Technische Universität München
Fakultät für Informatik

Assignment Problem with Constraints

Zuordnungsproblem mit Nebenbedingungen

Diplomarbeit

Ulrich Bauer

Aufgabenstellerin:	Prof. Dr. Angelika Steger
Betreuer:	Andreas Weißl
Abgabedatum:	15.Mai 2005

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 15. Mai 2005

Contents

1 Introduction **1**

1.1 Background 1

1.2 Logistic Problems in Warehousing 2

1.3 Overview 3

2 Problems **5**

2.1 Minimum Cost Flow 5

2.1.1 Duality 7

2.1.2 Complementary Slackness 8

2.2 Assignment 9

2.3 Asymmetric Assignment 11

2.4 Assignment with Subset Constraints 13

2.4.1 Strict Constraints 14

2.4.2 Loose Constraints 16

3 Basic Algorithms **19**

3.1 Push/Relabel Algorithms 19

3.2 Cost Scaling 20

3.3 Approximate Optimality 21

3.4 Generic Cost Scaling Algorithm (Goldberg/Tarjan) 21

3.5 Bipush (Ahuja/Orlin) 23

3.6 Auction Algorithm 24

3.7 Auction as a Flow Algorithm (double-push) 27

3.8 Forward/Reverse auction 30

3.9 Other Algorithms 31

4 Asymmetric Assignment 33

4.1 Auction Algorithm for Asymmetric Assignment 33

4.2 Virtual Nodes 34

4.3 Using a Heap for Prices 34

4.4 Relabeling Unmatched Nodes 36

4.5 Reverse Auction 36

5 Constrained Assignment 39

5.1 Using Minimum Cost Flow 39

5.2 Using Bipartite Network Circulation 39

5.3 Adapting the Auction Algorithm for Constrained Assignment 40

5.4 Strict Constraints 42

5.5 Loose Constraints 43

6 Implementation 45

6.1 Common Aspects 45

6.2 Auction with Heap 46

6.3 Forward/Reverse Auction 47

6.4 Auction for Constrained Assignment with Heaps 47

7 Computational Results 49

7.1 Edge Density 50

7.2 Asymmetry Between Vertex subsets 50

7.3 Graphs with Different Vertex Set Sizes 54

7.4 Graphs with Increasing Maximal Costs 54

7.5 Constrained Assignment 56

A Generating Sparse Random Bipartite Graphs 61

Chapter 1

Introduction

1.1 Background

The problem setting for this thesis was initiated by a joint project—"New combinatorial algorithms in logistics optimisation"—of TU München with Munich-based Axxom Software AG. The objective of this project was to develop new algorithms at TU München which are especially geared towards problems in logistics and should replace existing standard components of Axxom's optimisation software. The project focused not only on algorithms of theoretical interest, but particularly on algorithms which show very good performance in practice. In the context of the subproject "Assignment" [Unt03], an implementation of an algorithm for solving the well-known assignment problem has already been developed. This thesis is based on the work done during that project.

After Prof. Steger was appointed to ETH Zürich and her research group left for ETH, the subprojects that have already been started were completed in cooperation with Axxom; however, since the projects depended on research funds which were not transferable abroad, direct cooperation was cancelled afterwards for the time being.

Anyway, the research group of Prof. Steger continued work on the subject, with a focus on extending the existing algorithms to include additional constraints which are of vital importance for practical usage. Previously these constraints have only been taken into account in a postprocessing step on a solution found by a generic assignment algorithm, without making use of the information gathered during the execution of the algorithm. This could lead to unpredictable degradation in running time as well as in quality of the solution; the techniques used included genetic algorithms which are intractable with established techniques of algorithm analysis in most cases.

For that reason, the objective of this diploma thesis was to incorporate these constraints into the algorithm to improve the quality of the solution and to achieve a better running time.

1.2 Logistic Problems in Warehousing

Combinatorial optimisation plays an important role in logistics, and many of the basic problems and algorithms find a direct application in this area or even originate from it. For instance, shortest paths are used to find the cheapest transportation routes, network flow algorithms are used to optimise transportation of goods, the assignment problem asks for a cost-optimal assignment of workers to tasks or products to warehouse locations.

