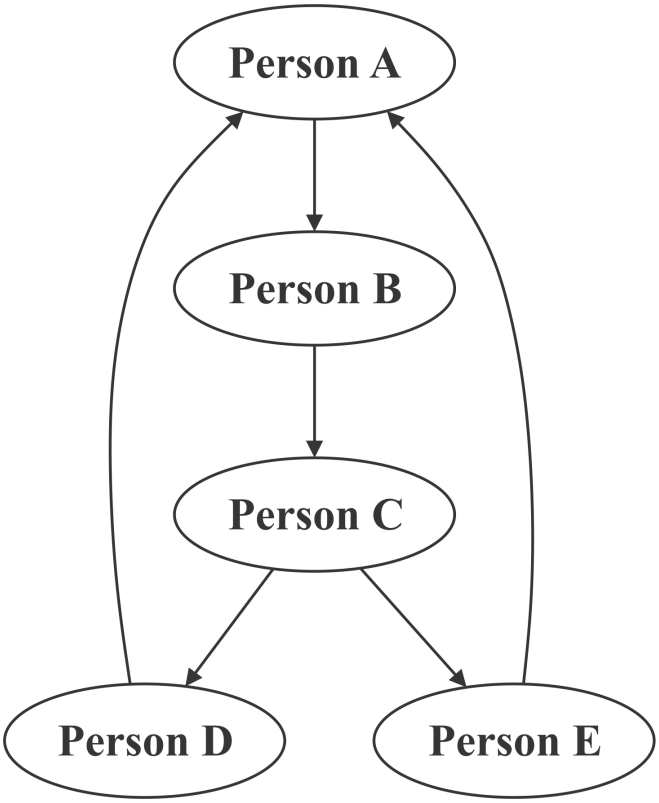


Figure 4.24: Illustrating repeated nodes (2)

4.2.5.1 Cycle Compression Implementation

At the heart of the cycle compression algorithm lies the functionality of determining the common node sequences between cycles. The longest common substring algorithm is perfectly applicable to this situation and makes the cycle compression technique very simple to implement. Once the common node sequences between cycles are found, they are sorted by length (longest sequences are substituted first) and then substituted with a single dummy node in the second cycle containing them. The first cycle which contains these nodes must remain untouched otherwise certain node sequences would be removed from the graph completely. Algorithm 14 contains the pseudocode for the longest common substring algorithm, sourced from `https://en.wikipedia.org/wiki/Longest_common_substring_problem`.

4.2.6 Implementation

A pseudocode implementation of the cost-optimised algorithm is given in Algorithm 15. The parameter G refers to the adjacency list of the graph. SM_G and SM_C refer to the situation matrices of the graph G and current component C, respectively.

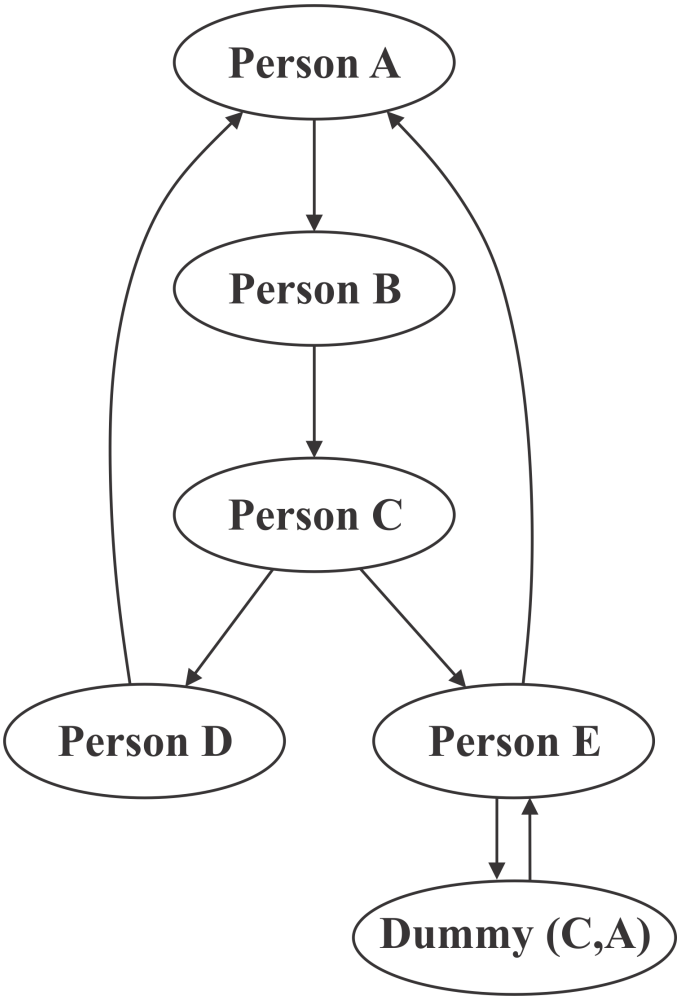


Figure 4.25: Illustrating repeated nodes (3)

Algorithm 14 Longest Common Substring

```
 1: procedure LCS(S[1..m], T[1..n])
 2:     L ← new array (1..m, 1..n)
 3:     ret ← ∅
 4:     z ← 0
 5:     for i = 1 to m do
 6:         for j = 1 to n do
 7:             if S[i] == T[j] then
 8:                 if i == 1 or j == 1 then
 9:                     L[i,j] = 1
10:                 else
11:                     L[i,j] = L[i-1,j-1] + 1
12:                 end if

13:                 if L[i,j] > z then
14:                     z = L[i,j]
15:                     ret = {S[i-z+1..i]}
16:                 else if L[i,j] == z then
17:                     ret = ret ∪ {S[i-z+1..i]}
18:                 end if
19:             else
20:                 L[i,j] = 0
21:             end if
22:         end for
23:     end for

24:     return ret
25: end procedure
```

Algorithm 15 Graph Enlargement: Cost-optimised Algorithm

```
1:  procedure ENLARGE(G)
2:      SM_G ← BuildMatrix(G)
3:      while SM_G ≠ [0, 0, 0, 0] do                    ▷ Evaluate [C,G,I,B]
4:          matrix ← SM_G

5:          for all component C ∈ G do
6:              determine SM_C
7:              if SM_C ≠ [0,0,0] then                  ▷ Only evaluate [C,G,I]
8:                  Solve using the rules defined on p. 49
9:              end if
10:         end for

11:         if bridge nodes ∈ G then
12:             Apply Sanders's Second Pass Algorithm (p. 29)
13:         end if
14:     end while
15: end procedure

16: procedure BUILDMATRIX(G)
17:     charity ← nodes with no incoming edges
18:     greedy ← nodes with no outgoing edges
19:     isolated ← nodes which are isolated
20:     bridges ← nodes which are bridges
21:     Initialise: C ← 0, G ← 0, I ← 0, B ← 0

22:     if charity ≠ ∅ then
23:         C = 1
24:     end if
25:     if greedy ≠ ∅ then
26:         G = 1
27:     end if
28:     if isolated ≠ ∅ then
29:         I = 1
30:     end if
31:     if bridges ≠ ∅ then
32:         B = 1
33:     end if

34:     return [C,G,I,B]
35: end procedure
```

4.2.6.1 Example: Sanders's Algorithm vs. the Cost-optimised Algorithm

To illustrate the difference between the cost-optimised and Sanders's algorithms, suppose one slightly alters the original problem as follows (see Figure 4.26): An extra node (no. 9) has been added to purposefully illustrate the different ways the two algorithms would go about solving this problem. In the solution generated by Sanders's algorithm (see Figure 4.27), a dummy node (no. 10) first connects nodes 2 and 9. Node 10 therefore becomes a bridge node between the two separate components, compounding the problem.

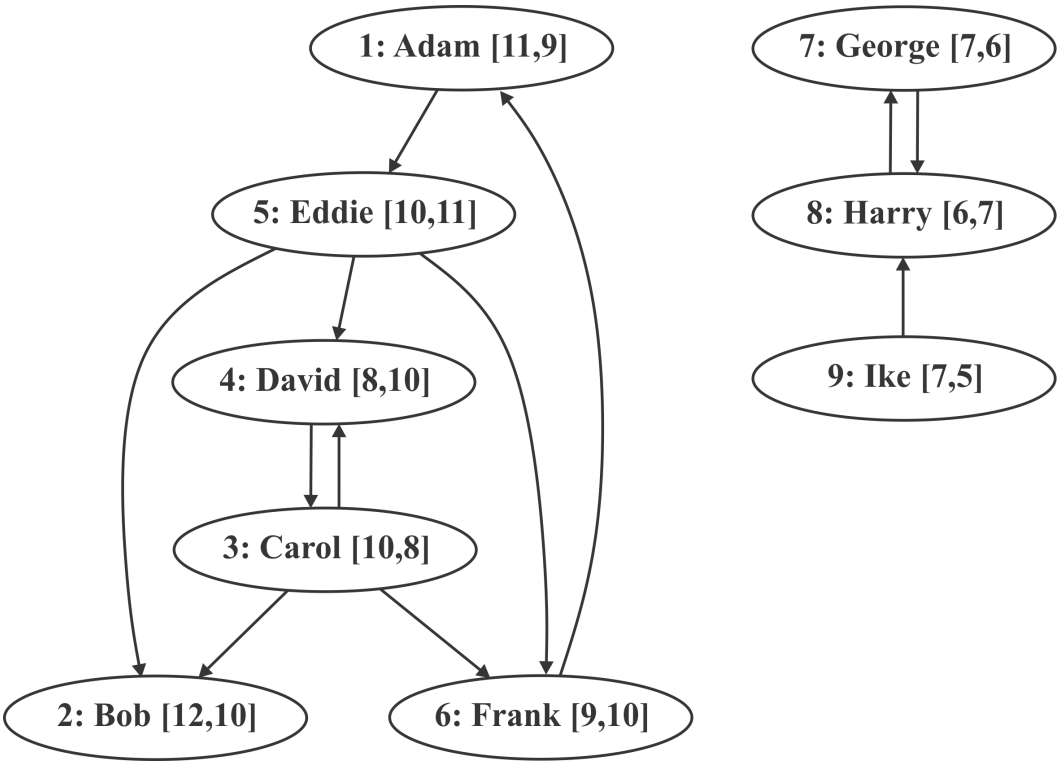


Figure 4.26: Altered version of the original problem

The creation of the bridge node requires a second pass of Sander's algorithm, leading to a solution which, in total, requires the addition of four unique dummy nodes (see Figure 4.28).

By implementing the changes to the original algorithm as discussed in this section, a solution which requires only two unique dummy nodes can be reached (see Figure 4.29). Note that no bridge is created between the separate components of the graph.

Table 4.1 outlines the main differences between the solutions provided by the two algorithms. The cost-optimised algorithm requires fewer unique dummy nodes, but with repeats of certain nodes. In this case $1 \rightarrow 5$ is repeated in 2 cycles generated by the cost-optimised algorithm. A cycle compression algorithm (see Section 4.2.5) can compress all but one of these

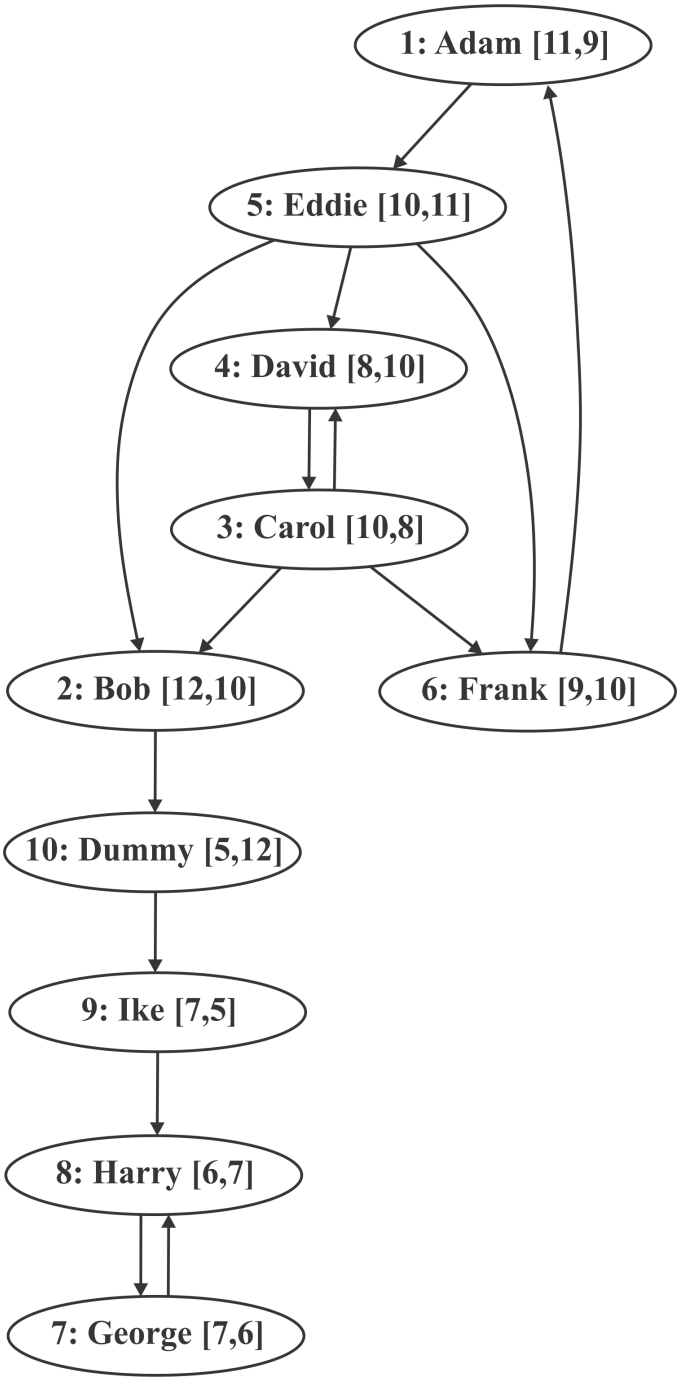


Figure 4.27: Solution generated by Sanders's algorithm (first pass)

repeats into a single node, thereby reducing the final number of nodes required.

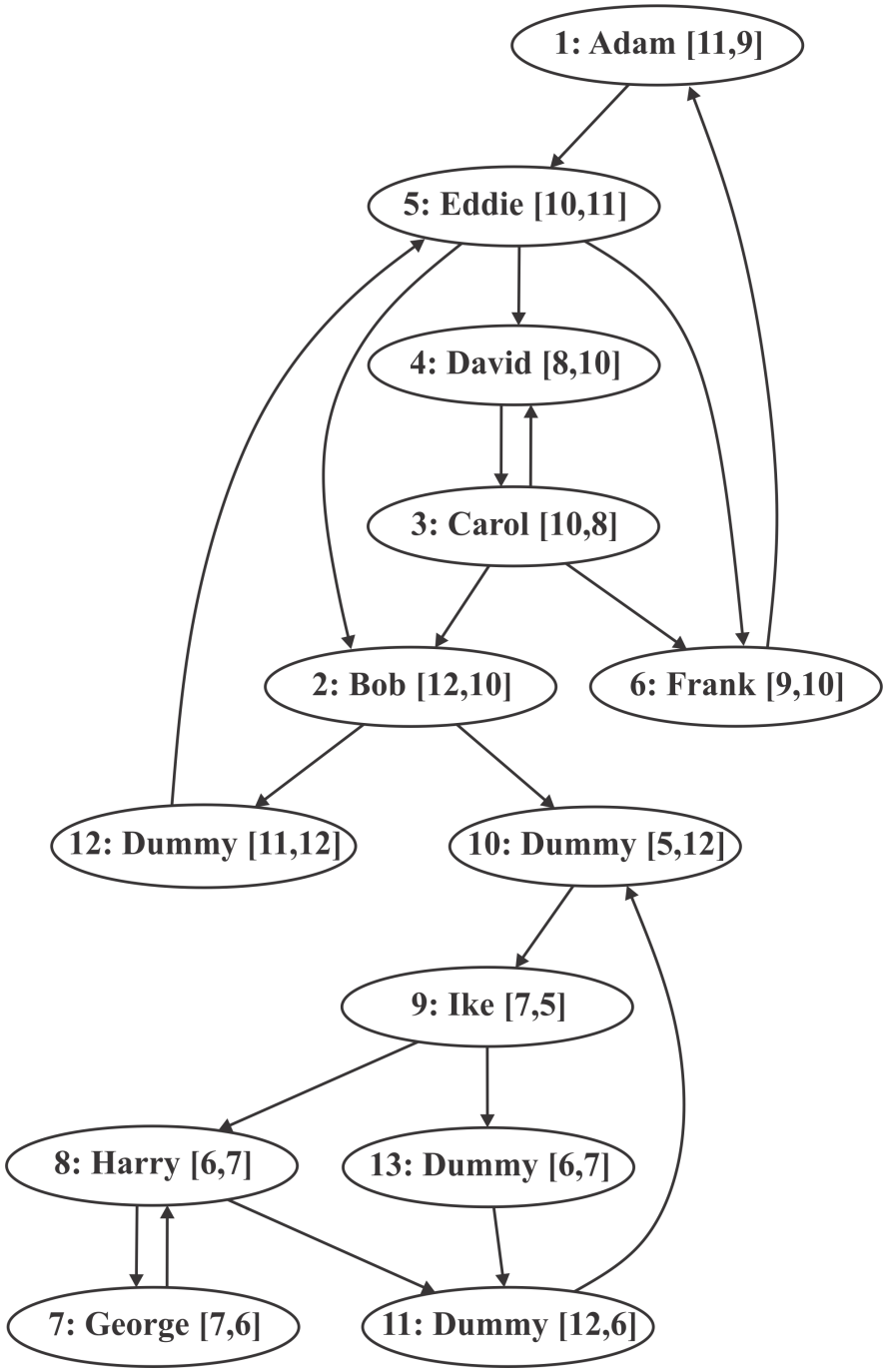


Figure 4.28: Solution generated by Sanders's algorithm (second pass)

