SOLVING THE STUDENT PROJECT ALLOCATION PROBLEM

Author: Sam Lightfoot
Supervisor: Ian Pratt-Hartmann

May 3, 2016
Contents

1 Introduction  2

2 Background  4
  2.1 Graphs ................................................................. 4
  2.2 Flow Networks .................................................... 5
    2.2.1 Cuts .............................................................. 6
    2.2.2 Residual Capacity and Augmenting Paths ................. 7
    2.2.3 Ford-Fulkerson Algorithm ................................. 8
    2.2.4 Shortest Paths ................................................ 10
    2.2.5 Bellman-Ford Algorithm ................................. 10
    2.2.6 Busacker-Gowen Algorithm ......................... 11

3 Problem Definition  12
  3.1 Definition of SPA-P .............................................. 12
  3.2 Modelling the SPA-P ........................................... 12
  3.3 The Algorithm .................................................. 13

4 Design  14
  4.1 Back-end Application ............................................ 14
    4.1.1 Back-end Application Requirements ..................... 14
    4.1.2 Application Architecture .................................. 15
    4.1.3 Technology Selection ..................................... 15
  4.2 Designing the Web Application ............................... 15
    4.2.1 Web Interface Requirements .............................. 15
    4.2.2 Application Architecture .................................. 16
    4.2.3 Technology Selection ..................................... 17
    4.2.4 Application Interface Design ........................... 17

5 Implementation  18
  5.1 Back-end Application ............................................ 18
    5.1.1 Reading the Data ........................................... 18
    5.1.2 Representing the Flow Network ......................... 19
    5.1.3 Building the Flow Network .................................. 19
    5.1.4 Optimizing the Flow ...................................... 20
    5.1.5 Outputting the Results ................................... 20
  5.2 Front-End Application ......................................... 20
    5.2.1 Template Design ........................................... 21
    5.2.2 Adding Functionality ..................................... 21
6 Evaluation
   6.1 Student Randomization ................................................. 22
   6.2 Quality Assurance ..................................................... 22
   6.3 Performance Testing ................................................... 22

7 Conclusion
   7.1 Achievements ............................................................ 25
   7.2 Challenges ................................................................ 25
   7.3 Future Work ................................................................ 25

Bibliography .......................................................................... 28

A Test Results ........................................................................ 29

B Busacker-Gowen Snippet ..................................................... 30
List of Figures

2.1 A directed graph. ...................................................... 4
2.2 A Bipartite graph containing two disjoint sets ................. 5
4.1 3-Tier Design Architecture .................................... 16
5.1 The flow network modified to fit the SPA problem .......... 20
5.2 A screenshot of the web interface ............................... 21
6.1 A performance graph to show how the runtime of the matching algorithm varies with the instance size. Note, each student and lecturer made 3 project choices. 23
6.2 A graph to show how the total ranking of the students’ matches varies with the instance size. Note, each student and lecturer made 3 project choices. 23
List of Tables

4.1 Prioritisation of Requirements for the Back-end Application .............. 14
4.2 Prioritisation of Requirements for the Web Application ................. 15
A.1 Performance data ........................................................................ 29
A.2 Total ranking data ....................................................................... 29
Abstract

With the increasing demand for centralized matching systems within universities, there exists a need for an all-in-one software package. One common use of a matching system is to automate the assignment of students to projects. This project details the development of a system, to be used by universities, that finds the optimal student-project matching. The scenario covered involves students creating a preference list of projects, which have associated capacity constraints, as well as lecturers selecting the projects they wish to supervise, who also have capacity constraints. We discuss the background theory used to implement the matching algorithm, explore how we model the problem and present results of how the program runs in practice.
Acknowledgements

I would like to thank my supervisor, Dr. Ian Pratt-Hartmann, for his continued guidance and encouragement over the past year. Further thanks go to my family and friends for helping me get to where I am today.
Chapter 1

Introduction

Many universities require students to complete a project as part of their undergraduate degree. A common scenario involves students selecting projects of interest from a list provided by the university faculty. Each project in the given list has been proposed by a university lecturer, usually with expertise in that field of study. There is a limit on the number of students that can be assigned to a particular project, and the number of students a lecturer is prepared to supervise. Lecturers can choose to supervise projects proposed by others.

A problem exists in finding the optimal match between students, projects and supervisors. A match must be found that assigns the largest number of students to a project of their choice, whilst also finding a lecturer willing to supervise each project. This may be trivial for a small number of students, but as the datasets in matching problems grow in size, so does the inherent problem complexity. Finding optimal solutions is near impossible without the use of an automated solving process.

In order to automate the solving process mathematical algorithms are required. The mathematical aspects of matching problems have been studied extensively since the 1960’s, when D. Gale and L. Shapley submitted a paper titled “College Admissions and the Stability of Marriage” [5]. In their paper they introduced the stable marriage problem and proposed an iterative algorithm that formed the basis for solving the similar college admissions problem (detailed in the same paper). The college admissions problem involves finding the optimal match between students and colleges, where the students rank colleges in order of preference. Since then, the stable roommates problem, a derivative of the stable marriage problem, was proposed and solved in 1984. More recently in 2003, a solution to the student-project allocation problem was proposed by D.J. Abraham, R.W. Irving, and D.F Manlove [2].

Nowadays, there is an increasing interest in using centralised matching schemes that employ efficient algorithms for the student-project allocation problem. Such automated systems are currently in use at the University of York [8] and the University of Southampton [3].

This report focuses on the problem of assigning students to projects subject to student preference lists and capacity constraints as described above. Within this report we refer to the problem as the Student-Project Allocation problem (SPA). The aim of this report is to present a solution to the SPA problem and detail the architecture of the system from which it is employed.

The exact details of the student-project assignment problem, which is formally defined in Chapter 3, are as follows. Each student forms a project preference list of a defined length (specified by the university). Lecturers have the ability to propose projects, but this is not a requirement. Lecturers can choose projects they wish to supervise (there is no limit on how many they choose). Lecturers have a capacity, which defines how many students they are
willing to supervise in total. Each project has a capacity, defining how many people can be assigned to that project. We wish to find a matching in which the maximum number of students are assigned to a project of their choice, such that no improved optimal matching exists. An improved matching would be defined as assigning a larger number of students or assigning the same number of students, but assigning them to a higher ranked project than before.