However, often these simple and abstract problems do not completely represent the actual optimisation problem that has to be solved. Sometimes slight deviations or additional constraints have to be taken into account to accurately describe the problem, and these may not come to attention until a real-world implementation shows the need for them. Consider as an example the history of the work preceding this thesis. Axxom, a company for logistics software, looks for a faster algorithm to optimise the distribution of products to warehouse locations. In cooperation with a university, efficient algorithms for solving the so-called assignment problem are found and implemented, and this implementation this then applied to the warehousing problem setting, with a significantly better performance. In the real-world application, however, it already had shown up before that the assignment problem did not completely describe the logistic problem to be solved. Sometimes the cheapest solution involved leaving a certain warehouse completely empty; similar products placed next to each other could lead to mix-up, which should be avoided. Consequently, the next step for the university group was to look closer at incorporating additional constraints into the assignment problem and adapting the found algorithms for the new problem where possible.

1.3 Overview

The rest of this thesis is structured as follows. In Chapter 2, we will present the problems we investigated, along with their generalisations, which are well-known and well-analysed combinatorial optimisation problems, namely the minimum-cost flow problem and the assignment problem. We look at the problems from a mathematical point of view and use Linear Programming theory to state some important facts that help us in finding and checking optimal solutions to our problems. We will state two versions of the assignment problem with constraints, one of which will be the main subject of this thesis.

In Chapter 3, we provide insight into some of the fastest known algorithms for these standard problems. Based on the theory we developed in Chapter 2, we will explain why these algorithms are correct. We will also show how two classes of algorithms, the auction algorithms and the push/relabel algorithms, are related.

In Chapter 4, we modify one of these algorithms, the auction algorithm, to efficiently solve a generalisation of the assignment problem, namely the asymmetric assignment problem, where one set is allowed to be larger than the other. We show, again based on the theoretical insight we gained in Chapter 2, which additional problems arise in this case, and we will present two previously developed approaches as well as our own approach to the problem.

Based on this algorithm, in Chapter 5 we will describe how to solve the assignment problem with constraints, which is again a generalisation of the asymmetric assignment problem. We will discuss different approaches and describe in detail how to modify an algorithm for the asymmetric assignment problem to be able to solve the assignment problem with constraints.

Chapter 6 covers some implementation-specific details that do not belong to the description of the algorithms themselves, but are worth noting nonetheless. This section is especially interesting for readers who also want to use or study the source code of the implemented algorithms.

In Chapter 7, we finally compare running times of our implementations empirically. We will look for anomalies and unexpected behaviour and try to explain these results, and we will rate the strengths and weaknesses of the specific implementations.

Chapter 2

Problems

In this chapter, we will introduce and analyse all of the problems covered in this thesis from a mathematical point of view. We will present linear programming (LP) versions of the problem and investigate the properties of these linear programs. Although no algorithms are developed in this chapter yet, the analysis of the structure of the problems will be used later for constructing the algorithms, which all heavily depend on the theory we will present here.

2.1 Minimum Cost Flow

We introduce a general and very fundamental network problem, the minimum cost flow problem. All of the problems described later on can be reduced to it, and a generic algorithm for solving this problem is a good starting point for developing specialised algorithms for the more specific problems.

The problem consists of finding a minimum cost way of transporting material through a network from supply to demand nodes. Every node on the network has a certain supply or demand of material. Every edge on the network has a lower and an upper bound on the amount of material that can be transported over it, and also a certain cost for every unit of material that is shipped over that edge. We want to find out how much material we have to send over each edge to meet all of these requirements at minimum cost.

The minimum cost flow problem can be expressed as a linear optimisation problem. This allows us to use an enormous amount of knowledge and algorithms developed around linear programming theory. For example, we could simply use the well-known simplex algorithm to solve the minimum cost flow problem. However, the special structure of this optimisation problem allows for algorithms with a much better guaranteed running time than generic LP algorithms; anyways, LP theory is always a vital component for the design and analysis of these algorithms.