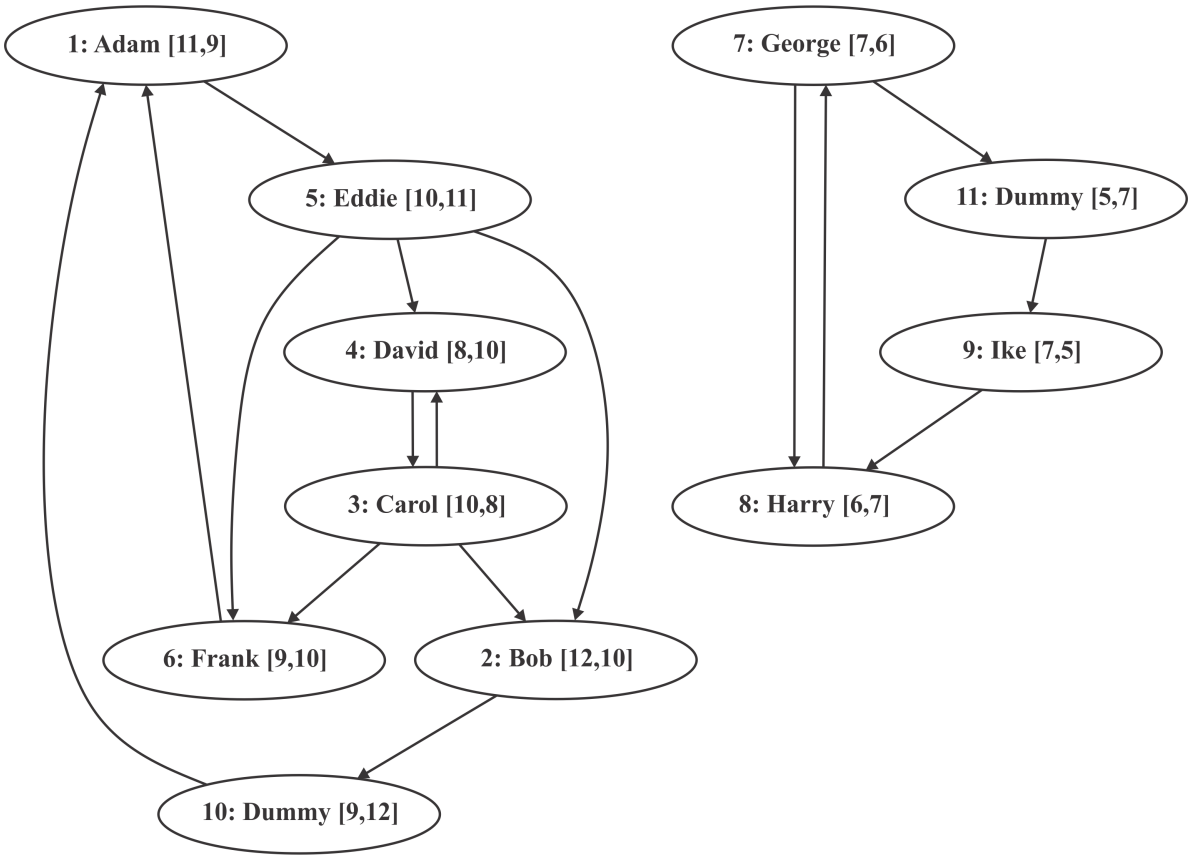


Figure 4.29: A solution requiring fewer unique dummy nodes

	Sanders's Algorithm	**Cost-optimised Algorithm**
Number of nodes	9	9
Unique dummy nodes	4	2
Cycles post-enlargement (min. cycles picking method)	$1 \to 5 \to 6$ $2 \to 12 \to 5 \to 4 \to 3$ $7 \to 8$ $9 \to 13 \to 11 \to 10$	$1 \to 5 \to 6$ $1 \to 5 \to 4 \to 3 \to 2 \to 10$ $7 \to 11 \to 9 \to 8$
Total dummy nodes with repeats	4	2
Total nodes with repeats	14	13
Repeated nodes	Not implemented	$1 \to 5$
Total nodes after compression	Not implemented	12

Table 4.1: Detailed example: Sanders's algorithm vs. the cost-optimised algorithm

4.2.7 Interpretation of Results

There are several limitations to both Sanders's original algorithm as well as the proposed cost-optimised algorithm:

- Both require multiple runs of a cycle enumeration algorithm (in this case, Tarjan's) - a very CPU-intensive operation since the growth in the number of cycles is nearly exponential (see Section 4.3.1 on p. 67).
- To determine the repeated number of dummy nodes and total nodes required, one needs to run a cycle picking algorithm which is also a CPU-intensive operation. Due to the conjectured NP-completeness of the cycle picking problem (Sanders, 2013*a*), a combinatorial algorithm must be used to evaluate all the possible combinations in order to find the most efficient combination. For the purposes of this dissertation, the *r*-combinations algorithm of Johnsonbaugh (2000) was used.
- The fact that fewer unique dummy nodes are required makes little difference to the total number of nodes necessary in the final solution (see Section 4.2.5). This is due to repeats of certain nodes to accommodate every individual node (person) belonging to the graph. There is, in general, only a minor improvement on the total number of nodes required.
- To counter the problem stated in the point above, we need to run a cycle compression algorithm (see Section 4.2.5); also a CPU-intensive operation.

For Tables 4.2, 4.3, and 4.4 please note the use of the following terminology (as well as further use of this terminology in succeeding sections):

- **Dummy Nodes (with repeats):** the total number of **Unique Dummy Nodes** once all their repeat occurrences in multiple cycles have been added up.
- **Total Nodes (with repeats):** the total number of nodes (both dummy and actual participant nodes), once all their repeat occurrences in multiple cycles have been added up.

To summarise, it can be said that a relatively small number of unique dummy nodes does not imply a cost-efficient graph and it follows that decreasing the number of unique dummy nodes required does not necessarily imply a decrease in the total number of nodes required; at least not without the help of auxiliary functions, such as cycle picking and cycle compression. The cost-optimised algorithm provides a minor improvement on Sanders's Algorithm, in general.

Seed: 123 789 456	**Sanders's Original Algorithm**			**Cost-optimised Algorithm**		
Number of Nodes	**Unique Dummy Nodes**	**Dummy Nodes (with repeats)**	**Total Nodes (with repeats)**	**Unique Dummy Nodes**	**Dummy Nodes (with repeats)**	**Total Nodes (with repeats)**
8	2	2	12	2	2	11
10	2	2	12	2	2	12
12	2	2	16	2	2	16
14	2	4	24	2	2	21
16	4	4	23	4	4	22
18	6	- (1)	-	4	-	-

1: the algorithm could not complete within a reasonable timeframe of approx. 1 hour

Table 4.2: Sanders's algorithm vs. the cost-optimised algorithm (1)

Seed: 170 888 264	**Sanders's Original Algorithm**			**Cost-optimised Algorithm**		
Number of Nodes	**Unique Dummy Nodes**	**Dummy Nodes (with repeats)**	**Total Nodes (with repeats)**	**Unique Dummy Nodes**	**Dummy Nodes (with repeats)**	**Total Nodes (with repeats)**
8	3	3	13	3	3	13
10	6	7	22	5	6	20
12	8	8	24	4	5	25
14	12	12	31	8	8	27
16	6	9	33	4	4	32
18	7	- (2)	-	5	-	-

2: the algorithm could not complete within a reasonable timeframe of approx. 1 hour

Table 4.3: Sanders's algorithm vs. the cost-optimised algorithm (2)

Seed: 923 462 908	**Sanders's Original Algorithm**			**Cost-optimised Algorithm**		
Number of Nodes	**Unique Dummy Nodes**	**Dummy Nodes (with repeats)**	**Total Nodes (with repeats)**	**Unique Dummy Nodes**	**Dummy Nodes (with repeats)**	**Total Nodes (with repeats)**
8	3	4	15	4	4	14
10	4	4	16	3	3	15
12	2	2	19	2	2	19
14	3	3	28	3	3	27
16	3	3	27	3	3	27
18	2	- (3)	-	2	-	-

3: the algorithm could not complete within a reasonable timeframe of approx. 1 hour

Table 4.4: Sanders's algorithm vs. the cost-optimised algorithm (3)

4.3 Speed Optimisation

It is also worth investigating whether or not the current running time of Sanders's algorithm could be optimised. As part of this investigation, the theoretical runtime of the current algorithm needs to be examined and solutions need to be found to any bottlenecks.

4.3.1 Growth in the Number of Cycles and its Effect

The number of cycles in the graph increases dramatically, even exponentially, in relation to the number of nodes in the graph. Table 4.5 reflects the growth in the number of cycles and Figure 4.30 offers a graphical representation of the data.

Seed	Number of nodes enumerated	Total number of cycles
123 789 456	26 *	446 164
170 888 264	30	263 258
923 462 908	30	111 516
482 367 498	30	35 113
532 499 987	30	648 297

*: the algorithm could not enumerate the number of cycles for 27 nodes within a reasonable timeframe of approx. 24 hours

Table 4.5: Growth in the number of cycles

The growth in the number of cycles creates a problem for the cycle enumeration and cycle picking algorithms, slowing down the algorithm even further. It would be ideal to have an algorithm which did not rely on cycle enumeration and cycle picking to guarantee that all nodes are in cycles when the algorithm completes.

4.3.2 Permutations and Permutation Matrices

The concept of adjacency matrices to represent graphs was previously explained in Section 2.3.2.1 (p. 12). This section and its subsections focus on permutations and permutation matrices, as well as transforming an adjacency matrix into a permutation matrix.

Permutations and permutation matrices are important concepts in group theory. Firstly, it is important to understand the definition of a permutation as defined in group theory. Rotman (1994) defines a permutation as:

"If X is a nonempty set, a permutation of X is a bijection α: $X \to X$."

A bijection is a function which is both one-to-one and onto. In essence, a

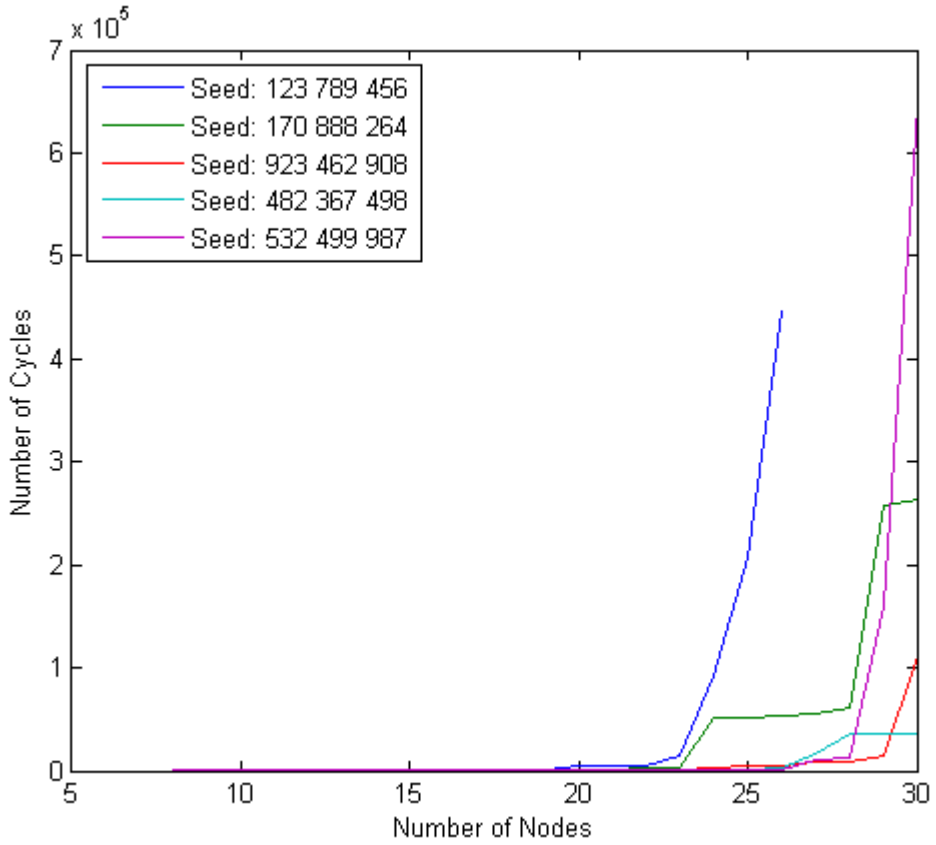


Figure 4.30: A graphical representation of the growth in the number of cycles; note the exponential characteristic of the slopes.

permutation is a rearrangement of elements. Suppose we have permutations:

$$\alpha = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

$$\beta = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

The permutation $\alpha(\beta(1)) = \alpha(2) = 2$ is an example of such a rearrangement of elements.

Any permutation can be written as a union of disjoint cycles (Fraleigh, 2003, p. 89), for example:

$$\alpha = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 3 & 8 & 6 & 7 & 4 & 1 & 5 & 2 \end{pmatrix} = (1,3,6)(2,8)(4,7,5)$$

Fraleigh (2003) provides the following proof that a permutation of a finite set is a product of disjoint cycles:

Let $B_1, B_2, ..., B_r$ be the orbits of α and let μ_i be the cycle defined by

$$\mu_i = \begin{cases} \alpha(x) \text{ for } x \in B_i \\ x \text{ otherwise.} \end{cases}$$

It follows that $\alpha = \mu_1\mu_2...\mu_r$. Since the equivalence-class orbits $B_1, B_2, ..., B_r$, being distinct equivalence classes, are disjoint, the cycles $\mu_1\mu_2...\mu_r$ are disjoint also. □

The fact that a permutation consists solely of disjoint cycles is very important and corresponds closely to the focus of this dissertation. It means that every element of the permutation (and its corresponding permutation matrix) will always be contained within a cycle, and there will be no repeats of elements in the same cycle or different cycles. This provides us with the following benefits:

- There will be no need for cycle enumeration.
- There will be no need for cycle picking.
- There will be no need to check for bridge nodes.
- There will be no need to check for repeated nodes. The number of nodes contained within the graph at first glance is the number of nodes necessary for every node to be contained within a unique cycle.

4.3.2.1 Permutation Matrices

Permutation matrices are square matrices filled with ones and zeroes (row-equivalent to the identity matrix), with the restriction that the ones may only occur once in every row and column. Matrix A is a valid permutation matrix:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Whilst matrix B is not:

$$B = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The permutation matrix represents a permutation of elements and the permutation matrix can be multiplied with another matrix to permute the elements of that matrix. Any permutation can be represented by a permutation matrix. For this dissertation the matrix represents the adjacency matrix of the graph, but instead of permuting the elements the focus is instead on generating disjoint cycles by adding dummy elements and transforming the graph's adjacency matrix to a permutation matrix.

4.3.3 Implementation

Below follows the steps taken by the algorithm to enlarge the graph; presented as a more involved set of instructions rather than pseudocode.

The process of transforming an adjacency matrix to a permutation matrix could be performed as per the following method. Suppose we have a graph G as illustrated below in Figure 4.31, with adjacency matrix:

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 1 & 2 & \end{pmatrix} \end{array}$$

T_r represents the sum of the ones in each row, while T_c represents the sum of the ones in each column. In a permutation matrix, each row and each column must add up to 1, only. That is to say, each row and each column may only contain a single entry of 1, the rest of the matrix must be populated with zeroes.

In this matrix, three rows have $T_r = 2$ (thus 1 redundant outgoing edge for each of the nodes v_1, v_3, v_5) and two columns have $T_c = 2$ (thus 1 redundant incoming edge for nodes v_4, v_6). Starting with T_r, work from bottom-to-top

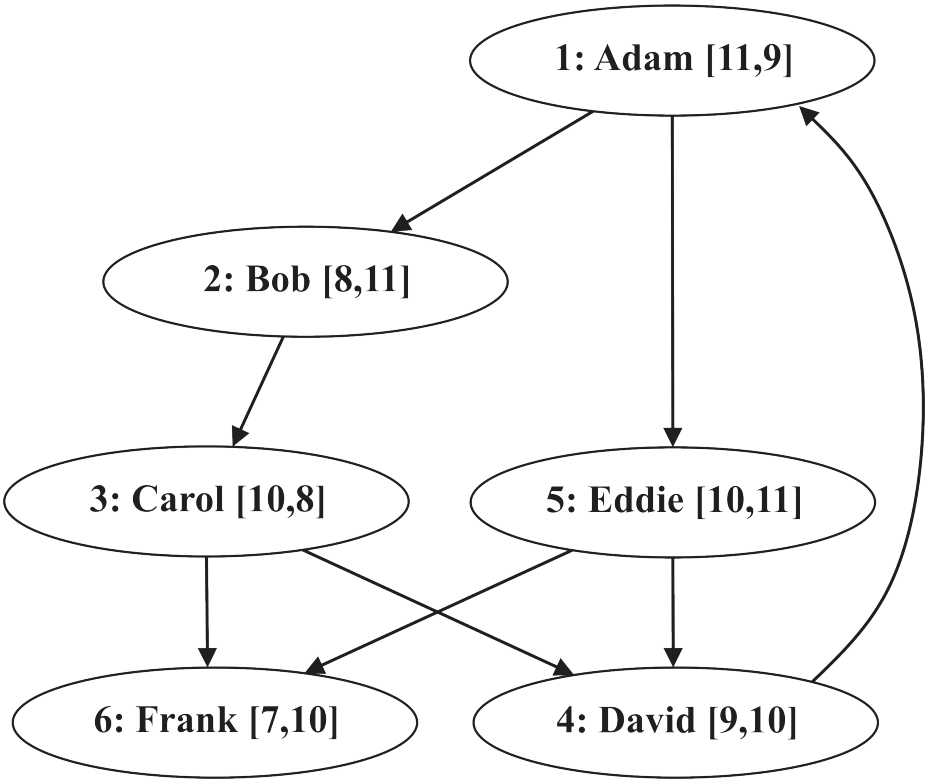


Figure 4.31: A permutation matrix will be constructed from the adjacency matrix of this graph

and right-to-left (the order is not important, the reader could also work from top-to-bottom and/or left-to-right), and simply remove the redundant entries of 1 and replace them with 0. Every value in T_r should now be either 0 or 1. Refresh the count for T_c to check if any nodes have redundant incoming edges.

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 1 & 0 & \end{pmatrix} \end{array}$$

In this case, the removal of all redundant outgoing edges has not solved the problem of redundant incoming edges. If any value for $T_c > 1$, again work from bottom-to-top and right-to-left, removing the redundant ones in each column.