The structure of the report is as follows. Chapter 2 explores the background theory behind the implemented matching algorithm. Chapter 3 presents a formal definition of the SPA problem and shows how it can be modelled. Chapter 4 discusses the design choices made to create the application. Chapter 5 outlines how the software was implemented and how the overall system works together. Chapter 6 provides an evaluation of the project and analyses test results. Finally, Chapter 7 summarizes the achievements and challenges of the project, along with a brief discussion on the future of the project.
Chapter 2

Background

This chapter details the background of the project, discussing concepts used to derive an algorithm for solving the SPA problem.

2.1 Graphs

Graphs provide us with a structure to represent relationships between pairs of objects. Graphs have applications in a host of different domains, including transportation networks (rail links), electrical engineering (circuits) and computer networking (network mappings). Algorithms have been developed that operate on graphs to assist with real life problems such as delivery routes for truck drivers. We start by detailing the notation used when dealing with graphs.

A graph $G$ is a pair $G = (V, E)$ consisting of a finite set $V \neq \emptyset$ and a set $E$ of two-element subsets of $V$. The elements of $V$ are called vertices. An element $e = (u, v)$ of $E$ is called an edge with end vertices $u$ and $v$. We say that $u$ and $v$ are adjacent or neighbours of each other. A vertex can also been called a node.

Edges in a graph are either directed or undirected. An edge $(u, v)$ is said to be directed from $u$ to $v$ if the pair $(u, v)$ is ordered, with $u$ preceding $v$. An edge $(u, v)$ is said to be undirected if the pair $(u, v)$ is not ordered. If all the edges in a graph are undirected, then the graph is called an undirected graph. Intuitively, a graph containing all directed edges is called a directed graph (or digraph). A graph with both directed and undirected edges is called a mixed graph.

An example of a directed graph is shown in Figure 2.1.

![Figure 2.1: A directed graph.](image)

Let $G = (V, E)$ be a graph and $V'$ a subset of $V$. Let $H = (V', E|V')$ be a graph where $E|V'$ represents all edges $e \in E$ which have both their vertices in $V'$. The graph $H$ is called the
induced subgraph on $V'$ which we denote $G|V'$. Each graph containing a subset of vertices and edges from another graph $J$ is said to be a subgraph of $J$.

A path in a graph represents a way to get from an origin vertex to a destination vertex by traversing edges in the graph. Formally, a path $p = (v_0, ..., v_k)$ of length $k$ connects vertices $v_0$ and $v_k$ if subsequent nodes in $p$ are connected by edges in $E$. Two vertices $u$ and $v$ of a graph $G$ are connected if there exists a path between them. If all pairs of vertices of $G$ are connected, $G$ itself is called connected. Cycles in a graph are paths that share the first and last node.

A bipartite graph is a graph in which the vertices can be split into two disjoint subsets $U$ and $V$, such that every edge connects a vertex in $U$ to one in $V$. They are useful for modelling relations between two different classes of objects. In a bipartite graph $G$, partitioned into two subsets, students $U$, and projects $V$, a connecting edge $e = (u, v)$, where $u \in U$ and $v \in V$ would indicate that that student $u$ has chosen project $v$. An example of a bipartite graph can be seen in Figure 2.2.

![Figure 2.2: A Bipartite graph containing two disjoint sets](image)

### 2.2 Flow Networks

A flow network $N$ is a tuple $\langle V, E, c, s, t \rangle$, where $\langle V, E \rangle$ is a directed graph; $f : E \rightarrow \mathbb{R}^+_0$, $c : E \rightarrow \mathbb{R}^+_0$ and $s, t \in V$. We call the value $c(e)$, for $e \in E$, the flow capacity of the edge; we call $s$ the source and $t$ the sink. A flow on $N$ is a mapping $f : E \rightarrow \mathbb{R}^+_0$ satisfying two following conditions:

- For each edge $e$,
  \[0 \leq f(e) \leq c(e) \quad (capacity \, rule).\]

- For each vertex $v$, excluding source $s$ and sink $t$, where $e^- \text{ and } e^+$ denote the start and end vertex of $e$, respectively.
  \[\sum_{e^- = v} f(e) = \sum_{e^+ = v} f(e) \quad (conservation \, rule)\]

Therefore, the capacity rule requires that the edge carries a nonnegative amount of flow which may not exceed the capacity of the edge. The conservation rule means that flows are
preserved: at each vertex, excluding the source and the sink, the amount that flows in also flows out. It is clear that the total flow leaving \( s \) should equal the total flow going into \( t \).

The value of a flow \( f \), denoted by \(|f|\), is equal to the total amount of flow leaving source \( s \). A flow \( f \) is said to be maximal if \(|f| \geq |f'|\) holds for every flow \( f' \) on \( N \). This leads us to one of the main problems in flow network theory; determining whether or not a flow is maximal in a given network.

### 2.2.1 Cuts

Let \( G = (V, E) \) be a graph and \( N = (G, c, s, t) \) be a flow network. A cut of \( N \) is a partition \( V \) of the vertices into two disjoint subsets \( S \) and \( T \), with \( s \in S \) and \( t \in T \), where \( V = S \cup T \). An edge \( e \) of \( N \) with origin \( u \in S \) and destination \( v \in T \) is said to be a forward edge of cut \((S, T)\). An edge \( e \) in the same cut with origin \( u \in T \) and destination \( v \in S \) is said to be a backward edge.

Let \( f \) be a flow for \( N \). The flow across cut \((S, T)\) on \( N \), is equal to the sum of the flows in the forward edges minus the sum of the flows in the backward edges of \( X \). The capacity of a cut \((S, T)\) is the sum of the capacities of the forward edges (i.e. those that are oriented from \( S \) to \( T \)). The capacity of a cut \((S, T)\) is defined as

\[
c(S, T) = \sum_{e^+ \in S, e^- \in T} c(e).
\]

A cut is said to be minimal if \( c(S, T) \leq c(S', T') \) holds true for every cut \((S', T')\). The following lemma illustrates that the capacity of a minimal cut provides the desired upper bound on the value of a flow (i.e. the maximum flow).