Let $G = (V, E)$ be a directed network with associated edge costs c_{ij} for each edge $(i, j) \in E$, denoting the cost to transport one unit of material over that edge. n denotes the number V of vertices, and m the number of edges in the graph. Moreover, let l_{ij} and u_{ij} be the lower and upper flow capacity bounds of an edge (i, j), and let $b(i)$ be the supply of node i. We call a node i *supply node* if $b(i) > 0$, *demand node* if $b(i) < 0$, and *transshipment node* if $b(i) = 0$. A minimum cost flow problem that has only transshipment nodes is also called a *minimum cost circulation* problem. The variables in the linear program, x_{ij}, indicate how many units of flow are sent over edge (i, j).

Some algorithms also expect the network to be *antisymmetric*: each edge can be looked at in both directions, with reversed cost ($c_{ji} = -c_{ij}$), reversed flow ($x_{ji} = -x_{ij}$) and reversed bounds ($l_{ji} = -u_{ij}$ and $u_{ji} = -l_{ij}$). This is merely a way of notation and not a restriction on the input, and allows a simpler description of some algorithms.

We can now state the minimum cost flow problem as a linear program:

$$\text{Minimise} \sum_{(i,j)\in E} c_{ij}x_{ij} \tag{2.1}$$

subject to

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = b(i) \qquad \text{for all } i \in V, \tag{2.2}$$

$$l_{ij} \le x_{ij} \le u_{ij} \qquad \text{for all } (i,j) \in E, \tag{2.3}$$

where $\sum_{i=1}^{n} b(i) = 0$.

We call the constraints in (2.2) *flow conservation constraints.* The two sums represent the total amount of flow leaving and entering the node, respectively. The flow conservation constraints demand that for every node the outgoing flow minus the incoming flow equals the supply/demand of the node.

Moreover, the flow must satisfy the *capacity constraints* represented in (2.3). These constraints say that for every edge the amount of flow must lie between the lower and upper bounds. In most cases, the lower bounds l_{ij} are expected to be 0 (e.g. in [GT90]) and are not stated in the problem.

Usually, the costs and capacities of the stated problem are integral, and integral solutions are required as well. In general, an *integral linear program* (ILP) is much harder to solve than a LP with real values, and the optimum of an ILP might be different form the optimum of an according LP (*integrality gap*). However, the node-edge incidence matrix has a special property called *total unimodularity*, which guarantees that every instance of the problem has an integral optimal solution (see [PS98] for details). Therefore, we can work with the LP version of the problem (the *LP relaxation*) and still find an optimal solution of the IP. To avoid fractional results, we will always work with integral values in our algorithms.

2.1.1 Duality

For every linear program, there exists a closely related linear program, called the *dual* program. It has the property that the objective function value of an optimal solution is identical to that of the primal program. This is called the *strong duality theorem* (see for example [PS98]). For a minimisation problem, the dual program is a maximisation problem and vice versa. Assuming that the lower capacity bounds l_{ij} are equal to 0, the dual problem for the minimum cost flow problem is

$$\text{Maximise} \sum_{i \in V} b(i)\pi(i) - \sum_{(i,j)\in E} u_{ij}\alpha_{ij} \tag{2.4}$$

subject to

$$\pi(i) - \pi(j) - \alpha_{ij} \le c_{ij} \qquad \text{for all } (i,j) \in E, \tag{2.5}$$

$$\alpha_{ij} \ge 0 \qquad \text{for all } (i,j) \in E. \tag{2.6}$$

In the context of minimum cost flow, the dual variables $\pi(i)$ can be interpreted as price labels on the nodes of the network, indicating the cost of obtaining a unit of commodity at that node. The prices of the head and tail of an edge are added and subtracted, respectively, from the cost of the edge, to calculate the actual price of pushing flow over that edge. We will use the following definition:

Definition 1. *The* reduced cost c^{π}_{ij} *of an edge* (i,j) *in a network* $G = (V,E)$ *with regard to a price function* $\pi : V \to \mathbb{R}_+$ *and a cost function is defined as*

$$c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j)$$

where c_{ij} *is the actual cost of the edge* (i,j)*.*

An interesting observation is that we can simply use the reduced costs instead of the actual cost for *any* price function π, since the optimal solutions for the minimum cost problem are the same for both the reduced costs and the actual costs, as noted in [AMO93, p. 43].

Sometimes the reduced costs are defined in a different way; for example in [GK95] the signs on the prices are reversed. We will only use the above definition, which is the same as in [AMO93].