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & \end{pmatrix} \end{array}$$

However, now note the number of zeroes in T_r and T_c (these two values should be equal), say n, and enlarge the graph by n dummy nodes. In this case $n = 2$ and we are left with the following adjacency matrix:

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \end{pmatrix} \end{array}$$

It is essential to keep track of the dummy nodes (v_7, v_8) and the original zero-row nodes (v_5, v_6 - from top to bottom) and zero-column nodes (v_6, v_5 - from right to left).

The graph is now ready to be enlarged. Pop the first zero-row node (v_5) off its stack, the first dummy node (v_7), and the first zero-column node (v_6). Connect $v_5 \to v_7 \to v_6$ only. Do not connect the last node in this sequence to the first. The matrix is altered as follows:

- $A[v_5, v_7] = 1$
- $A[v_7, v_6] = 1$

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & T_r \end{array} \\ \left(\begin{array}{ccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & \end{array} \right) \end{array}$$

The next selection of nodes to be popped is $v_6 \rightarrow v_8 \rightarrow v_5$ only.

- $A[v_6, v_8] = 1$
- $A[v_8, v_5] = 1$

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & T_r \end{array} \\ \left(\begin{array}{ccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \end{array} \right) \end{array}$$

Since both T_r and T_c are filled with ones (exclusively), the enlargement is complete (see Figure 4.32 on p. 74). The adjacency matrix has now been transformed into a permutation matrix and inherits the properties of a permutation; once again, of special interest to the focus of this dissertation are the following:

- The permutation matrix is a union of disjoint cycles: every node is contained within a cycle
- No node is ever repeated: every node appears exactly once, in exactly one cycle

In Figure 4.32, (7: Dummy [10,10]) is a redundant dummy node. It is possible to remove this node and still have a complete cycle and viable solution.

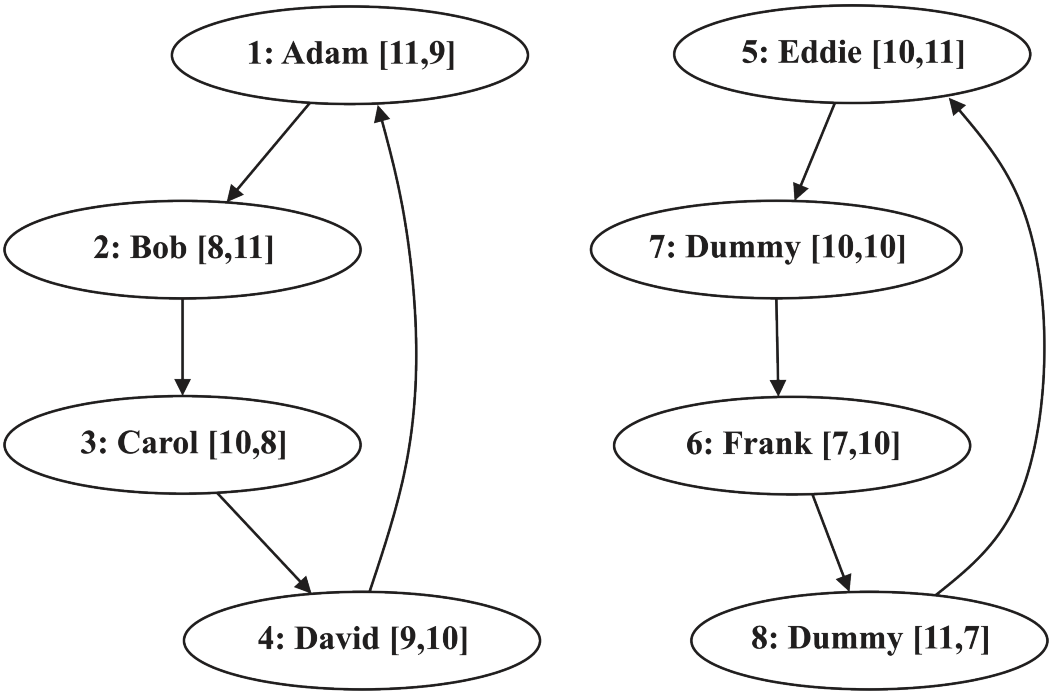


Figure 4.32: The final graph produced by the permutation matrix

However, the addition of 2 dummy nodes is a failsafe to construct a permutation matrix. **Also note that cycle picking is redundant for this enlargement method. When the algorithm destroys the redundant edges, it is in essence performing a procedure very much like cycle picking on itself.** For example, technically an edge should be present (one of many) between (3: Carol [10,8]) and (6: Frank [7,10]). The disjoint cycles property of permutation matrices produces cycles which resemble "picked" cycles.

4.3.3.1 Example: Implementation on Sanders's Original Shoe Matching Problem

Recall Sanders's original shoe matching problem as illustrated in Figure 4.33. Sanders's original algorithm solved the problem by adding 2 dummy nodes (see Section 4.2.3 on p. 47).

The speed optimised algorithm solves the problem as illustrated in Figure 4.34. Note that node (10: Dummy [10,10]) is again a redundant node, but necessary to guarantee a permutation matrix. The redundant node can simply be removed, and it could be argued that only a single dummy node is necessary to solve the problem. This puts the speed-optimised algorithm's solution on par with the cost-optimised algorithm with regards to node cost.

The removal of these redundant nodes are trivial. Simply discard any node $v_{x,y}$ where $x = y$, in other words: any nodes where the left shoe size equals the size of the right shoe. To illustrate the structure of the permu-

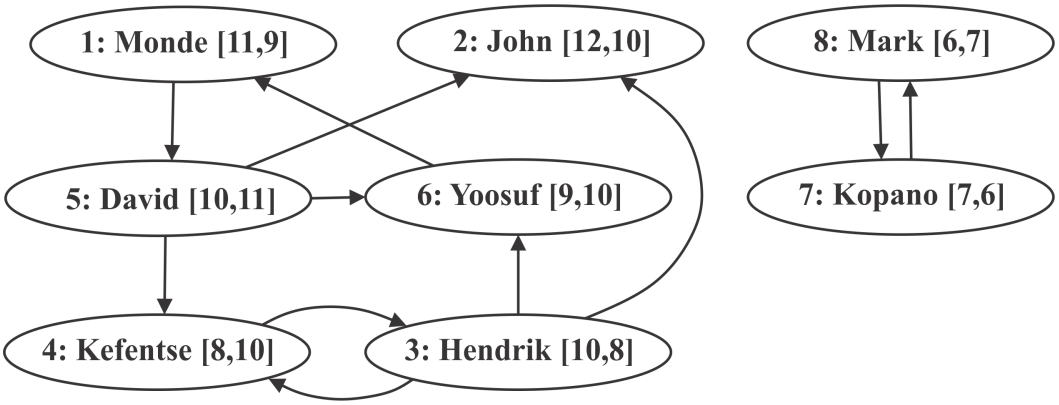


Figure 4.33: Sanders's original example before graph enlargement; note that node 2: John is not contained within a cycle

tation matrices (and the resulting graphs), these redundant nodes are **not** removed from the figures for the purposes of this dissertation.

4.3.4 Implementation on Larger Graphs

As an illustrative example of how the matrix enlargement algorithm alters larger graphs, consider the example in Figure 4.35.

The end result in Figure 4.36 illustrates clearly the impact of the transformation, from an adjacency matrix to a permutation matrix. Nodes are contained in exactly one cycle, only once. The transformation does alter the entire structure of the graph, but also ensures that each node is contained within a cycle.

4.3.5 Interpretation of Results

This new method of enlargement provides several benefits and improvements. The dramatic decrease in running time is a big improvement on the original algorithm.

The final graph produced forces nodes into specific cycles, which means nodes do not overlap between cycles. This, in turn, produces no repeated nodes. No cycle picking, cycle enumeration, or cycle compression is therefore necessary.

This new method of graph enlargement may also be applicable in a situation where one would want to reduce the total number of cycles. Section 4.3.4 illustrates how the entire structure of the graph can be reduced into a much simpler layout, with a drastic reduction in the number of cycles and drawing complexity of the graph.

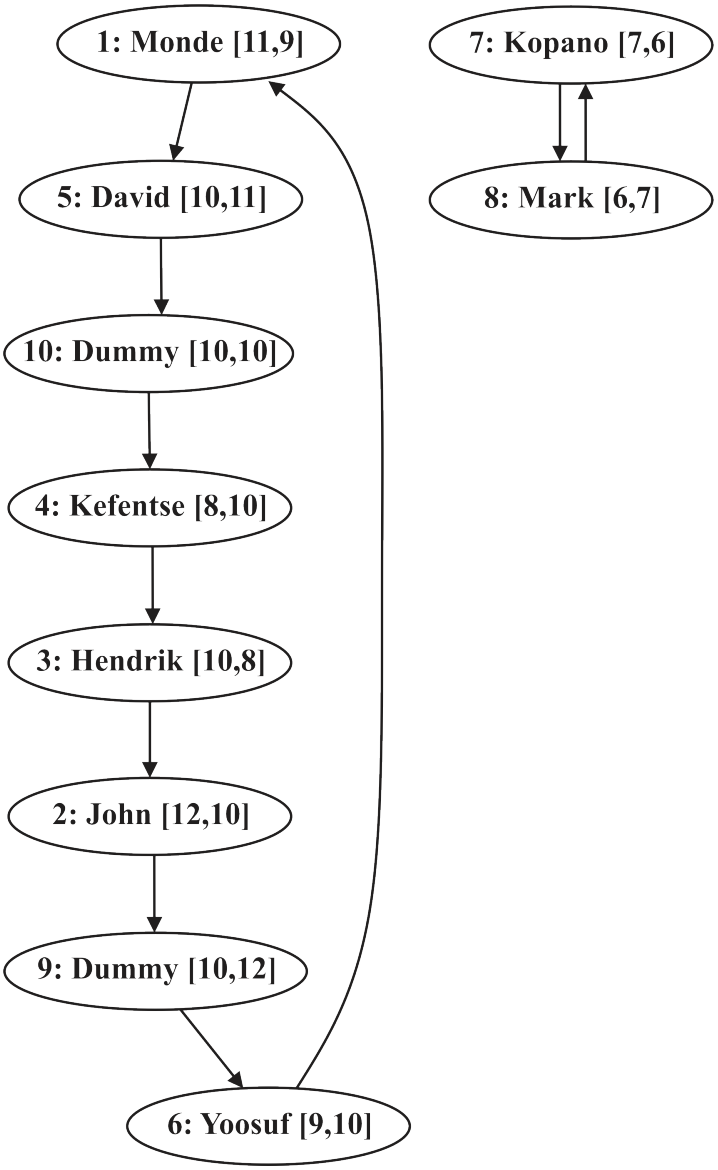


Figure 4.34: Sanders's original example solved by the speed-optimised algorithm

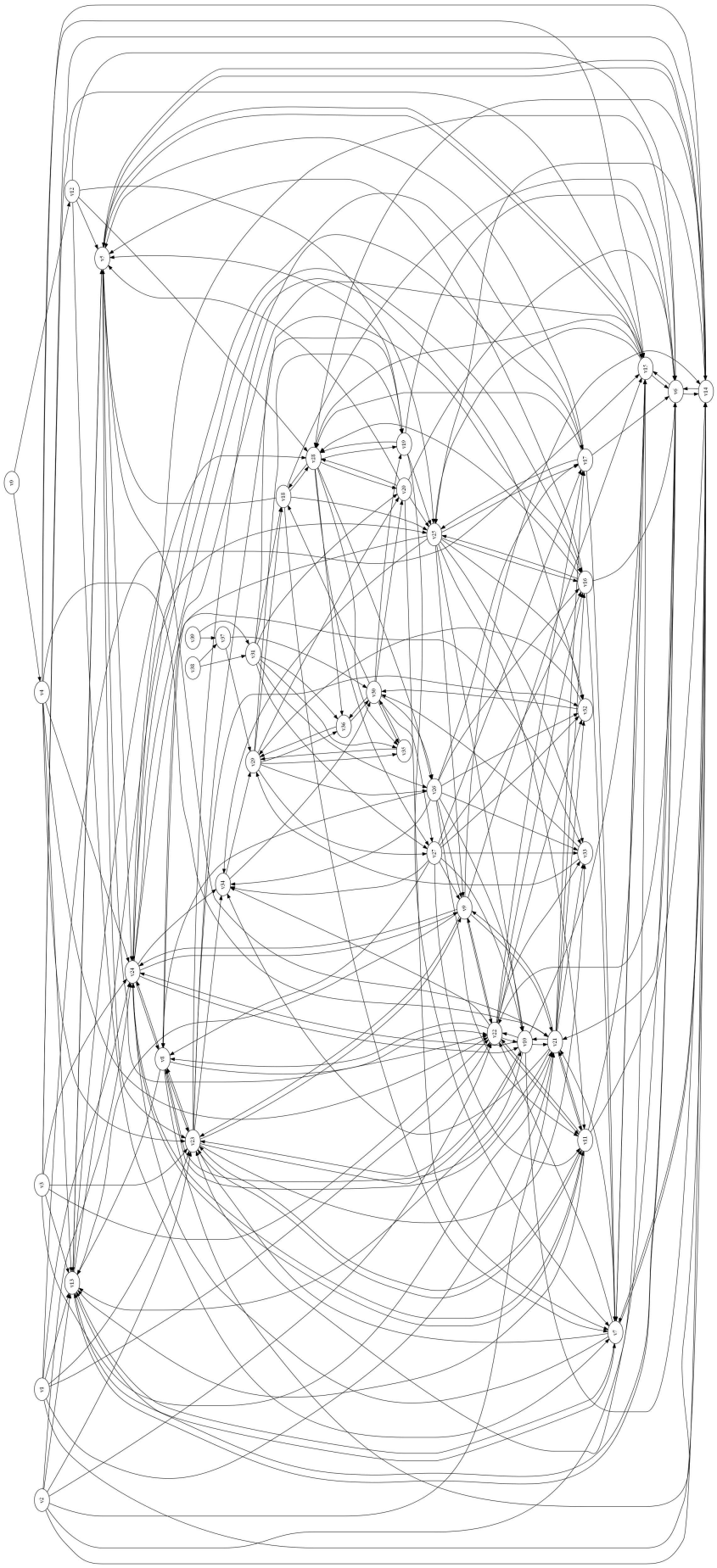


Figure 4.35: An example of the structure of a larger graph pre-enlargement

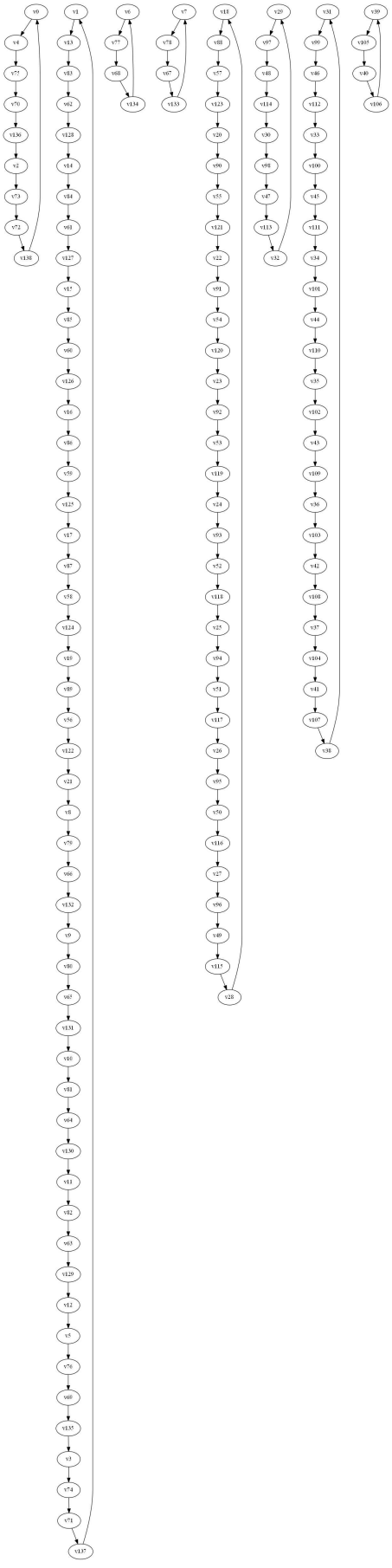


Figure 4.36: An example of the structure of a larger graph post-enlargement

The matrix enlargement method has the following drawbacks:

- The bulk of the algorithm runs in $O(n)$ time, but the function removing redundant incoming/outgoing edges runs in $O(n^3)$; thus theoretically reducing the efficiency of the entire algorithm to $O(n^3)$.
- Memory usage could become intensive. Unfortunately a permutation matrix is a sparse matrix, and a graph of 1000 nodes would require a 1000×1000 element array of which only 1000 elements would be used to indicate adjacency.
- The algorithm is often more efficient than Sanders's algorithm, but cost-efficiency (node-cost) over Sanders's algorithm is not guaranteed. (The increase in total node cost is generally marginal.)

Note that for the following tabulated runtimes, redundant dummy nodes (left shoe size equal to right shoe size) have been removed and the total node count numbers reflect the result of this procedure.