**Lemma 2.2.1.** Let \( N = (G, c, s, t) \) be a flow network, \( f \) a flow, and \((S, T)\) a cut then

\[
|f| = \sum_{e^- \in S, e^+ \in T} f(e) - \sum_{e^- \in S, e^+ \in T} f(e)
\]

Note, the flow cannot exceed the capacity of the cut. \(|f| = c(S, T)\) if and only if each edge \( e \) with \( e^- \in S \) and \( e^+ \in T \) is saturated, whereas each edge \( e \) with \( e^- \in T \) and \( e^+ \in S \) has no flow (\( f(e) = 0 \)).

**Proof.** Summing the flow conservation equation over all \( v \in S \) gives

\[
|f| = \sum_{v \in S} \left( \sum_{e^- = v} f(e) - \sum_{e^+ = v} f(e) \right)
= \sum_{e^- \in S, e^+ \in S} f(e) - \sum_{e^- \in S, e^+ \in T} f(e) + \sum_{e^- \in S, e^+ \in T} f(e) - \sum_{e^+ \in S, e^- \in T} f(e).
\]

The sum of the first two terms is 0. Observe \( f(e) \leq c(e) \) for all edges \( e \) with \( e^- \in S \) and \( e^+ \in T \) (forward edges in the cut), and \( f(e) \geq 0 \) for all edges \( e \) with \( e^+ \in S \) and \( e^- \in T \) (backward edges in the cut). This shows the desired inequality along with the assertion on the equality case.

\[\square\]

To summarize, for any given cut in a flow network \( N \), the capacity of the cut is an upper bound on any flow for \( N \).
2.2.2 Residual Capacity and Augmenting Paths

Suppose we have a maximum flow $f$, we need to prove no additional flow can be pushed along $f$. Using the concepts of residual capacity and augmenting paths we are able to prove that flow $f$ is maximal.

Consider $N$ to be a flow network in graph $G$ with capacity function $c$, source $s$, and sink $t$. Let $f$ be a flow on $N$. Given an edge $e = (u, v)$ of $G$, the residual capacity of $(u, v)$ on flow $f$, defined as

$$\Delta_f(u, v) = c(e) - f(e),$$

and the residual capacity from $v$ to $u$ is defined as

$$\Delta_f(v, u) = f(e).$$

In other words, the residual capacity on a flow $f$ is the amount of additional capacity that can be pushed from $s$ to $t$ along $f$. The residual capacity $\Delta_f$ of a path $\pi$ is the minimum residual capacity of all edges in $\pi$. This minimum value is the amount of additional flow we can push along the path $\pi$ before reaching the capacity limit of an edge in $\pi$.

A path $\pi$ from $s$ to $t$ is called an augmenting path on flow $f$ if the following hold

- $f(e) < c(e)$ if $e$ is a forward edge
- $f(e) > 0$ if $e$ is a backward edge

The following lemma shows we can always add the residual capacity of an augmenting path to an existing flow and resultantly get another valid flow.

**Lemma 2.2.2.** Let $\pi$ be an augmenting path for flow $f$ in network $N$. There is a flow $f'$ for $N$ of value $|f'| = |f| + \Delta_f(\pi)$.

**Proof.** We calculate the flow $f'$ by modifying the flow of the edges in path $\pi$:

$$f'(e) = \begin{cases} f(e) + \Delta_f(\pi) & \text{if } e \text{ is a forward edge} \\ f(e) - \Delta_f(\pi) & \text{if } e \text{ is a backward edge} \end{cases}$$

Note for any backward edge $e$ we subtract the flow on $e$ already taken by $f$. The minimum residual capacity on any edge is $\Delta_f(\pi) \geq 0$ which therefore means pushing additional flow along any forward edge cannot violate any capacity constraints. It is also impossible to result in a nonzero flow on a backward edge by subtracting $\Delta_f(\pi)$. Hence, the resulting flow $f'$ is a valid flow of value $|f| + \Delta_f(\pi)$.

Lemma 2.2.2 shows that if an augmenting path exists, the given flow is not maximal.

**Lemma 2.2.3.** If no augmenting path exists in network $N$ on flow $f$, then $f$ is a maximum flow. Cut $(S, T)$ of $N$ exists where $|f| = c(S, T)$.

**Proof.** Suppose $f$ is a flow for $N$ with no augmenting path. Consider cut $(S, T)$ where $S$ is the set of all vertices $v$ such that there exists an augmenting path from $s$ to $v$ ($s$ included). Set $T = V \setminus S$. Note that each edge $e = uv$ with origin $u \in S$ and endpoint $v \in T$ must be saturated: it could otherwise form an augmenting path from $s$ to $u$ to reach the point $v \in T$. In the same
way, each edge $e$ with origin $u \in T$ and endpoint $v \in S$ must be void (zero flow). Then Lemma 2.2.1 gives $|f| = c(S, T)$, implying $f$ is maximal.

By Lemma 2.2.1 and Lemma 2.2.3, we know that the value of a maximum flow is equal to the capacity of a minimum cut.

**Theorem 2.2.4. (Integral flow theorem)** Let $N = (G, c, s, t)$ be a flow network with all integral capacities. A maximal flow on $N$ exists such that all values $f(e)$ are integral.

**Proof.** For all $e$, setting $f_0(e) = 0$ results in an integral flow on $N$ of value 0. If this flow is not maximal, an augmenting path must exist on $f_0$. In this case we know the residual capacity $\Delta_f$ of an augmenting path must be positive, which allows us to construct an integral flow $f_1$ of value $\Delta_f$. Each step we repeat the same process, increasing the flow by an integral value. After a finite number of steps an augmenting path will no longer exist. We are left with an integral flow $f$, which by Lemma 2.2.3, we know is maximal.

### 2.2.3 Ford-Fulkerson Algorithm

The Ford-Fulkerson [10, 6] algorithm calculates the maximum flow in a flow network. It applies the greedy method to the augmenting-path approach as described in Theorem 2.2.4.

At each iteration of the Ford-Fulkerson algorithm an augmenting path $\pi$ is calculated along with the residual capacity of such path. Flow equal to the residual capacity of $\pi$ is pushed along $\pi$. The algorithm terminates when the flow $f$ no longer admits an augmenting path. Lemma 2.2.3 guarantees the flow $f$ is maximal on termination. Algorithm 1 gives a pseudo-code representation of the Ford-Fulkerson algorithm.

In order to compute the augmenting path we use a modified breadth-first search (BFS) traversal or depth-first search (DFS) traversal. The modification leads us to only consider outgoing edges of current vertex $v$ where $f(e) < c(e)$ and incoming edges of $v$ with nonzero flow.