Besides the variables $\pi(i)$, there are also additional dual variables α_{ij}. To understand how these variables can be interpreted, observe that we can rewrite (2.5) as

$$\alpha_{ij} \geq -c^{\pi}_{ij}. \tag{2.7}$$

Since we want to maximise $-u_{ij}\alpha_{ij}$, we always set α_{ij} to the lowest possible value, that is

$$\alpha_{ij} = \max(0, -c^{\pi}_{ij}). \tag{2.8}$$

Thus, it is sufficient to only compute the price values $\pi(i)$, as the dual variables $alpha_{ij}$ can be computed from the prices.

2.1.2 Complementary Slackness

Linear programming theory provides some simple and powerful tools to verify that a solution found by an algorithm is optimal. In fact, we will use one of these tools, the *complementary slackness conditions*—in a slightly relaxed form—also in the algorithms to work towards an optimal solution.

In a simple description, complementary slackness means that for each inequality constraint of the LP, either the constraint holds with equality or the according dual variable is zero (or both). Since the dual of a dual program is the primal program, the same is true for the dual LP. See [PS98] for more details. Note that for constraints of the form $x \geq 0$, the according dual variable does not appear in the dual objective function, and consequently the respective complementary slackness condition is not needed as well.

For our minimum cost flow LP, we get the following complementary slackness conditions:

$$(\pi(i) - \pi(j) - \alpha_{ij} - c_{ij})x_{ij} = 0 \qquad \text{for all } (i,j) \in E \tag{2.9}$$

$$(x_{ij} - u_{ij})\alpha_{ij} = 0 \qquad \text{for all } (i,j) \in E \tag{2.10}$$

Equation (2.9) is derived from $\pi(i) - \pi(j) - \alpha_{ij} \leq c_{ij}$ (2.5); Equation (2.10) from $x_{ij} \leq u_{ij}$ (2.3).

We use the reduced costs we just defined to formulate these conditions in a way that is more convenient:

Theorem 1. *A flow x is an optimal solution to the minimum cost flow problem, if for some collection of price values π the following conditions are satisfied for each edge (i, j):*

$$\text{If } c^{\pi}_{ij} > 0, \text{ then } x_{ij} = 0. \tag{2.11}$$

$$\text{If } 0 < x_{ij} < u_{ij}, \text{ then } c^{\pi}_{ij} = 0. \tag{2.12}$$

$$\text{If } c^{\pi}_{ij} < 0, \text{ then } x_{ij} = u_{ij}. \tag{2.13}$$

Proof. Equation (2.9) can be written as $(c^{\pi}_{ij} + \alpha_{ij})x_{ij} = 0$. Since $\alpha_{ij} \geq 0$, if the reduced costs c^{π}_{ij} are greater than 0, the sum $(c^{\pi}_{ij} + \alpha_{ij})$ is greater than 0, and x_{ij} has to be 0.

If $c^{\pi}_{ij} < 0$, then, according to Equation (2.7), $\alpha_{ij} > 0$. Now Equation (2.10) only holds if $x_{ij} = u_{ij}$.

If $x_{ij} < u_{ij}$, Equation (2.10) requires that $\alpha_{ij} = 0$. If now $x_{ij} > 0$ holds as well, then $c^{\pi}_{ij} + \alpha_{ij} = 0$ according to Equation (2.9), and so c^{π}_{ij} has to be 0. □

Complementary slackness conditions show a correlation between the primal and dual variables; they basically express the main idea of LP-duality. Often, they have an intuitive explanation. In this case, the complementary slackness conditions can be expressed informally as follows:

- If additional costs are involved, send as little as possible flow over the edge.
- If it does not matter, send an arbitrary amount of flow over the edge.
- If it is profitable, send as much flow as possible over the edge.

If these conditions are met at every edge, then the flow is optimal.

2.2 Assignment

The following problem is known as the (symmetric) *assignment* problem: Given a set X and a set Y of objects (both sets of equal size, i.e. $|X| = |Y| = \frac{n}{2}$), a collection of allowed pairs $E \subseteq X \times Y$, and a cost c_{xy} for each possible pair $(x, y) \in E$, we want to pair each element in X with an element in Y at minimum total cost. We assume that there is an allowed pair for every element, i.e. there is a solution to the problem; consequently $m \geq n$.