Seed: 123 789 456	**Sanders's Original Algorithm**		**Speed-optimised Algorithm**	
Number of Nodes	**Time (s)**	**Total Nodes (with repeats)**	**Time (s)**	**Total Nodes**
8	0.00099	12	0.00099	12
10	0.03000	12	0.00099	16
12	0.01799	16	0.00099	20
14	0.565000	24	0.00099	22
16	3.34599	23	0.00099	27
18	- (4)	-	0.00200	28
20	-	-	0.00099	31
50	-	-	0.00500	90
100	-	-	0.01900	187
200	-	-	0.08500	377
500	-	-	0.79800	965
1000	-	-	4.73500	1953

4: the algorithm could not complete within a reasonable timeframe of approx. 1 hour

Table 4.6: Sanders's algorithm vs. the speed-optimised algorithm (1)

Seed: 170 888 264	**Sanders's Original Algorithm**		**Speed-optimised Algorithm**	
Number of Nodes	**Time (s)**	**Total Nodes (with repeats)**	**Time (s)**	**Total Nodes**
8	0.00099	13	0.00099	12
10	0.00699	22	0.00099	13
12	0.14499	24	0.00100	17
14	32.28099	31	0.00099	22
16	39.58400	33	0.00099	25
18	- (5)	-	0.00099	29
20	-	-	0.00099	33
50	-	-	0.00499	89
100	-	-	0.01999	185
200	-	-	0.08800	388
500	-	-	0.79600	972
1000	-	-	4.60600	1944

5: the algorithm could not complete within a reasonable timeframe of approx. 1 hour

Table 4.7: Sanders's algorithm vs. the speed-optimised algorithm (2)

Seed: 923 462 908	**Sanders's Original Algorithm**		**Speed-optimised Algorithm**	
Number of Nodes	**Time (s)**	**Total Nodes (with repeats)**	**Time (s)**	**Total Nodes**
8	0.00099	15	0.00100	12
10	0.00300	16	0.00099	16
12	0.00600	19	0.00099	18
14	0.08800	28	0.00100	22
16	1.19600	27	0.00099	24
18	- (6)	-	0.00100	28
20	-	-	0.00099	33
50	-	-	0.00499	92
100	-	-	0.01900	190
200	-	-	0.08399	386
500	-	-	0.76199	962
1000	-	-	4.62000	1944

6: the algorithm could not complete within a reasonable timeframe of approx. 1 hour

Table 4.8: Sanders's algorithm vs. the speed-optimised algorithm (3)

4.4 Subgraph Algorithm

Suppose a graph, G, exists, with cycles $C = \{c_1, c_2, ..., c_n\}$, even though not all nodes are contained within cycles. It is possible to find an optimal combination of cycles such that the selected combination contains each node exactly once. Thus, existing cycles without duplicates can be isolated and set aside as *completed* and the remaining nodes will be used to construct a subgraph $H \subseteq G$. An existing enlargement method, such as the proposed cost-optimised algorithm, could then be used to enlarge H and ensure that all nodes within H are contained within cycles.

4.4.1 Finding an Optimal Combination of Cycles

Should no existing cycles exist within the graph, the previously proposed cost-optimised algorithm (see Section 4.2, on p. 43) proceeds as normal. Otherwise, an r-combination algorithm could be implemented to find an optimal combination of cycles, with regards to the number of nodes. A combinatorial algorithm is used due to the conjectured NP-completeness of the aforementioned optimisation. (Determining whether the optimisation problem is indeed NP-complete is outside the scope of this dissertation.)

A drawback of the combinatorial method to find a maximal combination is the extremely large number of possible combinations. This problem is compounded by the dramatic growth in the number of cycles (see Section 4.3.1, on p. 67). Assuming that 20 nodes could generate roughly 400 cycles, generating 2-combinations of these cycles would result in $79,800$ unique combinations. Generating 2-combinations of 3000 cycles would result in $4,498,500$ unique combinations. Computer memory is currently cheap, but not infinite, and it is possible for the computer and/or Python interpreter to run out of allocated memory whilst generating these combinations.

4.4.2 Implementation

A pseudocode implementation of the subgraph algorithm is given in Algorithm 16.

4.4.2.1 Example: Sanders's Algorithm vs. the Subgraph Algorithm

Suppose a graph G exists as illustrated in Figure 4.37 (p. 83). The complete list of cycles present in the graph is as follows:

- (1: Adam) $\rightarrow$ (2: Bob)

Algorithm 16 Graph Enlargement: Subgraph Algorithm

```
1: procedure ENLARGE(G)
2:     C ← all cycles c_1, c_2, ..., c_n ∈ G
3:     for r = 1 to len(cycles) do
4:         Generate r-combinations of all cycles
5:     end for

6:     Keep the cycles containing each node at most once
7:     Select the combination containing the maximum number of nodes
8:     SC ← selected cycle combination
9:     SC_v ← all vertices ∈ SC

10:    Construct H ⊆ G, where H_v = G_v − SC_v
11:    Enlarge H using the cost-optimised algorithm (see Section 4.2)
12: end procedure
```

- (1: Adam) → (2: Bob) → (7: George) → (4: David)
- (1: Adam) → (2: Bob) → (7: George) → (5: Eddie)
- (1: Adam) → (6: Frank) → (3: Carol)
- (1: Adam) → (6: Frank) → (3: Carol) → (7: George) → (4: David)
- (1: Adam) → (6: Frank) → (3: Carol) → (7: George) → (5: Eddie)
- (4: David) → (7: George)
- (5: Eddie) → (7: George)
- (9: Ike) → (11: Kenny)

The combination of cycles with maximal length (and with each node appearing at most once in the combination) is:

- (1: Adam) → (6: Frank) → (3: Carol)
- (5: Eddie) → (7: George)
- (9: Ike) → (11: Kenny)

Therefore, the nodes not contained within cycles, and also the nodes that $H \subseteq G$ will be comprised of (see Figure 4.38 on p. 83), are:

- (2: Bob)
- (4: David)

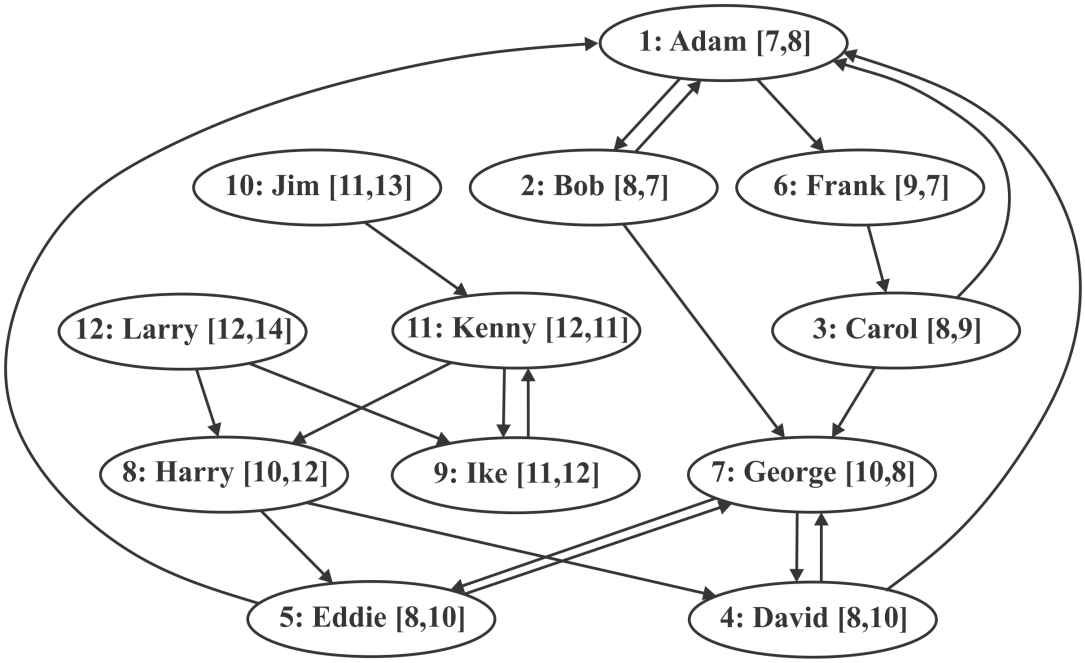


Figure 4.37: An example of a graph for the subgraph algorithm

- (8: Harry)
- (10: Jim)
- (12: Larry)

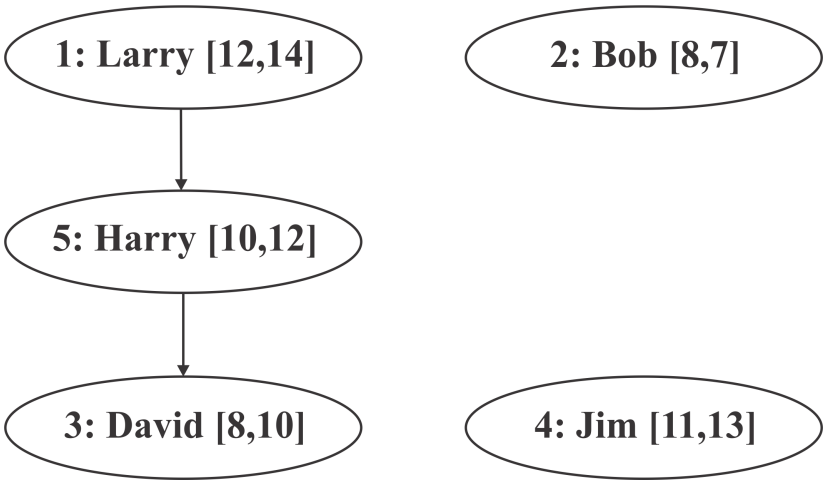


Figure 4.38: The graph $H \subseteq G$

Once the previously proposed cost-optimised algorithm is applied to this graph, H will be enlarged such that all nodes are contained within cycles. The solution is illustrated by Figure 4.39 (p. 84); please note that the nodes already contained within cycles before enlargement are not illustrated.

Note the cases followed by the cost-optimised algorithm. (1: Larry) is a charity node, (3: David) is a greedy node, and (2: Bob) and (4: Jim)

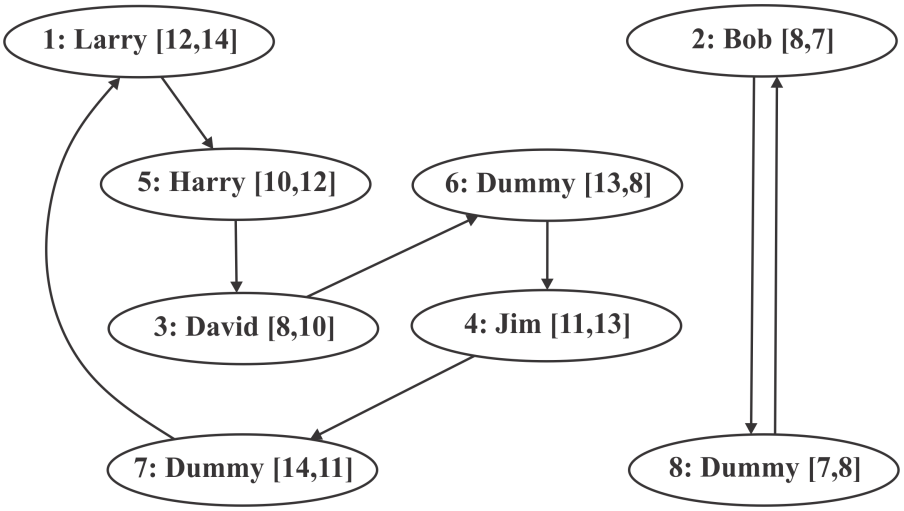


Figure 4.39: The graph H, once enlargement using the subgraph algorithm is complete (please note that the nodes which were contained within cycles pre-enlargement are not illustrated here)

are both isolated nodes. The cost-optimised algorithm identifies a case 7 situation (greedy, charity, and isolated) and creates a cycle containing nodes (1: Larry), (3: David), and (4: Jim). Only (2: Bob) is not contained within a cycle now, and the cost-optimised algorithm identifies a case 2 solution. A dummy node is added to (2: Bob) exclusively to create a cycle.

4.4.3 Interpretation of Results

The running times of the subgraph algorithm as well as the total number of nodes required are tabulated in Tables 4.9, 4.10, and 4.11. The advantages of the subgraph algorithm are as follows:

- If $H \subset G$, then $|V(H)| < |V(G)|$ and the graph H is faster to enlarge than G. The computing power necessary to enumerate and pick the cycles of H is potentially significantly less than for G.
- If the abovementioned holds true, it also implies that larger graphs can be enlarged; that is to say, relatively larger than is usually possible with Sanders's algorithm or the cost-optimised algorithm.

However, the algorithm also suffers from several disadvantages:

- Generating r-combinations on a large number of cycles is a memory-intensive operation and the computer/interpreter runs the risk of running out of allocated memory.
- The algorithm implements additional subroutines to solve problems which are conjectured to be NP-complete.

Seed: 123 789 456	**Sanders's Original Algorithm**			**Subgraph Algorithm**		
Number of Nodes	**Unique Dummy Nodes**	**Time (s)**	**Total Nodes (with repeats)**	**Unique Dummy Nodes**	**Time (s)**	**Total Nodes (with repeats)**
8	2	0.03199	12	3	0.00100	11
10	2	0.01900	12	2	0.00199	12
12	2	0.03699	16	4	0.00900	20
14	2	0.46399	24	2	1.85100	28
16	4	3.01000	23	4	1.83899	26
18	6	-	-	- (7)	-	-

7: out of memory exception

Table 4.9: Sanders's algorithm vs. the subgraph algorithm (1)

Seed: 170 888 264	**Sanders's Original Algorithm**			**Subgraph Algorithm**		
Number of Nodes	**Unique Dummy Nodes**	**Time (s)**	**Total Nodes (with repeats)**	**Unique Dummy Nodes**	**Time (s)**	**Total Nodes (with repeats)**
8	3	0.03299	13	4	0.00099	12
10	6	0.00900	22	3	0.00200	16
12	8	0.16500	24	6	0.00200	20
14	12	32.28800	31	7	0.00400	26
16	6	39.58400	33	9	0.00799	31
18	7	- (8)	-	12	0.02399	42
20	-	-	-	10	30.24000	43
22	-	-	-	11	30.00999	47

8: the algorithm could not complete within a reasonable timeframe of approx. 1 hour

Table 4.10: Sanders's algorithm vs. the subgraph algorithm (2)

Seed: 923 462 908	**Sanders's Original Algorithm**			**Subgraph Algorithm**		
Number of Nodes	**Unique Dummy Nodes**	**Time (s)**	**Total Nodes (with repeats)**	**Unique Dummy Nodes**	**Time (s)**	**Total Nodes (with repeats)**
8	3	0.00099	15	4	0.00099	14
10	4	0.00300	16	4	0.00200	17
12	2	0.00600	19	3	0.00199	15
14	3	0.08800	28	5	0.00499	19
16	3	1.19600	27	5	2.23600	21
18	2	- (9)	-	5	2.83000	25
20	-	-	-	- (10)	-	-
22	-	-	-	-	-	-

9: the algorithm could not complete within a reasonable timeframe of approx. 1 hour

10: out of memory exception

Table 4.11: Sanders's algorithm vs. the subgraph algorithm (3)

4.5 Summary

This chapter covered the experimental results of the proposed optimisations to Sanders's algorithm.

The proposed cost-optimisation algorithm is generally slightly more efficient than Sanders's, but it requires more complex subroutines without achieving major improvements in cost efficiency when looking at the total number of nodes in the graph. For an improvement in the unique number of dummy nodes, the cost-optimised algorithm is more effective.

The proposed speed-optimised algorithm is much more efficient in terms of speed over Sanders's, but occasionally at marginally greater node cost. This is not a common occurence, though. In many scenarios, it is on par with the node cost of the other three algorithms, while sometimes even being more cost-efficient.

The subgraph algorithm focused on first isolating an optimal combination of cycles, before constructing and enlarging a subgraph consisting of the nodes not contained within the selected combination. Overall, the subgraph algorithm displayed an improvement over Sanders's algorithm.

Chapter 5 will provide a greater overview of these results, including summarised results of a few more scenarios (different randomisation seeds).

Chapter 5

Results

Contents

5.1 Introduction

In Chapter 4 the results of new, experimental techniques were detailed and tabulated. It was observed that improvements to the original algorithm by Sanders (2013*a*) were indeed possible for the purpose of solving the shoe matching problem.

This chapter summarises the results observed during the experimental testing of the proposed improvements to Sanders's algorithm. It also covers the advantages of Sanders's original algorithm as a naïve, but efficient solution.

5.2 Cost-optimisation Results

The primary focus of the cost-optimised algorithm was to reduce the total number of nodes necessary to enlarge the graph, such that all nodes would be contained within cycles.

The following methods were proposed and implemented to improve the cost-efficiency of the algorithm:

- Redefining cost distribution (see Section 4.2.1 on page 43), such that the cost of acquiring dummy nodes are shared between all the nodes partaking. In other words, the cost of acquiring additional pairs of shoes are not burdened on a particular individual or group of people (cycle), but is instead shared by all who partake in the experiment.
- Graphs containing only isolated nodes (see Section 4.2.2 on page 43) were enlarged such that the isolated nodes would not be joined to a larger graph structure, unless it was optimal. In cases where it was not optimal, the isolated node would be joined to a single dummy node, to form a cycle of order 2.
- Bridge nodes were avoided as far as possible (see Section 4.2.3 on page 47), by only enlarging graphs per individual component. Components were never linked, preventing the creation of additional bridge nodes and compounding the problem.
- Rule-based decision making (see Section 4.2.4 on page 48) was introduced to the algorithm. The rules consisted of optimal solutions depending on the current situation of the nodes in the graph.
- Cycle compression (see Section 4.2.5 on page 54) allowed repeated sequences of nodes within cycles to be compressed, thereby saving on node cost. The longest common substring algorithm played a pivotal role in developing the cycle compression algorithm.

The results of these methods can be found in Section 4.2.7 on page 64. In summary, the cost-optimised algorithm *did* save on node cost to some extent, but the increased complexity of the new algorithm is a drawback.

5.3 Speed-optimisation Results

The primary focus of the speed-optimised algorithm was to dramatically decrease the running time of the algorithm, and to allow large graphs (1000+ nodes) to be enlarged.

The following methods were proposed and implemented to improve the speed-efficiency of the algorithm:

- The dramatic growth in the number of cycles (see Section 4.3.1 on page 67) was investigated and it was proposed that a solution should be found which did not rely on cycle enumeration or cycle picking. It was also proposed that cycle enumeration should not be necessary to ensure that all nodes were indeed contained within cycles post-enlargement.

- The benefits and background of permutation matrices (see Section 4.3.2 on page 67), and their applicability to this dissertation, were investigated and presented.
- A new method to transform an adjacency matrix to a permutation matrix (see Section 4.3.3 on page 70) was proposed. The permutation matrix has several advantages:
  - There is no need for cycle enumeration.
  - There is no need for cycle picking.
  - There is no need to check for bridge nodes.
  - There is no need to check for repeated nodes.