The runtime complexity of the Ford-Fulkerson algorithm depends on the method used to compute the augmenting path at each iteration. Consider a flow network $N$ with $n$ vertices and $m$ edges. Let $f^*$ be a maximal flow on $N$. At each iteration we increase the value of the flow by at least 1. Therefore, $|f^*|$ provides us with an upper bound for the number of times we search for an augmenting path before termination. Assuming we use a BFS/DFS traversal which takes $O(m)$ time, we can say the runtime complexity of the algorithm is $O(|f^*|m)$. The Ford-Fulkerson algorithm is a pseudo-polynomial-time algorithm, as its runtime is dependent on the input size and as well as the value of a numeric parameter.

The computation of an augmenting path can also be reduced to a simple path finding problem. Let $N = (G, c, s, t)$ be a flow network with a flow $f$. Consider flow network $(G', c', s, t)$ where $G'$ has the same vertex set as $G$. For each edge $e = uv$ of $G$ with $f(e) < c(e)$, an edge $e' = uv$ exists in $G'$ with $c'(e') = c(e) - f(e)$. For each edge $e = uv$ with $f(e) \neq 0$, $G'$ contains an edge $e'' = vu$ with $c'(e'') = f(e)$.

An augmenting path with respect to $f$ in $G$ corresponds to a directed path from $s$ to $t$ in $G'$. Therefore, $G'$ can be used to determine whether the flow on $f$ is maximal, and, if not, find an augmenting path. We call $N' = (G', c', s', t')$ the auxiliary network with respect to $f$. 

8
Algorithm 1 Ford-Fulkerson

1: Input: Flow network \( N = (G, c, s, t) \)
2: Output: A maximum flow \( f \) for \( N \)
3: for each edge \( e \in N \) do
4: \( f(e) \leftarrow 0 \)
5: \( stop \leftarrow false \)
6: repeat
7: traverse \( G \) starting at \( s \) to find an augmenting path for \( f \)
8: if an augmenting path \( \pi \) exists then
9: Calculate the residual capacity \( \Delta_f(\pi) \) of \( \pi \)
10: \( \Delta \leftarrow +\infty \)
11: for each edge \( e \in \pi \) do
12: if \( \Delta_f(e) < \Delta \) then
13: \( \Delta \leftarrow \Delta_f(e) \)
14: Push \( \Delta = \Delta_f(\pi) \) units of flow along path \( \pi \)
15: for each edge \( e \in \pi \) do
16: if \( e \) is a forward edge then
17: \( f(e) \leftarrow f(e) + \Delta \)
18: else
19: \( f(e) \leftarrow f(e) - \Delta \) \( \triangleright e \) is a backward edge
20: else
21: \( stop \leftarrow true \) \( \triangleright f \) is a maximum flow
22: until \( stop \)
2.2.4 Shortest Paths

In graph theory, a common problem involves finding the best path between two vertices. The definition of ‘best’ would, of course, depend on what the graph is representing. For example, a bus route planner may consider the shortest distance between two stops the best route, whereas a computer network would want a route with the fastest overall latency. To solve this problem we use weighted graphs.

Let $G = (V, E)$ be a graph with mapping $w : E \rightarrow \mathbb{R}$. We call $w(e)$ the weight or length of edge $e$ and the pair $(G, w)$, a network. For a path $\pi$, the length of $\pi$, denoted $w(\pi)$, with $n$ edges, is defined as the sum of the weights of the edges of $\pi$, that is

$$w(\pi) = \sum_{i=0}^{n} w(e_i)$$

The distance from vertex $u$ to vertex $v$ in $G$, denoted $d(u, v)$ is the length of the path of minimum total length from $u$ to $v$, if such a path exists. We call this the shortest path. If no path exists, we say $d(u, v) = \infty$. Note, an edge $e$ can have a negative weight, which can result in cycles of a negative length.

Before detailing a shortest path algorithm, let us deal with the concept of edge relaxation. For each vertex $u$ of $G$ we define a label $D[u]$ to represent the approximate distance in $G$ from $v$ to $u$. $D[u]$ will always contain the length of the current best path found. Initially, $D[v] = 0$ and $D[u] = \infty$ for each vertex where $u \neq v$. We define set $C = \emptyset$, our cloud of vertices. At each algorithm iteration, a vertex $u$ is chosen with the smallest $D[u]$ label such that $u \notin C$. Vertex $u$ is now added to $C$ and we update $D[y]$ of each vertex $y$ that is a neighbour of $u$ and outside of $C$, to show a better path from $z$ to $u$ may exist. This is known as the relaxation procedure.

2.2.5 Bellman-Ford Algorithm

The Bellman-Ford algorithm finds the shortest path in a directed graph that may contain edges of negative weight. It uses the edge relaxation technique as detailed above. Algorithm 3 provides a pseudocode representation of the Bellman-Ford algorithm.

Algorithm 2 Bellman-Ford

1: Input: A weighted digraph graph $G$ with $n$ vertices, and a vertex $v$ of $G$
2: Output: A label $D[u]$, for each vertex, defining the distance from $v$ to $u$ in $G$
3: $D[v] \leftarrow 0$
4: for each vertex $u \neq v$ of $G$ do
5: \hspace{1em} $D[u] \leftarrow \infty$
6: for $i \leftarrow 1$ to $n - 1$ do
7: \hspace{1em} for each directed edge $(u, z)$ outgoing from $u$ do
8: \hspace{2em} Perform relaxation on $(u, z)$
9: \hspace{2em} if $D[u] + w((u, z)) < D[z]$ then
10: \hspace{3em} $D[z] \leftarrow D[u] + w((u, z))$
11: if there are no edges left to relax then
12: return label $D[u]$ of each vertex $u$
13: else
14: $G$ contains a negative-weight cycle
The Bellman-Ford algorithm relaxes all the edges $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. At each iteration, the number of vertices with correctly calculated distances grows, until all vertices are labelled with their correct distance. The algorithms runtime complexity is $O(VE)$ where $V$ is the number of vertices and $E$ is the number of edges.

2.2.6 Busacker-Gowen Algorithm

The Busacker-Gowen [11, 12] algorithm is very similar to the Fork-Fulkerson algorithm, except each time the flow is updated, an augmenting path of minimal cost is chosen (see ‘Shortest Paths’ section). Finding a path of minimal cost requires the use of the auxiliary network.

Let $N = (G, c, s, t)$ be a flow network, with an integral capacity function $c$, and consider $\gamma$ a cost function such that no negative cost cycles exist in $G$. Algorithm 3 constructs an optimal flow of value $v$ on $N$ [7].

Algorithm 3 Busacker-Gowen

1: Input: Flow network $N = (G, c, s, t)$
2: Output: A minimum cost, maximum flow $f$ for $N$
3: for $e \in E$ do
4: $f(e) \leftarrow 0$
5: while $sol = true$ and $val < v$ do
6: construct the auxiliary network $N' = (G, c, s, t)$ with cost function $\gamma$
7: if $t$ is not accessible from $s$ in $G'$ then
8: $sol \leftarrow false$
9: else determine a shortest path $\pi$ from $s$ to $t$ in $(G', \gamma')$
10: $\delta \leftarrow \min \{c'(e) : e \in \pi\}$; $\delta' \leftarrow \min(\delta, v - val)$; $val \leftarrow val + \delta'$; augment $f$ along $\pi$ by $\delta'$

The boolean variable $sol$ indicates whether a solution exists, which would indicate there is a flow of value $v$ on $N$. If $sol$ is $true$ once the algorithm has terminated, then $f$ is an optimal flow of value $v$.

There are at most $v$ iterations of the while loop. At each iteration we build $N'$ and augment the flow $f$ which takes $O(|E|)$ steps. The shortest path with respect to $\gamma$ can be determined using the Bellman-Ford algorithm with complexity $O(|V||E|)$. Therefore, the optimal flow $v$ can be determined with complexity $O(|V||E|v)$. 

11
Chapter 3

Problem Definition

The following chapter provides a formal definition of the Student-Project Allocation problem (SPA) along with a solution to modelling the problem.

3.1 Definition of SPA-P

We can define an instance of the Student-Project Allocation problem (SPA) as follows.

**Definition 3.1.1.** Consider \( n, m \) and \( q \) to be positive integral values. Let \( S = \{s_1, s_2, ..., s_n\} \) be a set of students, let \( P = \{p_1, p_2, ..., p_m\} \) be a set of projects, and let \( L = \{l_1, l_2, ..., l_q\} \) be a set of lecturers. \( S, P \) and \( L \) are disjoint sets of cardinality \( n, m \) and \( q \) respectively. Each student \( s \in S \) ranks a subset of \( P \) in a strict order to form an acceptable projects set \( A_s \subseteq P \). Each lecturer \( l \in L \) forms a list of projects he wishes to supervise, denoted by \( A_l \). Each project \( p \in P \) and lecturer \( l \in L \) has a positive capacity, which we denote by \( c_p \) and \( c_l \) respectively.

An assignment \( M \) between student \( s \) and project \( p \) is denoted \((s, p)\). \( M \) is a subset of \( S \times P \) which implies \( p \in A_s \) (student \( s \) finds project \( p \) acceptable). Let us denote by \( M(p) \) the set of students assigned to project \( p \) and \( M(l) \) the students assigned to lecturer \( l \). Each project \( p \in P \) and lecturer \( l \in L \) has a positive capacity, which we denote by \( c_p \) and \( c_l \) respectively.

**Definition 3.1.2.** A matching \( M \) is an assignment such that:

i. \( |M(s)| \leq 1, \forall s \in S \),

ii. \( |M(p)| \leq c_p, \forall p \in P \),

iii. \( |M(l)| \leq c_l, \forall l \in L \)

**Definition 3.1.3.** Let us call the ranking of student \( s \in S \) the rank of \( s \)'s assigned project in \( M \). An optimal matching in the SPA problem is a maximal matching in which the sum of all students’ assigned project ranking is minimized.

3.2 Modelling the SPA-P

The SPA problem can be represented by adapting the SPA problem to flow network [1]. Computing a minimum cost maximum flow on the constructed flow network will provide us with an optimal matching between students, projects, and lecturers [2]. Let \( G = (V, E) \) be a directed graph, such that:
i. \( V = \{ s \mid s \in S \} \cup \{ p \mid p \in P \} \cup \{ l \mid l \in L \} \cup \{ \text{source}, \text{sink} \} \),

ii. \( E = \{ (\text{source}, s) \mid s \in S \} \cup \{ (s, p) \mid s \in S, p \in A_s \} \)
    \( \cup \{ (p, l) \mid p \in P, l \in A_l \} \cup \{ (l, \text{sink}) \mid l \in L \} \).

Every edge in \( G \) has an associated flow, which we represent by function \( f : E \to \{0, 1\} \), where:

\[
f(s) = \begin{cases} 
1 & \text{if } s \in S, p_j \in P \text{ and } p \in A, \\
0 & \text{otherwise}
\end{cases}
\]

Every edge in \( G \) also has an associated cost function, which we represent by \( c : E \to \mathbb{N} \), where:

\[
c(s, p) = \begin{cases} 
\text{ranking}(s, p) & \text{if } s \in S, p \in P \text{ and } p \in A, \\
0 & \text{otherwise}
\end{cases}
\]

Therefore, the SPA problem can be reduced to finding a minimum cost maximum flow for graph \( G \).

### 3.3 The Algorithm

The Busacker-Gowen algorithm, as explained in the ‘Background’ chapter, was implemented to solve the SPA problem. It is clear this provides an optimal matching between students and projects as the minimum cost restriction ensures the matching computed is optimal, whilst the flow requirement guarantees the maximum number of students are matched.

The runtime complexity of the algorithm is \( O(\sqrt{v |V||E|}) \) where \( v \) is the maximum flow (i.e. number of students), \( |V| \) is the number of vertices and \( |E| \) is the number of edges. It is evident that the more students, lecturers and projects there are, the greater the runtime execution will be. Furthermore, the larger the length of students preference list, the longer the execution time, as this results in a greater overall number of edges.