The results of these methods can be found in Section 4.3.5 on page 75. In summary, the speed-optimised algorithm is **highly** efficient, but is occasionally marginally more expensive to implement in terms of node-cost than Sanders's algorithm. However, in plenty of scenarios, it has even proved to be more cost-efficient than Sanders's algorithm.

5.4 Subgraph Algorithm Results

The subgraph algorithm focused on first isolating an optimal combination of cycles, before constructing and enlarging a subgraph consisting of the nodes not contained within the selected combination.

The results of these methods can be found in Section 4.4.3 on page 84. In summary, the subgraph algorithm showed positive results, being one of the most efficient with regards to minimising the total number of nodes in the graph, including repeated nodes across multiple cycles. Also see the summarised performance of this algorithm applied to additional scenarios in Section 5.6.

The subgraph algorithm required the addition of extra subroutines, making it a possibly less attractive solution than the other proposed algorithms.

5.5 The Advantages of the Naïve Solution

In essence, Sanders's original algorithm still provides a good solution to the shoe matching problem, albeit applicable to smaller graphs only. The naïve solution of simply adding dummy nodes where necessary is acceptably cost-efficient, in general, and is easy to implement in code. The cost-optimised and subgraph algorithms save on dummy nodes at the cost of repeated cycles, requiring a cycle compression algorithm to counter the

aforementioned consequence. The minor savings on node-cost require greater code complexity and additional subroutines, but it can be argued that the main point of the shoe matching algorithm is to help the participants save on cost, making the proposed cost-optimised algorithm or the subgraph algorithm the ideal solution. The speed optimised algorithm is a better fit in the case of a large population of participants as neither Sanders's algorithm, the cost-optimised algorithm, nor the subgraph algorithm are capable of handling such large graphs. The total node cost of the speed-optimised algorithm is generally on par with Sanders's original algorithm.

5.6 Comparative Results Across All Algorithms

Table 5.1 provides an overview of the results (total number of nodes in the graph, including repeated nodes in multiple cycles) across all four algorithms. The most **cost-efficient solution** (not necessarily the fastest) is indicated in ***bolded italics***. The original algorithm, at best, ties with at least one of the optimised algorithms.

Table 5.2 contains the number of unique dummy nodes (repeated nodes across multiple cycles are **not** counted) across all four algorithms for the different seed and size combination. The results are very interesting: whilst the cost-optimised algorithm has the fewest number of unique dummy nodes in 22 of 25 quantifiable cases (88%), it has the fewest number of total nodes in only 9 of 25 quantifiable cases (36%). It is clear that fewer unique dummy nodes are not necessarily indicative of fewer total nodes in the graph.

Tables 5.3 and 5.4 contain summarised success rates across all graph sizes and seeds laid out in tables 5.1 and 5.2. The success rate is given as a percentage, and represents the ratio of optimal cases (least number of nodes, either in the form of unique dummy nodes or total nodes) against all quantifiable result cases for each algorithm. In the case of an OutOfMemory exception, the result is left blank and does not count towards the algorithms success rate.

Please see Appendix A (p. 106) for additional comparative results between the four algorithms.

Seed	Nodes	Sanders's Algorithm (Total Nodes)	Cost-optimised Algorithm (Total Nodes)	Speed-optimised Algorithm (Total Nodes)	Subgraph Algorithm (Total Nodes)
123 789 456	8	12	***11***	12	***11***
	10	***12***	***12***	16	***12***
	12	***16***	***16***	20	20
	14	24	***21***	22	28
	16	23	***22***	27	26
	18			***28***	
170 888 264	8	13	13	***12***	***12***
	10	22	20	***13***	16
	12	24	25	***17***	20
	14	31	27	***22***	26
	16	33	32	***25***	31
	18			***29***	42
923 462 908	8	15	14	***12***	14
	10	16	***15***	16	17
	12	19	19	18	***15***
	14	28	27	22	***19***
	16	27	27	24	***21***
	18			28	***25***
482 367 498	8	***12***	***12***	13	***12***
	10	18	19	***15***	19
	12	21	25	***19***	***19***
	14	22	26	22	***19***
	16	37	30	26	***25***
	18	35		30	***27***
532 499 987	8	14	15	***13***	***13***
	10	17	16	16	***15***
	12	19	19	19	***18***
	14	***20***	***20***	22	21
	16	***25***	***25***	***25***	***25***
	18			29	***28***

Table 5.1: Comparative results across all algorithms (total number of nodes)

Seed	Nodes	Sanders's Algorithm (Unique Dummies)	Cost-optimised Algorithm (Unique Dummies)	Speed-optimised Algorithm (Unique Dummies)	Subgraph Algorithm (Unique Dummies)
123 789 456	8	***2***	***2***	4	3
	10	***2***	***2***	6	***2***
	12	***2***	***2***	8	4
	14	***2***	***2***	8	***2***
	16	***4***	***4***	11	***4***
	18			***10***	
170 888 264	8	***3***	***3***	4	4
	10	6	5	***3***	***3***
	12	8	***4***	5	6
	14	12	8	8	***7***
	16	6	***4***	9	9
	18			***11***	12
923 462 908	8	***3***	4	4	4
	10	4	***3***	6	4
	12	***2***	***2***	6	3
	14	***3***	***3***	8	5
	16	***3***	***3***	8	5
	18			10	***5***
482 367 498	8	***4***	***4***	5	***4***
	10	***4***	***4***	5	***4***
	12	***5***	***5***	7	***5***
	14	***5***	***5***	8	***5***
	16	***3***	***3***	10	6
	18	***3***		12	6
532 499 987	8	***4***	***4***	5	5
	10	4	***3***	6	5
	12	***3***	***3***	7	6
	14	***3***	***3***	8	5
	16	***4***	***4***	9	7
	18			11	***7***

Table 5.2: Comparative results across all algorithms (unique number of dummy nodes)

Sanders's Algorithm (Total Nodes Success Rate %)	Cost-optimised Algorithm (Total Nodes Success Rate %)	Speed-optimised Algorithm (Total Nodes Success Rate %)	Subgraph Algorithm (Total Nodes Success Rate %)
19	36	40	59

Table 5.3: Summarised comparative results across all algorithms (minimising the total number of nodes)

Sanders's Algorithm (Unique Dummies Success Rate %)	Cost-optimised Algorithm (Unique Dummies Success Rate %)	Speed-optimised Algorithm (Unique Dummies Success Rate %)	Subgraph Algorithm (Unique Dummies Success Rate %)
77	88	7	34

Table 5.4: Summarised comparative results across all algorithms (minimising the unique number of dummy nodes)

5.7 Summary

To summarise, the new algorithms were overall more efficient than Sanders's original algorithm. At best, Sanders's original algorithm tied with one of the three new algorithms.

The cost-optimised algorithm was successful in minimising the unique number of dummy nodes necessary, but this led to an increase in repeated nodes across multiple cycles. It is apparent that fewer unique dummy nodes does not always imply that the total number of nodes required to satisfy the shoe matching problem will decrease. However, it is by far the most successful algorithm in achieving a fully cyclical graph whilst minimising the number of unique dummy nodes.

The speed-optimised algorithm greatly decreased the running times of the algorithm, without incurring a major decrease in cost-efficiency (node-cost). The speed-optimised algorithm is capable of handling large graphs (1000+ nodes) and has no need of cycle enumeration, cycle picking, or cycle compression algorithms. By discarding these CPU intensive operations, the algorithm is able to solve the shoe matching problem on graphs with an order of 1000 nodes within a matter of seconds on a modern desktop computer.

The subgraph algorithm has shown promising results overall. In some

cases it is less efficient than Sanders's algorithm, but across 5 unique scenarios (different randomisation seeds, see Section 5.6), it has the highest success rate in terms of minimising the total node cost.

Chapter 6

Future Work

Contents

6.1 Introduction

The work contained within this dissertation is largely theoretical: whilst pseudocode is provided to reproduce the results, these specific algorithms have not yet been implemented in a real-life scenario. In addition, several topics related to the work contained within this dissertation were outside the scope of this research endeavour. Below are listed some major topics to look at in order to improve on the work done already:

- Ideas on Future Algorithms
- NP-completeness of the problem and algorithms
- Porting the code to a lower level language
- A practical implementation of the algorithm

The following sections examine each of these in greater detail, in turn.

6.2 Ideas on New Algorithms And Improvements to Existing Algorithms

Listed below are some ideas to improve the algorithms covered in this dissertation, as well as new ideas for future algorithm development.

6.2.1 Involving Minimum Spanning Trees

A possible future solution could involve the construction of a minimum spanning tree (referred to as "MSP" hereafter). It would then be possible to split the newly constructed MSP at any branching points. Whenever a node, say *n*, branches into multiple nodes, break the redundant branches to construct separate components, replacing node *n* with dummy nodes where required. For example, if node *n* branched into 3 new nodes, 2 of the edges (branches) need to be broken and *n* should be replaced with 2 repeats of *n* as dummy nodes for the new components this process creates. It should then be simple to create a cycle from these separated components (previously branches). See Figures 6.1 and 6.2 for an illustration of the process.

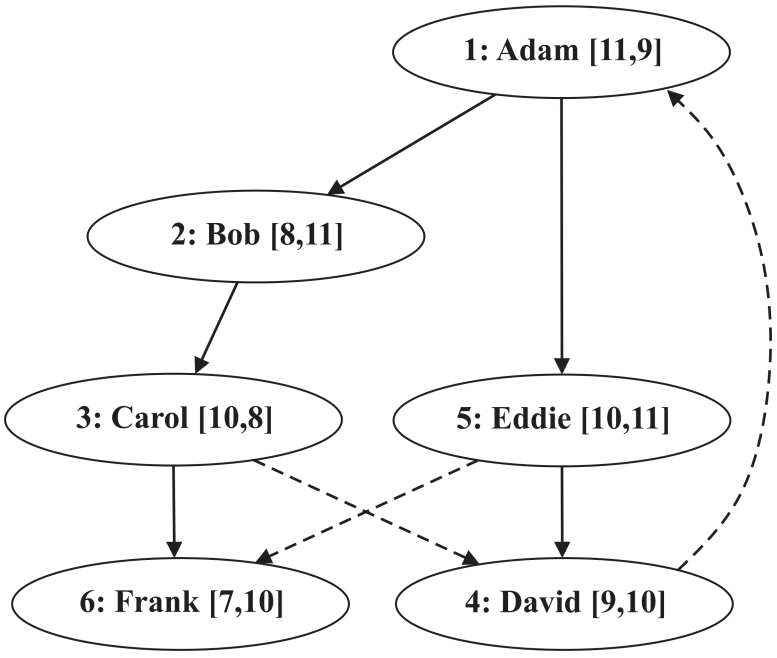


Figure 6.1: An example of a graph for the spanning tree algorithm, pre-enlargement (dotted lines indicate non-MSP branches that were originally present within the graph)

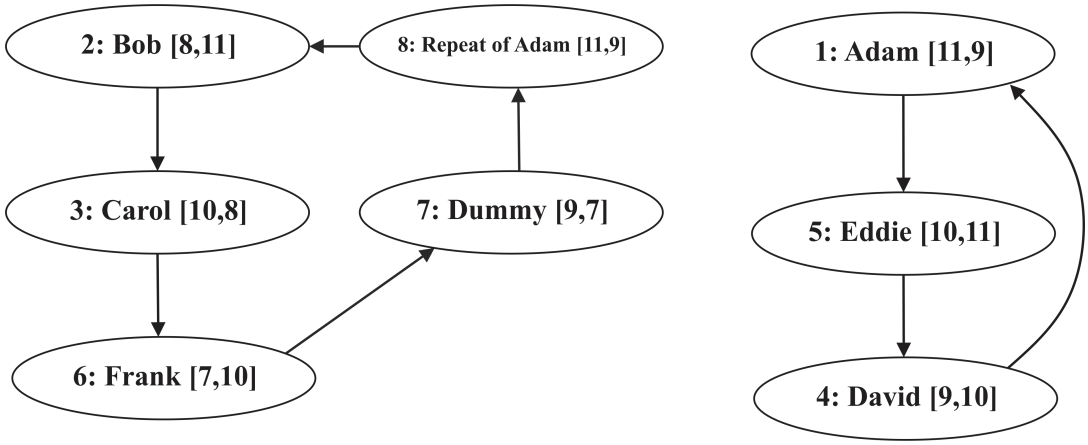


Figure 6.2: An example of a graph for the spanning tree algorithm, post-enlargement; note the repeat of Adam's node after breaking the branch

6.2.2 Calculated Selection of Intermediate Nodes

Currently whenever the cost-optimised algorithm or Sanders's algorithm requires "some node" as an intermediate step in enlarging the graph, a node is arbitrarily selected. It is possible that vertices with lower degrees may be a better choice, in order to reduce repeated nodes across multiple cycles. It may also argued be that nodes with a higher degree are a better choice since this allows more cycle combination choices for the cycle picking processes.

6.2.3 Improvements to the Subgraph Algorithm

The subgraph algorithm currently makes use of the cost-optimised algorithm to enlarge the subgraph $H \subseteq G$. It may be worth exploring whether using the speed-optimised algorithm (or another enlargement algorithm) instead will improve the node cost- and speed-efficiency of the subgraph algorithm.

6.2.4 Divide and Conquer with Parallel Computing

Parallel computing refers to the method of computation wherein a problem is divided into smaller subproblems, and then each subproblem is solved separately, in parallel, to produce a final, combined solution. Investigating whether subroutines such as cycle enumeration and cycle picking could be parallelised, could potentially lead to great performance improvements.

In the case of component-wise enlargement, each component can be enlarged in parallel. Parallelising the subroutines can also enable the algorithms to be run on a computing cluster.

6.3 NP-completeness

Several problems related to graph enlargement are potentially NP-complete, including:

- Cycle picking (see Section 2.5 on p. 20 and Section 3.8 on p. 37)
- Selecting the optimal combination of cycles with each node appearing at most once (see Section 4.4.1 on p. 81)

Investigating the NP-completeness of these problems, and possibly optimising the subroutines designed to solve them, should be undertaken in future.

6.4 Porting to an Alternative Language

Python is a language which is logical to write and easy to implement, but its performance is slower than that of lower level languages. Python is not strongly typed and the Python interpreter needs to determine the type of each object (integer, float, string, etc.) before it can manipulate the object.

The code could be ported to several other languages, including:

- Go: Google's new language resembles Python in several ways, including syntactically, but is optimised for parallel computing and objects are cast into their respective types at compile time. This allows the compiled binaries to be executed much faster than they would have been interpreted by Python. The ease of parallelising tasks could mean that certain algorithms, such as cycle enumeration, could be processed at a much faster speed.
- C/C++: A lower level, strongly-typed language should provide much better performance than the Python interpreter.

6.5 Practical Implementation

A good project for a third-year or honours level student might be to implement the algorithm in a practical manner, perhaps in the form of a website with a database backend. An administrator could add specific shoe types that people could express interest in. Users can register on the site and select a shoe design of their choice. A graph would be constructed per certain unique constraints such as shoe type, intended gender, etc. The website should then automatically match people using the algorithms discussed in this dissertation. Similar work on the practical, real-life application of the algorithm has been done by Sanders (2013*b*).

Chapter 7

Conclusion

This dissertation focused on graph enlargement from a different perspective: by adding vertices to a graph instead of focusing on edge augmentation as originally defined by Eswaran and Tarjan (1976). Traditionally graph augmentation had always been focused on edge augmentation. Graph enlargement via the addition of vertices to the graph was thus seen as an opportunity to extend the field. Sanders (2013*a*) had already done valuable work in this regard and laid the foundation for the content in this dissertation.

The original research aim of this dissertation was to improve on the work done by Sanders (2013*a*) and was stated as follows:

> **To develop an improved algorithm(s) for the graph enlargement problem**; in terms of:
>
> - Faster run time and the capability to handle larger graphs
> - Fewer total nodes required to enlarge the graph such that all nodes are contained within cycles

Multiple optimisations were proposed, split between cost-efficiency and speed-efficiency, as well as lessening the input size (by enlarging only a subgraph consisting of nodes not contained within cycles). The optimised algorithms were effective and efficient overall, and it can be left up to the user to decide which algorithm is ideally suited to their problem.

For the cost-optimisation algorithm, it was found that by restricting graph enlargement to component-wise enlargement, the creation of bridge nodes between components could be stifled. For strictly isolated nodes, instead of adding two dummy nodes to incorporate it into a larger existing cycle, it is more cost-efficient to add a single dummy node. This creates a 2-node cycle between the previously isolated node and the new dummy node. Due to the decrease in unique number of dummy nodes, there was a greater chance for some nodes to be more regularly repeated within cy-

cles. The addition of a cycle compression subroutine helped to minimise the total number of nodes in the graph by compressing redundant repeats of nodes. It can also be argued that the rule-based decision making techniques of the cost-optimised algorithm add a semblance of **intelligence** to the algorithm.

For the speed-optimised algorithm, the bottlenecks in the original algorithm and the new cost-optimised algorithm had to be identified. Cycle enumeration and cycle picking were found to have an extremely negative impact on performance. The growth in the number of cycles can be close to exponential in relation to the number of nodes in the graph. The possibility of cycle picking being an NP-complete problem required all the possible combinations of cycles to be enumerated. By transforming the adjacency matrix of the graph to a permutation matrix, it was possible to mathematically ensure unique, disjoint cycles without the need for cycle picking and cycle enumeration algorithms. The algorithm is occasionally accompanied by marginally increased total node cost (only in some instances), but the powerful increase in performance could outweigh this drawback depending on the user's need.

The results of the subgraph algorithm (see Section 5.4) were positive overall. The algorithm proceeded by creating a subgraph $H \subseteq G$ of the original graph. H consisted of all the nodes not contained within an optimal combination (greatest number of nodes) of existing cycles. The graph H was then augmented using the new cost-optimised algorithm. Experimental trials of the algorithm showed better cost- and speed-efficiency than Sanders's algorithm across several scenarios.