Consideration was given to Dijkstra’s shortest path algorithm [4] to be used within the Busacker-Gowen algorithm. However, it cannot find the shortest path if the graph contains cycles of negative length, which would have caused a problem if the cost function used assigned some edges negative integer values. Given this, the Floyd-Warshall all-pairs shortest path algorithm was explored which is able to deal with negative cost cycles. The Floyd-Warshall algorithm calculates the distance between all nodes in a graph, operating with \( O(|V|^3) \) complexity. The SPA problem only required the distance from one source node to be found, meaning the Floyd-Warshall algorithm was a viable option, but included a lot of unnecessary calculations which would have had a significant effect on the overall runtime performance. For this reason, the Bellman-Ford single source shortest paths algorithm was implemented, with \( O(|V||E|) \) runtime complexity.

A code snippet from the main method of the Busacker-Gowen algorithm that was implemented can be found in Appendix B.
Chapter 4

Design

In the following chapter I detail the design choices made in order to create the software application.

4.1 Back-end Application

The back-end application took the largest portion of my project time as it required the implementation of a complex algorithm. The application is to be used by a system administrator and is to be run once all project selections have been made.

4.1.1 Back-end Application Requirements

This section details the requirements of the back-end application. The application must run the matching algorithm and output the results to a database. The following requirements were identified:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Must</td>
<td>Allow the execution of the algorithm</td>
</tr>
<tr>
<td>Should</td>
<td>Allow algorithm parameters to be modified (for example, load-balancing)</td>
</tr>
<tr>
<td>Could</td>
<td>Allow parameters to be modified (e.g. capacities)</td>
</tr>
<tr>
<td>Will not</td>
<td>Allow database data modification (students, projects, etc.)</td>
</tr>
</tbody>
</table>

Table 4.1: Prioritisation of Requirements for the Back-end Application

As seen in Table 4.1.1 the scope of the back-end requirements was very narrow. However, it was essential the algorithm was complete as this was the backbone of the project. Further functionality was considered to be an 'extra'.
4.1.2 Application Architecture

The application adopted a flat structure, containing only two layers. The main layer handled the user interface and the operation of the algorithm, whilst the second layer dealt with database operations. The reason for coupling the user interface with the application layer was because of the inherent simplicity (as long as the user could make the algorithm run, no more was required). A separate database layer enabled a defined set of data access methods to perform all database activity. As a result, this reduced the overall number of SQL queries written as they were replaced with simple method calls. This also reduces the work required if the database structure is modified, as only the wrapper methods must be updated to reflect the modifications.

4.1.3 Technology Selection

All back-end code was written in Python (version 2.7). Choosing a high level language such as Python made it easier to focus on implementing the core algorithm, rather than spending time managing memory allocation and manipulating data structures. Furthermore, the object-oriented nature of Python simplified the process of handling data, allowing the creation of instance methods to perform input validation, for example.

The abundance of available Python packages was a significant pull factor. The package ‘PeeWee’, in particular, as detailed in the following ‘Implementation’ chapter, enabled easy database querying and object mapping.

4.2 Designing the Web Application

The web application provided functionality to allow students and supervisors to make selections, propose projects and choose their preferred projects. The requirements specified were gathered at the start of the project and formed the basis of development from there onwards.

4.2.1 Web Interface Requirements

The following requirements for the web interface were identified:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Must</td>
<td>1. Display all proposed projects in a library</td>
</tr>
<tr>
<td></td>
<td>2. Allow students to view &amp; select projects from such library</td>
</tr>
<tr>
<td></td>
<td>3. Allow supervisors to select &amp; propose new projects</td>
</tr>
<tr>
<td></td>
<td>4. Allow students &amp; supervisors to view and modify their current choices</td>
</tr>
<tr>
<td>Should</td>
<td>1. Allow parameters to be modified (e.g. capacities)</td>
</tr>
<tr>
<td></td>
<td>2. Have the ability to create new users</td>
</tr>
<tr>
<td>Could</td>
<td>1. Allow algorithm parameters to be modified (for example, load-balancing)</td>
</tr>
<tr>
<td>Will not</td>
<td>1. Provide email functionality to notify users</td>
</tr>
</tbody>
</table>

Table 4.2: Prioritisation of Requirements for the Web Application
The requirements were ranked using the MoSCoW method which is a prioritization technique used in software development to analyse the importance stakeholders place on the delivery of each requirement. This method was used as it would not have been possible to implement all the features initially desired. Ranking the requirements ensured key parts of the project were completed in time whilst also providing a visual aid to help with weekly workload planning.

A set of non-functional requirement must also be considered:

1. **Scalability** - Adding further functionality to the system must be possible with little rework necessary.

2. **Usability** - The interface must be suitable for its given audience (i.e. students and supervisors).

3. **Efficiency** - Solid programming practices must be adhered to in order to achieve the best possible runtime.

The incorporation of these non-functional requirements lead to an overall quality improvement in the web application. These requirements were also applied to the back-end development.

### 4.2.2 Application Architecture

The web application adopted the 3-tier client-software architecture approach which splits development into 3 sections as seen in Figure 4.1:

![3-Tier Design Architecture](image)

**Figure 4.1: 3-Tier Design Architecture**

The client tier is the top-most level of the application and is represented by the user interface. The main function of the client tier is to display information to the user in an understandable format. The business (or application) tier performs command processing, content generation and is performed on an application server. It acts as a middle man between the client and database tier, transporting data when required. The database tier is where all the relevant data is stored. On request of the business tier it will retrieve and pass the specified data. The separation of the 3 tiers allows any tier to be upgraded or replaced independently which is necessary to ensure the software is scalable, contributing to the non-functional requirement mentioned above.

Another possibility was to use the Model-View-Controller (MVC) design pattern commonly used in web applications. However, the multi-tiered approach is more geared towards scalability whereas MVC benefits those using Test-Driven-Development (TDD).
### 4.2.3 Technology Selection

HTML5 was used alongside CSS for the front-end layout of the web application. HTML5 allows for much cleaner and more descriptive code (in comparison to HTML4) which is beneficial for any future development that may occur. It also provides adaptive mobile browser support and automatic form input validation which contributes to the non-functional goal of usability.

The server scripting language PHP was used as part of the application layer to perform operations as instructed by the user. PHP integrates seamlessly with MySQL and hence it is trivial to parse and send queries to the SQL database. The majority of web servers run a PHP server as standard, so no additional services are will be required to install and operate the software when employed into university systems.

The database management system used was MySQL and hence SQL queries were sent to the database to retrieve and insert the necessary data. A relational database such as MySQL was chosen as this allows for efficient storage of data and avoids unnecessary data duplication that is associated with flat-file databases.

### 4.2.4 Application Interface Design

The web interface was designed to be self-evident, such that the user would know what everything is and how to use it just by looking at it. In order to achieve this, it was important pages had well chose names, the layout was intuitive and the text was all carefully crafted to achieve near-instantaneous recognition [9].
Chapter 5

Implementation

This chapter outlines the implementation of the software, detailing how the conceptual models from the design chapter were developed. The chapter is split into two major sections in the same way as the design chapter; one for the front-end and one for the back-end. Note, the front-end section only contains a brief overview, which reflects how my time was distributed between the front and back-end.

5.1 Back-end Application

The back-end application, responsible for executing the matching algorithm, has three major functions which operate sequentially:

1. Reading and writing data to and from the database
2. Building the flow network from the retrieved data
3. Optimizing the flow using the Busacker-Gowen algorithm

5.1.1 Reading the Data

The data stored in a MySQL database must be read by the back-end Python application. A Python module named ‘PeeWee’ assisted with this problem. PeeWee is a lightweight object-relational mapper (ORM) that provides an API for interacting with relational databases. The main motivation for using an ORM such as PeeWee is that it automatically maps database query results to object members (it will generate a student object automatically when reading from the student table, for example).

An example of a PeeWee query is shown below. The following code retrieves the projects chosen by a specified student.

```python
def get_projects(self):
    return Projects.select().join(Selections, on=Selections)
    .where(Selections.studentid == self.id)
```

The query shown will return a list of Project objects as opposed to a raw SQL query which returns plain text data. PeeWee provides functionality to write data to the database in the same way that it is read.
5.1.2 Representing the Flow Network

In order to represent the flow network a flow-cost graph class was created. The core data structure was a Python dictionary that, when nested inside itself, formed a $n \times n$ matrix (where $n$ is the total number of nodes). A separate object was created to store the edge information. At each position in the matrix (that was not null) resided an instance of such object. For example, at matrix position [5, 7] you would find an object that contained the flow, capacity and cost values for the edge created by ordered nodes 5 and 7.

Instance methods were written to avoid direct manipulation of the graph data structure. In doing this human programming error is prevented which saves a significant amount of time during debugging (as well as making the code more user-friendly).

5.1.3 Building the Flow Network

The application was required to read data from the database and build a directed flow network to represent the data accordingly. This included reading all students, projects and lecturers, along with the choices they made prior to running the application. In order to do this a builder class was created. It operated as follows:

1. Add all the nodes the flow network iteratively (students, projects and lecturers)
2. Connect the nodes to represent the selections made by students and lecturers
3. Add edge information (flow, cost and capacity)

A graphical representation of the flow network built can be seen in Figure 5.1. From our source node to each student, we assigned a capacity of 1 to represent that each student can be assigned to, at most, 1 project. In order to represent the capacity of a project we created a duplicate project node, and created an edge between both project nodes, assigned with the capacity of the project. We can see in Figure 5.1 that Project V has a capacity of 2, for example. Each supervisor is represented by one network node, and their project capacity is assigned to the edge between themselves and the sink node. On the edges directed outward from student “Tom”, we can see the assigned cost to each of his choices. His first choice is represented by 1, and his last choice by 3 (we assume in this case each student can make 3 project choices). The edge formed between the rightmost project node and the supervisor node is assigned a capacity equal to that of the project. By default, every other edge in the flow network is assigned a cost of 0 and a capacity of 1.
Python’s object-oriented nature allowed me to create objects for each of the students, projects and lecturers and write instance methods to speed up operations on each of the objects. An example of such a method allowed me to add the given object to the flow network with no extra parameters required (also a preventative measure against future bugs).

5.1.4 Optimizing the Flow

Once the flow network had been constructed and a flow added using data from the database, the Busacker-Gowen algorithm executed and optimized the given flow. The matching algorithm runs iteratively, each time augmenting along a path of minimal cost (and resultantly forming one match) until there is no longer a path through the flow network from source to sink (see Algorithm 3).

5.1.5 Outputting the Results

Identifying the matches on the optimized flow required the graph to be scanned along each complete path from source to sink. A path from source to sink is representative of a single matching between student, project and supervisor. A triple nested for loop was implemented that scanned the edges formed between the students, projects and supervisors.

5.2 Front-End Application

The front-end web interface provided the functionality to allow students and lecturers to choose and propose projects, respectively. The implementation of the web interface can be broken up into the following two stages:

1. Designing and developing the template
2. Implementing the desired functionality
5.2.1 Template Design

The web interface had a simple set of requirements as seen in the ‘Design’ chapter. To reflect this I kept the layout basic and uncluttered, such that all actions could be completed within two clicks.

The interface consisted of 4 separate pages. Each component of functionality was given its own page (for example, view library, view choices etc.) to increase the speed that all tasks can be performed on the website. The home page can be seen in Figure 5.2.

![Figure 5.2: A screenshot of the web interface](image)

5.2.2 Adding Functionality

All back-end functionality was written in PHP alongside SQL for the database querying. The MySQLi package provided the ability to send SQL queries through PHP in a secure manner. A database access wrapper class was written, that all others classes evoked to perform any database related activity. This added a layer of abstraction to the back-end which is ideal for future changes, as only the wrapper class would need to be modified.
Chapter 6

Evaluation

This chapter describes the testing performed in the project and the measures taken to avoid any other identified problems.

6.1 Student Randomization

In order to achieve fairness it was required that the students were randomized before they were inserted into the flow network. This was necessary because of the nature of the matching algorithm, in which a match is identified at each iteration. As a result, the students who were matched in the early stages of the algorithm were much more likely to receive their first choice project than someone matched towards the end. Randomization removed any ordering the students may have had within the database, thus giving each student an equal chance of being inserted at position $x$ when generating the flow network.

6.2 Quality Assurance

The matching algorithm was tested extensively on data in which the optimal match was already known to ensure the algorithm was functioning correctly. However, only small sets of data were used in these tests (4 students, 4 projects, 4 supervisors) as larger datasets were too complex to solve without automation. In order to account for this, I tested many different small datasets, with known optimal matchings, until I was fully convinced the algorithm was implemented as required.