Despite the efficiency of the optimisations, the original, naïve solution is also still a good fit to the original shoe matching problem and possesses several advantages (see Section 5.5, p. 90), notably a good compromise between speed, implementation complexity, and efficiency, although due to the exponential growth of the number of cycles in a graph (see Section 4.3.1), and the NP-completeness of certain problems (e.g. cycle picking), it becomes impractical to implement on larger graphs. In this dissertation, given the nature of the synthetic data (see Section 3.5), graph size was capped at roughly 24 nodes, although sometimes even fewer. The matrix enlargement method (the core of speed-optimised algorithm) takes care of the enlargement of large graphs, with 1000+ nodes. The idea of transforming an adjacency matrix to a permutation matrix for the purpose of graph enlargement is a novel and innovative procedure.

Graph enlargement, is still a new concept with plenty of potential for future research opportunities and possible applications to real life problems.

References

Bondy, J. A. and Murty, U. S. R. (2008), *Graph Theory (Graduate Texts in Mathematics)*, 1st edn, Springer, London, UK.

Chartrand, G., Lesniak, L. and Zhang, P. (2011), *Graphs and Digraphs*, 5th edn, Chapman and Hall/CRC Press, Boca Raton, Florida.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009), *Introduction to Algorithms, Third Edition*, 3rd edn, The MIT Press.

Egeland, G. and Engelstad, P. E. (2009), The economy of redundancy in wireless multi-hop networks, *in* 'Proceedings of the 2009 IEEE Conference on Wireless Communications & Networking Conference', WCNC'09, IEEE Press, Piscataway, NJ, USA, pp. 3023–3028.
URL: *http://dl.acm.org/citation.cfm?id=1688345.1688872*

Eswaran, K. (1973), *Representation of Graphs and Minimally Augmented Eulerian Graphs with Applications in Data Base Management*, Research reports // IBM, IBM Thomas J. Watson Research Division.
URL: *http://books.google.co.za/books?id=41bJtgAACAAJ*

Eswaran, K. P. and Tarjan, R. E. (1976), 'Augmentation problems', *SIAM Journal on Computing* **5**(4), 653–665.
URL: *http://dx.doi.org/10.1137/0205044*

Even, S. (2011), *Graph Algorithms*, 2nd edn, Cambridge University Press, New York, NY, USA.

Fraleigh, J. B. (2003), *A First Course In Abstract Algebra*, 7th edn, Pearson Education Inc.

Frank, A. (1990), Augmenting graphs to meet edge-connectivity requirements, *in* 'FOCS', IEEE Computer Society, pp. 708–718.

Frank, H. and Chou, W. (1970), 'Connectivity considerations in the design of survivable networks', *Circuit Theory, IEEE Transactions on* **17**(4), 486–490.

Goodman, S. E., Hedetniemi, S. T. and Slater, P. J. (1975), 'Advances on the hamiltonian completion problem', *J. ACM* **22**(3), 352–360.
URL: *http://doi.acm.org/10.1145/321892.321897*

Goodman, S. and Hedetniemi, S. (1974), On the hamiltonian completion problem, *in* R. A. Bari and F. Harary, eds, 'Graphs and Combinatorics', Vol. 406 of *Lecture Notes in Mathematics*, Springer Berlin Heidelberg, pp. 262–272.
URL: *http://dx.doi.org/10.1007/BFb0066448*

Hawick, K. A. and James, H. A. (2008), Enumerating Circuits and Loops in Graphs with Self-Arcs and Multiple-Arcs, *in* 'Proceedings of the 2008 International Conference on Foundations of Computer Science', CSREA, Las Vegas, USA, pp. 14–20.

Hopcroft, J. and Tarjan, R. (1973), 'Algorithm 447: Efficient algorithms for graph manipulation', *Communications of the ACM* **16**(6), 372–378.
URL: *http://doi.acm.org/10.1145/362248.362272*

Johnson, D. B. (1975), 'Finding All the Elementary Circuits of a Directed Graph.', *SIAM Journal on Computing Comput.* **4**(1), 77–84.
URL: *http://dblp.uni-trier.de/db/journals/siamcomp/siamcomp4.html#Johnson75*

Johnsonbaugh, R. (2000), *Discrete Mathematics*, 5th edn, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Kruskal, J. (1956), 'On the shortest spanning tree of a graph and the travelling salesman problem', *Proceedings of the American Mathematical Society* **7**(1), 48–50.

Liu, H. and Wang, J. (2006), A New Way to Enumerate Cycles in Graph, *in* 'Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services', AICT-ICIW '06, IEEE Computer Society, Washington, DC, USA, pp. 57–59.
URL: *http://dl.acm.org/citation.cfm?id=1116162.1116223*

Naor, D., Gusfield, D. and Martel, C. U. (1997), 'A fast algorithm for optimally increasing the edge connectivity', *SIAM J. Comput.* **26**(4), 1139–1165.

Prim, R. (1957), 'Shortest connection networks and some generalizations', *Bell System Technical Journal* **36**(6), 1389–1401.

Rosenthal, A. and Goldner, A. (1977), 'Smallest augmentations to biconnect a graph', *SIAM J. Comput.* **6**(1), 55–66.

Rotman, J. J. (1994), *Graduate Texts in Mathematics: An Introduction to the Theory of Groups*, 4th edn, Springer.

Sanders, I. (2013*a*), Cooperating to buy shoes: An application of picking cycles in directed graphs, *in* 'Proceedings of the South African Institute of Computer Scientists and Information Technologists (Theme: "A Connected Society")', East London, pp. 8–16.
URL: *http://dl.acm.org/citation.cfm?id=2517086*

Sanders, I. (2013*b*), Cooperating to buy shoes in the real world: online cycle picking in directed graphs, *in* 'Proceedings of the South African Institute of Computer Scientists and Information Technologists (Theme: "A Connected Society")', East London, pp. 286–294.
URL: *http://dl.acm.org/citation.cfm?id=2513474*

Sheng Hsu, T. and Ramachandran, V. (1993), 'Finding a smallest augmentation to biconnect a graph', *SIAM J. Comput.* **22**(5), 889–912.

Tarjan, R. E. (1972), Enumeration of the Elementary Circuits of a Directed Graph, Technical report, Ithaca, NY, USA.

Tiernan, J. C. (1970), 'An Efficient Search Algorithm to Find the Elementary Circuits of a Graph', *Communications of the ACM* **13**(12), 722–726.
URL: *http://doi.acm.org/10.1145/362814.362819*

Weinblatt, H. (1972), 'A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph', *Journal of the ACM* **19**(I), 43–56.

Xulvi-Brunet, R. and Sokolov, I. M. (2007), 'Growing networks under geographical constraints', *Phys. Rev. E* **75**, 046117.
URL: *http://link.aps.org/doi/10.1103/PhysRevE.75.046117*

Appendices

Appendix A

Additional Comparative Results Across Algorithms

These appendices present additional shoe matching scenarios (different randomisation seeds and graph sizes). The results of all four algorithms are displayed side by side and compared to one another.

The algorithm(s) achieving the fewest number of total nodes for each scenario has its result highlighted in ***bolded italics***. In the case of an OutOfMemory exception, the result is left blank and does not count towards the algorithms success rate.

Success rates are expressed as percentages and represent the following ratio: the number of times the algorithm has solved the shoe matching problem with the fewest number of nodes (compared to its competitors), divided by the total number of quantifiable results the algorithm produced. The resulting percentage is then rounded to the nearest integer.

The rest of this chapter contains the results of 20 randomly picked seeds, divided into subgroups of 5 seeds per page. Table A.1 contains a summary of these results.

20 seeds, 6 graph sizes per seed = 120 scenarios	**Sanders's Algorithm (Total Nodes)**	**Cost-optimised Algorithm (Total Nodes)**	**Speed-optimised Algorithm (Total Nodes)**	**Subgraph Algorithm (Total Nodes)**
Success Rate % (Minimising Total Nodes)	**20**	**37**	**53**	**57**

Table A.1: Summarised additional comparative results across all algorithms

Seed	Nodes	Sanders's Algorithm (Total Nodes)	Cost-optimised Algorithm (Total Nodes)	Speed-optimised Algorithm (Total Nodes)	Subgraph Algorithm (Total Nodes)
586 015 916	8	17	***14***	***14***	***14***
	10	***16***	***16***	***16***	***16***
	12	***19***	***19***	20	20
	14	***22***	26	23	***22***
	16	***26***		***26***	27
	18	***31***		***31***	***31***
273 100 389	8	22	15	***14***	15
	10	26	***17***	18	***17***
	12	23	23	21	***20***
	14	23	***22***	23	***22***
	16	***26***	***26***	27	***26***
	18	31	***30***	31	***30***
513 577 387	8	25	***14***	***14***	***14***
	10	27	18	***16***	18
	12	35	24	***20***	24
	14	38	28	***23***	28
	16	37	30	***26***	30
	18	45	35	***30***	38
912 644 420	8	***10***	***10***	13	***10***
	10	17	***15***	17	***15***
	12	25	***20***	***20***	***20***
	14	25	***21***	22	25
	16			***25***	
	18			***29***	
422 223 344	8	12	13	***10***	13
	10	18	***15***	***15***	***15***
	12	25	20	***19***	20
	14	29	25	***23***	29
	16	30		25	***22***
	18			***29***	37
Success Rate % (Minimising Total Nodes)		**26**	**54**	**63**	**54**

Table A.2: Additional comparative results across all algorithms (1)

Seed	Nodes	Sanders's Algorithm (Total Nodes)	Cost-optimised Algorithm (Total Nodes)	Speed-optimised Algorithm (Total Nodes)	Subgraph Algorithm (Total Nodes)
550 095 080	8	15	***11***	13	***11***
	10	15	15	15	***13***
	12	21	18	18	***16***
	14	24	***20***	22	27
	16	26	23	27	***21***
	18			***30***	
444 662 587	8	15	14	***13***	14
	10	***14***	15	17	17
	12	***19***	20	21	22
	14	25	***23***	24	***23***
	16	29		***26***	
	18			***30***	
664 788 481	8	***13***	***13***	***13***	***13***
	10	16	16	***15***	***15***
	12	21	***19***	***19***	***19***
	14	25	***23***	***23***	***23***
	16	28	***26***	28	30
	18		30	***28***	44
461 731 130	8	***11***	***11***	12	12
	10	***12***	***12***	14	14
	12	***13***	***13***	17	***13***
	14	21	***20***	21	
	16		24	***22***	
	18		***23***	29	
735 161 101	8	14	13	***11***	17
	10	22	21	15	***14***
	12	31	23	***19***	***19***
	14	27	24	22	***20***
	16	24		26	***23***
	18			***29***	
Success Rate % (Minimising Total Nodes)		**25**	**48**	**43**	**61**

Table A.3: Additional comparative results across all algorithms (2)

Seed	Nodes	Sanders's Algorithm (Total Nodes)	Cost-optimised Algorithm (Total Nodes)	Speed-optimised Algorithm (Total Nodes)	Subgraph Algorithm (Total Nodes)
475 887 106	8	27	12	***11***	***11***
	10	31	15	15	***14***
	12	33	29	***19***	22
	14		30	23	***22***
	16			***25***	
	18			***28***	
664 469 990	8	14	13	12	***11***
	10	14	14	14	***13***
	12	17	17	17	***16***
	14	23	22	21	***19***
	16			***25***	26
	18			***28***	39
965 309 708	8	14	15	***12***	***12***
	10	23	21	***15***	16
	12	27	25	***19***	20
	14	27	24	23	***22***
	16	***21***	23	26	***21***
	18	***30***		***30***	***30***
456 777 039	8	18	13	***12***	13
	10	***13***	***13***	14	***13***
	12	22	***18***	***18***	***18***
	14	***19***	***19***	22	20
	16	22	***21***	26	***21***
	18	***27***	***27***	29	
222 892 396	8	12	12	***10***	***10***
	10	19	16	15	***14***
	12	28	24	***18***	23
	14	34	24	***22***	23
	16	36	***26***	27	30
	18			***29***	
Success Rate % (Minimising Total Nodes)		**21**	**25**	**53**	**62**

Table A.4: Additional comparative results across all algorithms (3)

Seed	Nodes	Sanders's Algorithm (Total Nodes)	Cost-optimised Algorithm (Total Nodes)	Speed-optimised Algorithm (Total Nodes)	Subgraph Algorithm (Total Nodes)
809 723 839	8	14	15	***13***	15
	10	***17***	18	***17***	18
	12	26	22	***19***	***19***
	14	28	***23***	***23***	***23***
	16	31	28	25	***24***
	18		34	30	***26***
508 456 853	8	13	***12***	13	***12***
	10	18	***15***	16	17
	12	20	19	19	***18***
	14	26	***22***	24	***22***
	16	31		***27***	
	18	34		***29***	
393 236 722	8	14	14	***12***	14
	10	15	15	14	***13***
	12	21	21	***17***	18
	14	19	19	18	***17***
	16	21	21	25	***19***
	18			29	***28***
568 828 692	8	13	14	13	***12***
	10	18	18	***16***	19
	12	23	20	***19***	***19***
	14	24	***21***	23	22
	16	***24***		27	25
	18			31	***29***
346 854 015	8	18	14	***12***	14
	10	24	18	***16***	18
	12	27	23	***18***	19
	14	28	31	***22***	24
	16	37	46	***26***	29
	18			***30***	34
Success Rate % (Minimising Total Nodes)		**8**	**21**	**53**	**50**

Table A.5: Additional comparative results across all algorithms (4)

Appendix B

Ethical Clearance

Please see page 112 for the ethical clearance document.

UNISA | college of science, engineering and technology

Dear Mr Jan Johannes van der Linde (48412449)

Date: 2015-02-25

Application number:
033/JJVDL/2015

REQUEST FOR ETHICAL CLEARANCE: (Augmenting directed graphs to ensure all nodes are contained in cycles)

The College of Science, Engineering and Technology's (CSET) Research and Ethics Committee has considered the relevant parts of the studies relating to the abovementioned research project and research methodology and is pleased to inform you that ethical clearance is granted for your research study as set out in your proposal and application for ethical clearance.

Therefore, involved parties may also consider ethics approval as granted. However, the permission granted must not be misconstrued as constituting an instruction from the CSET Executive or the CSET CRIC that sampled interviewees (if applicable) are compelled to take part in the research project. All interviewees retain their individual right to decide whether to participate or not.

We trust that the research will be undertaken in a manner that is respectful of the rights and integrity of those who volunteer to participate, as stipulated in the UNISA Research Ethics policy. The policy can be found at the following URL:
http://cm.unisa.ac.za/contents/departments/res_policies/docs/ResearchEthicsPolicy_apprvCounc_21Sept07.pdf

Please note that the ethical clearance is granted for the duration of this project and if you subsequently do a follow-up study that requires the use of a different research instrument, you will have to submit an addendum to this application, explaining the purpose of the follow-up study and attach the new instrument along with a comprehensive information document and consent form.

Yours sincerely

Prof Ernest Mnkandla
Chair: College of Science, Engineering and Technology Ethics Sub-Committee

Prof IOG Moche
Executive Dean: College of Science, Engineering and Technology

RECEIVED
2015 -02- 26
OFFICE OF THE EXECUTIVE DEAN
College of Science, Engineering and Technology

University of South Africa
College of Science, Engineering and Technology
The Science Campus
C/o Christiaan de Wet Road and Pioneer Avenue,
Florida Park, Roodepoort
Private Bag X6, Florida, 1710
www.unisa.ac.za/cset

UNISA | college of science, engineering and technology

Appendix C

SAICSIT 2015 Submission

A paper based on the work contained within this dissertation was accepted for the SAICSIT 2015 conference and published in the associated proceedings. Please see the next page for the paper.

Enlarging Directed Graphs To Ensure All Nodes Are Contained In Cycles

J. J. van der Linde
School of Computing
University of South Africa
UNISA Science Campus, Florida
janvdl@outlook.com

I. D. Sanders
School of Computing
University of South Africa
UNISA Science Campus, Florida
sandeid@unisa.ac.za

ABSTRACT

Many algorithms in graph theory add or remove either edges or nodes (or both) to solve a given problem. Graph augmentation typically concerns the addition of edges to a graph to satisfy some connectivity property of the graph. This paper focuses on the addition of vertices to a graph to satisfy a specific connectivity property: ensuring that all the nodes of the graph are contained within cycles. A distinction is made between graph augmentation (edge addition), and graph enlargement (vertex addition).

The particular problem addressed here is the enlarging of a digraph which is an abstraction defined in the "shoe matching problem" and represents people who require different sizes of shoes. To be able to satisfy all of the participants, every node (person) in the digraph must be contained in at least one cycle. This paper looks at ways to improve on the original approach to *graph enlargment*. It redefines the cost model used in the original work, presents three improvements to the original approach and shows that these approaches do indeed offer benefits in terms of the number of nodes needed to solve the problem and/or the speed of enlargement.

Categories and Subject Descriptors

G.2.2 [**Mathematics of Computing**]: Graph Theory—*Graph Algorithms*

Keywords

directed graphs, graph enlargement, cycle picking

1. INTRODUCTION

Many problems in graph theory involve changing the structure of the graph by (implictly or explicitly) adding or removing edges or nodes. For example, minimum spanning tree algorithms discard some edges [1], biconnectedness can be achieved by adding or removing edges [1], etc. In some real world situations like computer networking nodes and edges can be added to increase redundancy and hence reliability [2, 14]. *Graph augmentation*, in general, relates to finding the number of vertices or edges that should be *added* to a certain graph to satisfy a particular connectivity property [4, 3, 7, 11]. Graph augmentation has traditionally been associated with the addition of edges to a graph [4], and thus this paper refers to graph augmentation via the addition of vertices and related edges as *graph enlargement* instead. The particular graph enlargement problem addressed in this paper stems from the shoe matching problem [10].

The shoe matching problem [10] involves people who require different sizes of shoes between their left and right feet. Owing to the fact that shoes are sold as pairs, a person requiring different sized shoes would have to buy 2 pairs of shoes in order to satisfy their needs. By cooperating with other individuals with a similar need, it would be possible for participants to save money. If Adam requires shoes of sizes $(8L, 10R)$ and Bob requires shoes of sizes $(10L, 8R)$, it would be possible for Adam and Bob to team up. If Adam buys one pair of size 10 shoes, and Bob buys a pair of size 8 shoes, they could each swap one of the shoes with each other. This allows both Adam and Bob to satisfy their specific footwear needs at the cost of a single pair of shoes each. If this straightforward one-to-one matching is not possible then the problem becomes that of finding a "cycle" of people such that enough pairs of shoes could be bought and everyone's needs would be satisfied. For example, if Tom's requirements are $(10L, 8R)$, Fred's requirements are $(8L, 9R)$ and Monde's requirements are $(9L, 10R)$ then buying three pairs of shoes (sizes 8, 9 and 10) would satisfy all three parties.