6.3 Performance Testing

Performance testing was done to simulate different scenarios with varying numbers of students, projects and lecturers. The time for the algorithm to complete execution was recorded each test. All the tests were benchmarked on the same PC with specifications as follows:

- **CPU**: Intel i7-2860QM @ 2.50 GHz (4 cores, 8 threads)
- **RAM**: 16GB DDR3 @ 1600 MHz
- **Operating System**: Windows 7 64-bit
The tests performed involved 3 randomly chosen projects assigned to each students preference list (with an even probability distribution over all projects). The same applied to the lecturers.

Figure 6.1 shows how the execution time of the algorithm varies with the instance size. Each test used an equal number of students, projects and lecturers. For example, 20 students, 20 projects and 20 lecturers. The graph clearly indicates a significant increase in execution time as the number of nodes increases, which would correspond to an increase in students, projects and lecturers. Note, the instance size is the complexity for the given test.

![Execution time dependence on instance size](image1)

Figure 6.1: A performance graph to show how the runtime of the matching algorithm varies with the instance size. Note, each student and lecturer made 3 project choices.

![Students’ total rankings dependence on instance size](image2)

Figure 6.2: A graph to show how the total ranking of the students’ matches varies with the instance size. Note, each student and lecturer made 3 project choices.

It is clear to see in Figure 6.2 that as the instance size grows the total ranking of the stu-
dents projects grows, which is expected since the ranking will grow as the number of students increases. Note the gradient decrease as the instance size grows, particularly from around 0.15 onwards. This is an indication that more students are getting a higher ranked project of their choice as the instance size grows.

Test result data can be found in Appendix A.
Chapter 7

Conclusion

This final chapter concludes the development work completed during the project. It highlights the challenges and achievements experienced and also details further possible development.

7.1 Achievements

A solution to the Student-Project Allocation problem was found. A matching algorithm was developed and implemented that output an optimal matching between students and projects, such that no student could obtain a better project in the same maximal matching. The algorithm performed with a runtime that is viable to be used within university institutions.

7.2 Challenges

Due to time constraints I was not able to perform the extensive runtime testing that I initially planned. Configuring different datasets to perform tests on was time consuming and as a result the tests included in the report contained very few data points. However, I believe the data gathered was sufficient to show how the execution time varied with the instance size, which is a critical part of performance analysis.

7.3 Future Work

The project in its current state performs well and has no known issues. However, there is a lot of room for development with both the front-end and back-end. The most prominent features for future research would be:

- Load balancing for lecturers
- Improving the algorithm runtime

The implementation of load balancing would increase the overall fairness amongst lecturers, evening out the distribution of projects to lecturers. Currently, the algorithm will assign the best project possible to the student, regardless of the lecturers remaining capacity. A possible implementation would assign a cost function to the lecturers project choices, such that each additional project they are assigned has a higher cost associated with it.
I recently became aware there exists a modified version of Dijkstra’s algorithm that enables it to be used with negative cost values and terminate as soon as the required path costs have been calculated. I would be interested to see, when implemented, how this affects the overall runtime performance of the algorithm. Dijkstra’s worst case runtime is $O(V^2)$, which is no better than the Bellman-Ford shortest paths algorithm currently implemented. However, with early termination we may see an improvement over the Bellman-Ford algorithm when run on a large dataset.
Glossary

**SPA**  Student-Project Allocation

**SPA-P**  Student-Project Allocation problem

**API**  Application programming interface

**Foreign Key**  a column or combination of columns that is used to establish and enforce a link between the data in two tables
Bibliography


Appendix A

Test Results

<table>
<thead>
<tr>
<th>No. of Students</th>
<th>Instance Size</th>
<th>No. of Instances</th>
<th>Total Matches</th>
<th>Average time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>36450</td>
<td>5</td>
<td>9</td>
<td>3.69</td>
</tr>
<tr>
<td>14</td>
<td>137200</td>
<td>5</td>
<td>14</td>
<td>19.16</td>
</tr>
<tr>
<td>19</td>
<td>342950</td>
<td>5</td>
<td>19</td>
<td>59.05</td>
</tr>
<tr>
<td>25</td>
<td>781250</td>
<td>5</td>
<td>25</td>
<td>180.24</td>
</tr>
<tr>
<td>30</td>
<td>1350000</td>
<td>3</td>
<td>30</td>
<td>373.86</td>
</tr>
<tr>
<td>35</td>
<td>2143750</td>
<td>3</td>
<td>35</td>
<td>677.54</td>
</tr>
<tr>
<td>40</td>
<td>3200000</td>
<td>2</td>
<td>40</td>
<td>1168</td>
</tr>
<tr>
<td>49</td>
<td>5882450</td>
<td>1</td>
<td>49</td>
<td>2587</td>
</tr>
<tr>
<td>59</td>
<td>10268950</td>
<td>1</td>
<td>59</td>
<td>5455</td>
</tr>
</tbody>
</table>

Table A.1: Performance data

<table>
<thead>
<tr>
<th>No. of Students</th>
<th>Instance Size</th>
<th>No. of Instances</th>
<th>Total Ranking (Cost)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>36450</td>
<td>5</td>
<td>53</td>
</tr>
<tr>
<td>19</td>
<td>342950</td>
<td>5</td>
<td>126</td>
</tr>
<tr>
<td>25</td>
<td>781250</td>
<td>5</td>
<td>159</td>
</tr>
<tr>
<td>30</td>
<td>1350000</td>
<td>3</td>
<td>191</td>
</tr>
<tr>
<td>35</td>
<td>2143750</td>
<td>3</td>
<td>218</td>
</tr>
<tr>
<td>40</td>
<td>3200000</td>
<td>2</td>
<td>252</td>
</tr>
<tr>
<td>49</td>
<td>5882450</td>
<td>1</td>
<td>308</td>
</tr>
<tr>
<td>59</td>
<td>10268950</td>
<td>1</td>
<td>372</td>
</tr>
</tbody>
</table>

Table A.2: Total ranking data
Appendix B

Busacker-Gowen Snippet

The following is a code excerpt from the main Busacker-Gowen method.

```python
while True:
    cpt += 1
    # Bellman Ford shortest paths algorithm
    path = self.g.get_resulting_network_with_costs()
    .get_shortest_path(self.g.index_of_source(),
                       self.g.index_of_sink())

    # if path is not empty
    if path:
        delta = self.get_possible_augmentation(path)
        self.update_flow(path, delta)
    # fail condition
    if not path or wanted_flow == self.g.get_graph_flow():
        break
```

30