Sanders [10] abstracted the shoe matching problem to a graph theory problem by treating people needing different sized shoes as nodes, and creating directed edges when the left shoe size of any participant was equal to the right shoe size of any other participant. A cycle enumeration algorithm (in the original work, that of Liu and Wang [8]) was then applied to determine all of the cycles in the digraph. The results of the cycle enumeration were then used to determine whether all the nodes in the original graph were contained in cycles. If there were nodes which were not in cycles then Sanders applied a graph enlargement algorithm to add dummy nodes and the required additional edges to complete cycles and thus to ensure that all nodes in the original graph were in cycles in the "enlarged" graph. A cycle picking algorithm was then applied to produce an optimal selection of cycles where each node in the original graph was in at least one of the selected cycles.

SAICSIT '15, September 28-30, 2015, Stellenbosch, South Africa

 ISBN 978-1-4503-3683-3/15/09...$15.00

DOI: http://dx.doi.org/10.1145/2815782.2815825

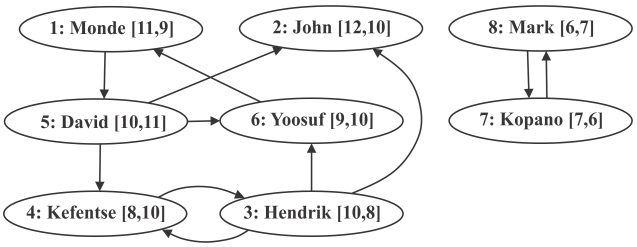


Figure 1: Sanders's original illustration of a shoe matching problem, pre-enlargement [10]

The graph enlargement algorithm of Sanders is naïve and does not always give a good solution. As an example, suppose we have a graph as illustrated in Figure 1. The problematic node in this case is (2: John) – John is not in a cycle and has no one to cooperate with in buying shoes. In the solution (see Figure 2), two dummy nodes have been added. A bridge node linked the two components and caused the component consisting of nodes (7: Kopano) and (8: Mark) to become part of the problem when it was initially a completed cycle. Figure 3 presents an alternative solution requiring only a single dummy node to satisfy everyone's footwear needs. In addition, the cost criterion presented by Sanders had problems – it spread the cost of dummy nodes across people in the same cycle as the dummy nodes which meant that people in cycles without dummy nodes benefited unduly.

The aim of this research was to find an improvement to the graph enlargement algorithm of Sanders [10]. The new algorithms presented in this paper could be useful in other circumstances where graphs must have their nodes contained within cycles.

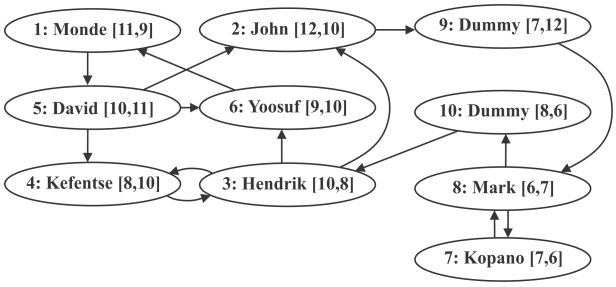


Figure 2: Sanders's original illustration of a shoe matching problem, post-enlargement [10]

Section 2 presents the problem in greater detail. The research methodology is provided in Section 3. Section 4 presents an improvement on Sanders's algorithm and Section 5 extends this approach by enlarging only a necessary subgraph of the original graph. Section 6 presents a novel way of enlarging graphs by transforming the adjacency matrix of the graph to a permutation matrix. Section 7 discusses the results of these new algorithms in greater detail. Sections 8 and 9 cover possible future work and the conclusions drawn respectively.

2. THE PROBLEM

The shoe matching problem attempts to satisfy the footwear needs of people with differently-sized feet in a cost-saving manner. The ideal solution would be that all the participants are able to cooperate with one another in such a way that each participant only needs to buy one pair of shoes (no dummy nodes required). In general, this is a very rare occurrence and dummy nodes are usually required. The goal is therefore to find a solution where each participant needs to pay for less than two pairs of shoes.

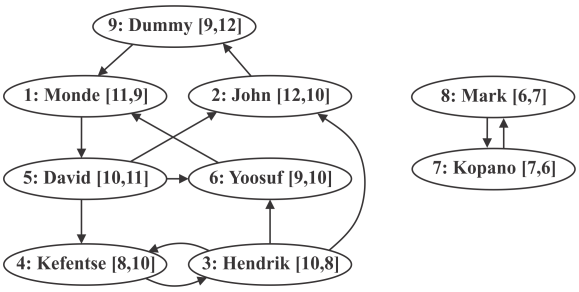


Figure 3: An alternative solution, post-enlargement, requiring only a single dummy node

The original graph enlargement algorithm for the shoe matching problem [10] was based on the observation that there are four cases which would result in a node not being in a cycle.

1. The node is *isolated* from the rest of the graph (it has neither incoming nor outgoing edges).
2. The node has only incoming edges.
3. The node has only outgoing edges.
4. The node is a "bridge" (it is on path between two other nodes in the graph which are either in cycles or a case 2 or case 3 node).

The approach applied first deals with cases 1 to 3 and then (if necessary) addresses bridge nodes. The first phase of the algorithm uses the existing graph to create a set of nodes that have no outgoing edges, a set of nodes that have no incoming edges, a set of isolated nodes and a set of all the other nodes in the graph. The approach is then to repeatedly link nodes in the sets together to form a new path. There are two cases here. In the first case, if there is still an isolated node then a path is formed by connecting a node from the set of nodes with no outgoing edges to an isolated node to a node with no incoming edges by means of two new "dummy nodes". These dummy nodes are required in order to deal with the abstraction of the real world situation that only allows an edge between two nodes if the left shoe size of the first node is equal to the right shoe size of the second node and if both nodes have shoe sizes that are different. In the second case, if there are no remaining isolated nodes then a node with no outgoing edges is connected to a node with no incoming edges using a single dummy node. Note that in both cases, if there is no remaining node with no outgoing edges or no remaining node with no incoming edges (or both) then such nodes are replaced by an arbitrarily chosen node from the original graph. There is no attempt made in the algorithm to determine which nodes are the "best" to use at any stage or even to try to determine the effect of choosing a given node. Essentially the selection is random.

Dealing with cases 1, 2 and 3 using the approach outlined

above will connect those nodes to the graph and may result in a cycle but may also lead to a case 4 situation. Or a case 4 could occur in the original graph. Dealing with a case 4 situation involves looking forward from the bridge node along the path containing the bridge node until a node which is not a bridge node is found, then similarly looking back along the path including the bridge node until a non-bridge node is found and then connecting those two nodes using a dummy node to complete a cycle. This process results in a node (or in some cases more than one node) which is not in a cycle in the original graph now being included in a cycle.

The approach described above results in every node in the original graph being in at least one cycle and each dummy node that is added also being in a cycle.

The focus of this paper is on trying to improve the original graph enlargement algorithm by making use of some knowledge of the structure of possible graphs and also considering more desirable outcomes.

Graph enlargement in this paper shall be defined as:
Instance: Given a digraph, $G(V, E)$ such that any node $v \in V$ has attributes (v_x, v_y) where $v_x, v_y \in \mathbb{Z}^+$, and for any two nodes $u, v \in V$ an edge uv is in E when $u_x = v_y$.
Problem: Find a set of nodes V' and associated edges E', such that $G_E(V_E, E_E)$ satisfies the property $P \leftarrow$ *all nodes in* V_E *are contained within cycles.* Here $V_E = (V \cup V')$ and $E_E = (E \cup E')$; any node, $w \in V_E$ has attributes (w_x, w_y) where $w_x, w_y \in \mathbb{Z}^+$; and for any two nodes $w, z \in V_E$ an edge w, z is in E_E when $w_x = z_y$).

The cost of the graph enlargement can be minimised by minimising the total number of nodes required (all nodes have equal cost/weight).

Sanders's cost distribution model places the cost burden of the dummy nodes on the participants *by cycle*. In other words, a cycle of 5 participants containing a single pair of dummy shoes will cost each participant of that cycle the value of $1 + \frac{1}{5}$ pairs of shoes. If there is a second cycle in the graph with 2 participants containing 1 pair of dummy shoes, then each participant will pay for $1 + \frac{1}{2}$ pairs of shoes. In this model a person who appears in more than one cycle is required to buy that many pairs of shoes. Thus someone who is in many cycles would have to buy many pairs of shoes.

In this paper the cost distribution model is redefined. The cost burden of all the dummy nodes is spread across all the participants (nodes) in the graph. If the addition of two dummy nodes are necessary to ensure all nodes are in cycles then two unique dummy nodes are added. However, these dummy nodes may be present in multiple cycles, and to ensure that these cycles are always satisfied, the dummy nodes are counted repeatedly across all cycles. If two dummy nodes are required to ensure the existence of three unique cycles, then in total six dummy nodes (pairs of shoes) are required. The issue of a few people being forced to buy more than one pair of shoes is also addressed in this work. This is discussed in more detail later.

3. METHOD

3.1 Overview

The focus of this research was to improve on the graph enlargement algorithm presented by Sanders. This meant being able to compare any new algorithms to the original so the overall approach was to:

1. Generate appropriate test data and represent this data as a digraph.
2. Enumerate all of the cycles in the graph.
3. Determine whether graph enlargement was required and apply a graph enlargement algorithm if necessary.
4. Pick the cycles in the graph to return a solution.
5. Evaluate the cost of enlarging the graph in terms of the number of nodes required.

These steps are expanded on in the subsections below. Note that the implementation in this study was done in Python and this had some impact on the decisions made in terms of the algorithms used.

3.2 Test data

Due to a lack of academic literature on the subject, shoe size data for graph construction was randomly generated (randomisation seeds could be set to ensure the generated pairs were kept constant across multiple runs). The following assumptions were made:

- Gender and style differences are not considered. A different graph could be constructed for each gender and style combination.
- Average shoe sizes range from 9 to 11, with a standard deviation of 2 (a range from 7 to 13).
- Once the left shoe size is generated the right shoe size is derived from it by adding a random value between -2 to 2 to the left shoe size.

3.3 Cycle enumeration

Once the tested data had been generated, the graph was represented as an adjacency list (in the case of the permutation matrix method, it was represented by an adjacency matrix) and Tarjan's cycle enumeration algorithm [12] was used to enumerate all the cycles within a graph. Tiernan's algorithm [13] was also evaluated but encountered regular recursion depth errors when the number of nodes in the graph exceeded 12 nodes.

3.4 Graph enlargement methods

The original algorithm [10] was implemented as were three other approaches as listed below and described in more detail in later sections of the paper. This paper presents the following methods of graph enlargement:

- An improvement on the original approach which minimises the unique number of dummy nodes required to enlarge the graph, in order to satisfy the connectivity property that all nodes must be contained within cycles (see Section 4).
- Creating a subgraph $H \subseteq G$. H consists of all the nodes in G that are not optimally contained in cycles. H is then augmented separately (see Section 5).
- Transforming the adjacency matrix to a permutation matrix (see Section 6).

Once the graph had been enlarged, it was necessary to find a minimal solution in terms of the number of cycles required to ensure that all nodes in the enlarged graph are contained in at least one of the selected cycles. Cycle picking is discussed below.

3.5 Cycle picking

Cycle picking is defined as "an optimisation problem where cycles are chosen from a directed graph under the constraint that any node that is in a cycle in the original directed graph must be in at least one of the chosen cycles" [10].

Several methods of cycle picking were considered in the original work: *Minimum number of cycles*; *All small cycles*; and *Minimum total cycle length.*

This work focused on the minimum number of cycles method of cycle picking. The minimum number of cycles method selects an optimal combination of cycles, such that every node which was contained within a cycle before is still contained within at least one cycle, but whilst minimising the total number of cycles necessary. A cycle enumeration algorithm will enumerate all the cycles and r-combinations of cycles will be evaluated. This allows the maximum number of potential matches possible, but it can be a time consuming process, depending on the number of nodes (and is possibly NP-complete [10]). When all the possibilities have been generated, the algorithm selects the minimum number of cycles with the restriction that any node contained within a cycle(s) in the original graph, must be contained in at least one cycle in the new graph.

3.6 Evaluation

To evaluate the efficiency of the new algorithms, 60 shoe matching problem scenarios were simulated across Sanders's algorithm and the three newly developed algorithms. The algorithms were assigned a percentage score, which represents the following ratio: the number of times the algorithm achieved the lowest number of total nodes required to solve the shoe matching problem (compared to the other algorithms), divided by the total number of quantifiable results produced by the specific algorithm. It is possible for more than one algorithm to produce the lowest number of total nodes, the result then counts as a success for all the algorithms with the most efficient result.

4. COST-OPTIMISED ENLARGEMENT

This approach attempts to minimise the unique number of dummy nodes required to augment the graph. Initially, it was developed to test whether minimising the unique number of dummy nodes would produce a more cost-efficient solution, in terms of total node-cost. It was found that minimising the unique number of dummy nodes caused these dummy nodes to occur repeatedly across multiple cycles, i.e. fewer dummy nodes formed part of an increased number of cycles to solve the shoe matching problem. In smaller graphs this is generally not a problem and the solutions produced are still an improvement on those produced by Sanders's original algorithm.

The cost-optimised algorithm follows the same enlargement logic as Sanders's original algorithm [10], except for a few changes, listed below.

The following methods generally reduce the unique number of dummy nodes required within the enlarged graph:

- Isolated nodes would not be joined to larger structures unless it would be more optimal to do so. It is generally more cost efficient to simply furnish the isolated node with a dummy node and form a cycle of two nodes.
- Bridge nodes were avoided where possible. To avoid bridge nodes, graphs were enlarged by component and components were not linked.
- Some basic rules for enlargement and rule-based decision making were introduced to ensure that optimal solutions were generated for the current situation of the graph.
- Compressing cycles to compress repeated sequences of nodes within cycles.

This algorithm plays a pivotal role in the *subgraph enlargement algorithm* described in Section 5 below.

5. SUBGRAPH ENLARGEMENT METHOD

Suppose G is a graph with cycles $C = \{c_1, c_2, ..., c_n\}$, then it is possible to find $C' \subseteq C$ such that C' is maximal and any node in a cycle in C' appears exactly once. Thus, existing cycles without duplicates can be isolated and set aside as *completed* and the remaining nodes will be used to construct a subgraph $H \subseteq G$.

The subgraph enlargement method makes use of the cost-optimised enlargement algorithm, which minimises the number of unique dummy nodes required (see Section 4). The cost-optimised algorithm is used to enlarge H and ensure that all nodes within H are contained in cycles.

To ensure the selection of a subgraph $H \subseteq G$ is optimal and due to the NP-completeness of this procedure, we will again use a combinatorial approach. The optimal solution is defined as **setting aside** a combination of cycles containing each node at most once, whilst also trying to maximise the number of nodes contained within these cycles. This selection is then marked as *completed* and the remaining nodes are used to construct the subgraph H.

Suppose we have a graph G as illustrated below in Figure 4.

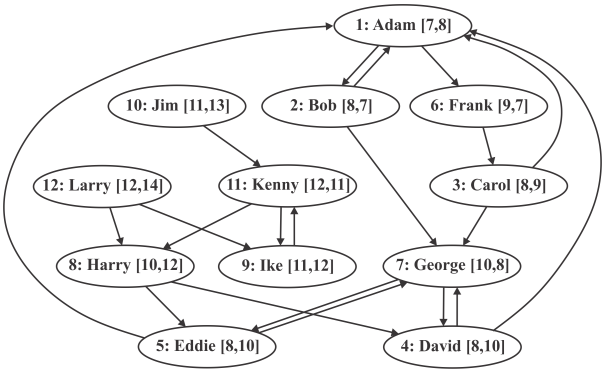


Figure 4: The graph G to which the subgraph enlargement algorithm will be applied

The complete list of cycles present in the graph are as follows:

- (1: Adam) → (2: Bob)
- (1: Adam) → (2: Bob) → (7: George) → (4: David)
- (1: Adam) → (2: Bob) → (7: George) → (5: Eddie)
- (1: Adam) → (6: Frank) → (3: Carol)
- (1: Adam) → (6: Frank) → (3: Carol) → (7: George) → (4: David)
- (1: Adam) → (6: Frank) → (3: Carol) → (7: George) → (5: Eddie)
- (4: David) → (7: George)

- (5: Eddie) → (7: George)
- (9: Ike) → (11: Kenny)

The combination of cycles with maximal length (and with each node appearing at most once in the combination) is:

- (1: Adam) → (6: Frank) → (3: Carol)
- (5: Eddie) → (7: George)
- (9: Ike) → (11: Kenny)

The nodes thus not contained within cycles, and also the nodes that $H \subseteq G$ will be comprised of (see Figure 5), are:

- (2: Bob)
- (4: David)
- (8: Harry)
- (10: Jim)
- (12: Larry)

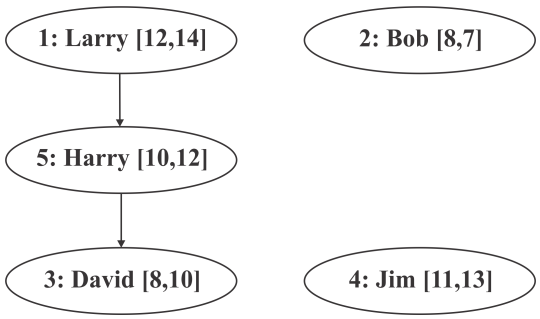


Figure 5: The graph $H \subseteq G$

Once the cost-optimised algorithm (i.e., minimising unique number of dummy nodes) is applied to this graph, H will be enlarged such that all nodes are contained within cycles. The solution is illustrated by Figure 6. Note that the nodes already contained within cycles before enlargement are not illustrated.

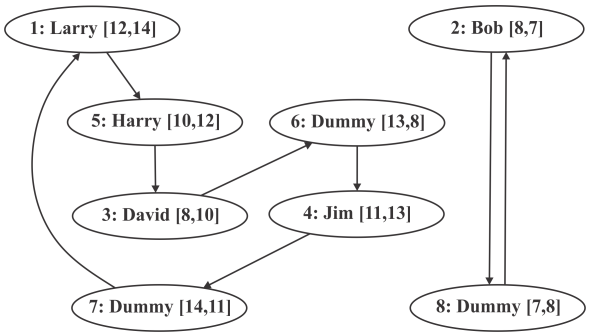


Figure 6: The graph H, once enlargement using the subgraph algorithm is complete (please note that the nodes which were contained within cycles pre-enlargement are not illustrated here)

6. PERMUTATION MATRIX METHOD

Any directed graph, G, can be represented by an adjacency matrix, say A. Adjacency matrices are square matrices. For any element in the matrix, say $A[i, j]$, if the value of the element is 1, it indicates an arc from node $i \to j$. A lack of adjacency between nodes is indicated by a zero [5].

Permutations and permutation matrices are important concepts in group theory. A permutation can be defined as follows [9]: **"If X is a nonempty set, a permutation of X is a bijection α: $X \to X$."**

A bijection is a function which is both one-to-one and onto. In essence, a permutation is a rearrangement of elements. Suppose we have permutations:

$$\alpha = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

$$\beta = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

The permutation $\alpha(\beta(1)) = \alpha(2) = 2$ is an example of such a rearrangement of elements.

Any permutation can be written as a union of disjoint cycles [6], for example:

$$\alpha = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 3 & 8 & 6 & 7 & 4 & 1 & 5 & 2 \end{pmatrix} = (1, 3, 6)(2, 8)(4, 7, 5)$$

A permutation of a finite set is a product of disjoint cycles as proven by [6, p. 89]:

Let $B_1, B_2, ..., B_r$ be the orbits of α and let μ_i be the cycle defined by

$$\mu_i = \begin{cases} \alpha(x) & \text{for } x \in B_i \\ x & \text{otherwise.} \end{cases}$$

It follows that $\alpha = \mu_1\mu_2...\mu_r$. Since the equivalence-class orbits $B_1, B_2, ..., B_r$, being distinct equivalence classes, are disjoint, the cycles $\mu_1\mu_2...\mu_r$ are disjoint also. □

The fact that a permutation consists solely of disjoint cycles is very important. It means that every element of the permutation (and its corresponding permutation matrix) will always be contained within a cycle, and there will be no repeats of elements in the same cycle or different cycles. This provides us with the following benefits:

- There will be no need for cycle enumeration.
- There will be no need for cycle picking.
- There will be no need to check for bridge nodes.
- There will be no need to check for repeated nodes. The number of nodes contained within the graph at first glance is the number of nodes necessary for every node to be contained within a unique cycle.

The above processes, especially cycle enumeration and cycle picking, hamper performance and cause Sanders's original algorithm to become a nonviable solution on larger graphs (25+ nodes). The matrix enlargement method can solve the shoe matching problem for graphs of 1000+ nodes within seconds.

Permutation matrices are square matrices filled with ones and zeroes (row-equivalent to the identity matrix), with the restriction that the ones may only occur once in every row and column. Matrix A is a valid permutation matrix:

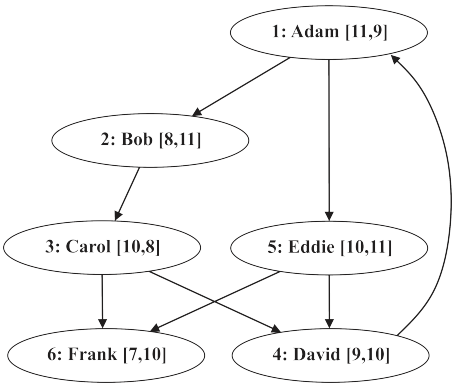


Figure 7: A permutation matrix will be constructed from the adjacency matrix of this graph

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Suppose we have a graph G as illustrated in Figure 7, with adjacency matrix:

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 1 & 2 & \end{pmatrix} \end{array}$$

T_r represents the sum of the ones in each row, while T_c represents the sum of the ones in each column. In a permutation matrix, each row and each column must add up to 1, only. That is to say, each row and each column may only contain a single entry of 1, the rest of the matrix must be populated with zeroes.

In this matrix, three rows have $T_r = 2$ (thus 1 redundant outgoing edges for each of the nodes v_1, v_3, v_5) and two columns have $T_c = 2$ (thus 1 redundant incoming edge for nodes v_4, v_6). Starting with T_r, work from bottom-to-top and right-to-left (the order is unimportant; this is the author's preference), and simply remove the redundant entries of 1 and replace them with 0. Every value in T_r should now be either 0 or 1. Refresh the count for T_c to check if any nodes have redundant incoming edges.

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 1 & 0 & \end{pmatrix} \end{array}$$

In this case, the removal of all redundant outgoing edges has not solved the problem of redundant incoming edges. If any value for $T_c > 1$, again work from bottom-to-top and right-to-left, removing the redundant ones in each column.

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & \end{pmatrix} \end{array}$$

However, now note the number of zeroes in T_r and T_c (these two values should be equal), say n, and enlarge the graph by n dummy nodes. In this case $n = 2$ and we are left with the following adjacency matrix:

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \end{pmatrix} \end{array}$$

It is essential to keep track of the dummy nodes (v_7, v_8) and the original zero-row nodes $(v_5, v_6$ - from top to bottom) and zero-column nodes $(v_6, v_5$ - from right to left). These nodes should be places on three separate stacks, namely the dummy stack, zero-row node stack, and zero-column node stack.

The graph is now ready to be enlarged. Pop the first zero-row node (v_5) off its stack, the first dummy node (v_7), and the first zero-column node (v_6). Connect $v_5 \rightarrow v_7 \rightarrow v_6$ only. Do not connect the last node in this sequence to the first. The matrix is thus altered as follows:

- $A[v_5, v_7] = 1$
- $A[v_7, v_6] = 1$

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & \end{pmatrix} \end{array}$$

The next selection of nodes to be popped is $v_6 \rightarrow v_8 \rightarrow v_5$ only.

- $A[v_6, v_8] = 1$
- $A[v_8, v_5] = 1$

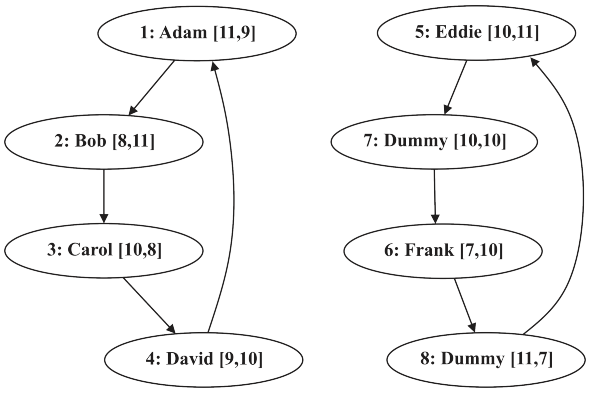


Figure 8: The final graph produced by the permutation matrix

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ T_c \end{array} \begin{array}{c} \begin{array}{ccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & T_r \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \end{pmatrix} \end{array}$$

Since both T_r and T_c are filled with ones (exclusively), the enlargement is complete (see Figure 8). The adjacency matrix has now been transformed into a permutation matrix and inherits the properties of a permutation; once again, of special interest to the focus of this article are the following:

- The permutation matrix is a union of disjoint cycles: every node is contained within a cycle (see [6, p. 89])
- No node is ever repeated: every node appears exactly once, in exactly one cycle

In Figure 8, (7: Dummy [10,10]) is a redundant dummy node. It is possible to remove this node and still have a complete cycle and viable solution. However, the addition of 2 dummy nodes is a failsafe to construct a permutation matrix. **Also note that cycle picking is redundant for this enlargement method. When the algorithm destroys the redundant edges, it is in essence performing a procedure very much like cycle picking on itself.** For example, technically an edge should be present (one of many) between (3: Carol [10,8]) and (6: Frank [7,10]). The disjoint cycles property of permutation matrices produces cycles which resemble "picked" cycles.

7. RESULTS AND DISCUSSION

Both the matrix and subgraph enlargement methods have favourable results overall. The results are displayed in Tables 1, 2, and 3. Tables 1 and 2 present the individual test scenarios and their results for each algorithm, while Table 3 presents the success rates of the algorithms across the results of Tables 1 and 2.

Each scenario is represented by a combination of a seed and graph size (number of nodes). For each seed, as the number of nodes increases, the graph simply expands on the previous graph. In other words, for any specific seed, the graph generated of size 8, for example, will be a subgraph of the graph generated of size 10.

The algorithm(s) achieving the fewest number of total nodes for each scenario has its result highlighted in ***bolded italics***. In the case of an OutOfMemory exception, the result is left blank and does not count towards the algorithm's success rate. Success rates are expressed as percentages and represent the following ratio: the number of times the algorithm has solved the shoe matching problem with the least number of nodes (compared to its competitors), divided by the total number of quantifiable results the algorithm produced. The resulting percentage is then rounded to the nearest integer.

For the sake of completeness, the results of the cost-optimised algorithm (i.e., minimising unique number of dummy nodes) is also included, since it is the enlargement method used by the subgraph enlargement algorithm.

The cost-optimised algorithm was successful in minimising the unique number of dummy nodes necessary, but this led to an increase in repeated nodes across multiple cycles. It is apparent that fewer unique dummy nodes do not always imply that the total number of nodes required to satisfy the shoe matching problem will decrease. It is, however, by far the most successful algorithm in achieving a graph of the required form whilst minimising the number of unique dummy nodes.

The permutation matrix algorithm greatly decreased the running times of the algorithm, without incurring a major decrease in cost-efficiency (node-cost). The permutation matrix algorithm is capable of handling large graphs (1000+ nodes) and has no need of cycle enumeration, cycle picking, or cycle compression algorithms. By discarding these CPU intensive (and some NP-complete) operations, the algorithm is able to solve the shoe matching problem on graphs with an order of 1000 nodes within a matter of seconds on a modern desktop computer.

The subgraph algorithm has shown promising results overall. In some cases it is less efficient than Sanders's algorithm, but overall it has the highest success rate in terms of minimising the total node cost across all the algorithms and is the ideal solution for the shoe matching problem involving graphs with order less than 30 nodes.

8. FUTURE WORK

The algorithms covered in this paper do meet the aim of enlarging the existing digraph at generally reduced cost (in terms of number of necessary nodes). This section presents a few ideas for future work in the area.

8.1 Involving Minimum Spanning Trees

A possible future solution could involve the construction of a minimum spanning tree (MST). It would then be possible to split the newly constructed MST at any branching points. Whenever a node, say n, branches into multiple nodes, break the branches to construct separate components and replace node n with dummy nodes as required. For example, if node n branched into 3 new nodes, 2 of the edges (branches) need to be broken and n should be replaced with 2 dummy nodes for the new components this process creates. A cycle can then be created from each of the separated components.

Seed	Nodes	Sanders's Algorithm (Total Nodes)	Cost-optimised Algorithm (Total Nodes)	Permutation-matrix Algorithm (Total Nodes)	Subgraph Algorithm (Total Nodes)
586 015 916	8	17	***14***	***14***	***14***
	10	***16***	***16***	***16***	***16***
	12	***19***	***19***	20	20
	14	***22***	26	23	***22***
	16	***26***		***26***	27
	18	***31***		***31***	***31***
273 100 389	8	22	15	***14***	15
	10	26	***17***	18	***17***
	12	23	23	21	***20***
	14	23	***22***	23	***22***
	16	***26***	***26***	27	***26***
	18	31	***30***	31	***30***
513 577 387	8	25	***14***	***14***	***14***
	10	27	18	***16***	18
	12	35	24	***20***	24
	14	38	28	***23***	28
	16	37	30	***26***	30
	18	45	35	***30***	38
912 644 420	8	***10***	***10***	13	***10***
	10	17	***15***	17	***15***
	12	25	***20***	***20***	***20***
	14	25	***21***	22	25
	16			***25***	
	18			***29***	
422 223 344	8	12	13	***10***	13
	10	18	***15***	***15***	***15***
	12	25	20	***19***	20
	14	29	25	***23***	29
	16	30		25	***22***
	18			***29***	37

Table 1: Comparative results across all algorithms (1/2)

8.2 Calculated Selection of Intermediate Nodes

Currently whenever the cost-optimised or Sanders's algorithm requires "some node" as an intermediate step in enlarging the graph, a node is arbitrarily selected. It is possible that vertices with lower degrees may be a better choice, in order to reduce repeated nodes across multiple cycles. It may also argued be that nodes with a higher degree are a better choice since this allows more cycle combination choices for the cycle picking processes. Future work could explore this idea.

8.3 Improvements to the Subgraph Algorithm

The subgraph algorithm currently makes use of the cost-optimised algorithm to enlarge the subgraph $H \subseteq G$. It may be worth exploring whether using the permuatation matrix algorithm (or some other enlargement algorithm) instead will improve the node cost- and speed-efficiency of the subgraph algorithm.

8.4 Divide and Conquer with Parallel Computing

Investigating whether subroutines such as cycle enumeration and cycle picking could be parallelised, could potentially lead to great performance improvements. In the case of component-wise enlargement, each component can be enlarged in parallel. Parallelising the subroutines can also enable the algorithms to be run on a computing cluster.

9. CONCLUSION

This paper has presented several improvements to Sanders's original graph enlargement algorithm. A fairer cost distribution model has been presented, which does not penalise participants of smaller cycles. The burden of cost is placed equally across all the participants.

The subgraph algorithm provides better results overall than Sanders's original algorithm, by only enlarging a subgraph of nodes from the original graph. The subgraph enlargement algorithm is indeed more cost-efficient (in terms of node cost) than Sanders's original algorithm.

A novel approach of graph enlargement, by transforming the adjacency matrix representing the digraph to a permutation matrix, was also presented. This new approach does away with NP-complete procedures, and does not require auxiliary functions, such as cycle enumeration or cycle picking. As such, this method of enlargement is much faster than Sanders's original enlargement algorithm, whilst often

Seed	Nodes	Sanders's Algorithm (Total Nodes)	Cost-optimised Algorithm (Total Nodes)	Permuation-matrix Algorithm (Total Nodes)	Subgraph Algorithm (Total Nodes)
475 887 106	8	27	12	***11***	***11***
	10	31	15	15	***14***
	12	33	29	***19***	22
	14		30	23	***22***
	16			***25***	
	18			***28***	
664 469 990	8	14	13	12	***11***
	10	14	14	14	***13***
	12	17	17	17	***16***
	14	23	22	21	***19***
	16			***25***	26
	18			***28***	39
965 309 708	8	14	15	***12***	***12***
	10	23	21	***15***	16
	12	27	25	***19***	20
	14	27	24	23	***22***
	16	***21***	23	26	***21***
	18	***30***		***30***	***30***
456 777 039	8	18	13	***12***	13
	10	***13***	***13***	14	***13***
	12	22	***18***	***18***	***18***
	14	***19***	***19***	22	20
	16	22	***21***	26	***21***
	18	***27***	***27***	29	
222 892 396	8	12	12	***10***	***10***
	10	19	16	15	***14***
	12	28	24	***18***	23
	14	34	24	***22***	23
	16	36	***26***	27	30
	18			***29***	

Table 2: Comparative results across all algorithms (2/2)

being more cost-efficient as well.

10. REFERENCES

[1] G. Chartrand, L. Lesniak, and P. Zhang. *Graphs and Digraphs*. Chapman and Hall/CRC Press, Boca Raton, Florida, 5th edition, 2011.

[2] G. Egeland and P. E. Engelstad. The economy of redundancy in wireless multi-hop networks. In *Proceedings of the 2009 IEEE Conference on Wireless Communications & Networking Conference*, WCNC'09, pages 3023–3028, Piscataway, NJ, USA, 2009. IEEE Press.

[3] K. Eswaran. *Representation of Graphs and Minimally Augmented Eulerian Graphs with Applications in Data Base Management*. Research reports // IBM. IBM Thomas J. Watson Research Division, 1973.

[4] K. P. Eswaran and R. E. Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–665, 1976.

[5] S. Even. *Graph Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2011.

[6] J. B. Fraleigh. *A First Course In Abstract Algebra*. Pearson Education Inc., 7th edition, 2003.

[7] H. Frank and W. Chou. Connectivity considerations in the design of survivable networks. *Circuit Theory, IEEE Transactions on*, 17(4):486–490, Nov 1970.

[8] H. Liu and J. Wang. A New Way to Enumerate Cycles in Graph. In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*, AICT-ICIW '06, pages 57–59, Washington, DC, USA, 2006. IEEE Computer Society.

[9] J. J. Rotman. *Graduate Texts in Mathematics: An Introduction to the Theory of Groups*. Springer, 4th edition, 1994.

[10] I. Sanders. Cooperating to buy shoes: An application of picking cycles in directed graphs. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists (Theme: "A Connected Society")*, pages 8–16, East London, 2013.

[11] T. Sheng Hsu and V. Ramachandran. Finding a smallest augmentation to biconnect a graph. *SIAM J. Comput.*, 22(5):889–912, 1993.

[12] R. E. Tarjan. Enumeration of the Elementary Circuits

	Sanders's Algorithm (Total Nodes)	Cost-optimised Algorithm (Total Nodes)	Permuation-matrix Algorithm (Total Nodes)	Subgraph Algorithm (Total Nodes)
Total Number of Quantifiable Scenarios	51	48	60	34
Scenarios with Least Number of Total Nodes	12	19	35	31
Number of OutOfMemory Scenarios	9	12	0	6
Success Rate % (Minimising Total Nodes)	24 %	40 %	58 %	57 %

Table 3: Success rates across all algorithms

of a Directed Graph. Technical report, Ithaca, NY, USA, 1972.

[13] J. C. Tiernan. An Efficient Search Algorithm to Find the Elementary Circuits of a Graph. *Communications of the ACM*, 13(12):722–726, Dec. 1970.

[14] R. Xulvi-Brunet and I. M. Sokolov. Growing networks under geographical constraints. *Phys. Rev. E*, 75:046117, Apr 2007.