In the context of graphs, this problem can be expressed as the problem of finding a minimum weight perfect matching in the weighted, bipartite graph $G = ((X \cup Y), E)$ with edge weight c_{xy} for each edge $(x, y) \in E$. We can reduce the assignment problem to the minimum cost flow problem on this graph by setting $b(i) = 1$ for all $x \in X$ and $b(i) = -1$ for $y \in Y$, and capacity $u_{ij} = 1$ for all edges $(i, j) \in E$. An integral solution for this problem clearly corresponds to an optimal solution of the assignment problem.

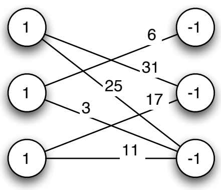


Figure 2.1: Minimum cost flow formulation of the symmetric assignment problem

The linear program for the assignment problem can be simplified to

$$\text{Minimise} \sum_{(i,j) \in E} c_{ij} x_{ij} \tag{2.14}$$

subject to

$$\sum_{j:(i,j) \in E} x_{ij} = 1 \qquad \text{for all } i \in X, \tag{2.15}$$

$$\sum_{i:(i,j) \in E} x_{ij} = 1 \qquad \text{for all } j \in Y, \tag{2.16}$$

$$x_{ij} \geq 0 \qquad \text{for all } (i, j) \in E. \tag{2.17}$$

Again, we are looking for an integral solution, but of course we can work with the LP relaxation again. The most significant simplification in this LP is that the upper capacity bounds u_{ij} are not needed, since the other constraints already imply that $x_{ij} \leq 1$. Therefore, in the dual program the α_{ij} are not needed (all α_{ij} are zero).

The according dual program is

$$\text{Maximise} \sum_{i \in X} \pi(i) - \sum_{j \in Y} \pi(j) \tag{2.18}$$

subject to

$$\pi(i) - \pi(j) \leq c_{ij} \qquad \text{for all } (i,j) \in E. \tag{2.19}$$

As a consequence, we know that for an optimal assignment, all reduced costs are nonnegative ($c^{\pi}_{ij} \geq 0$), since $\alpha_{ij} = \max(0, -c^{\pi}_{ij}) = 0$ for all edges $(i,j) \in E$. Consequently, the complementary slackness conditions reduce to

$$\text{If } c^{\pi}_{ij} > 0, \text{ then } x_{ij} = 0. \tag{2.20}$$

$$\text{If } x_{ij} = 1, \text{ then } c^{\pi}_{ij} = 0. \tag{2.21}$$

Observe that if we have found an optimal assignment, the dual variables $\pi(i)$ for $i \in X$ can be calculated from the values $\pi(j)$, $j \in Y$, and the flow variables x_{ij}: If $x_{ij} = 1$, then $\pi(i) = c_{ij} + \pi(j)$. Therefore these variables are somewhat redundant, and we can state a version of the complementary slackness conditions in a form that does not require them:

$$\text{If } x_{ij} = 1, \text{ then } c_{ij} + \pi(j) \leq c_{ik} + \pi(k) \ \forall\ (i,k) \in E. \tag{2.22}$$

Equation (2.21) is obviously satisfied by the way we defined $\pi(i)$: for the node j assigned to i, $\pi(i) = c_{ij} + \pi(j)$ and therefore $c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j) = 0$. To show that Equation (2.20) follows from (2.22), observe that c^{π}_{ij} can only be greater than 0 if i is not assigned to j: for $x_{ij} = 1$, we set $\pi(i) = c_{ij} + \pi(j)$ and therefore $c_{ij} = 0$.

2.3 Asymmetric Assignment

If the sets X and Y are not of equal size (say, $n_1 < n_2$, where $n_1 = |X|$ and $n_2 = |Y|$), the problem is called *asymmetric* assignment problem. We still have to search for a minimum weight maximal matching. However, a perfect matching does not exist; some of the elements in Y have to remain unmatched.

We can, as a result, not simply adopt the reduction to minimum cost flow for the symmetric assignment problem, since we cannot assign demand -1 to each node in Y. Instead, we have to introduce an additional node t and edges (y,t) for each $y \in Y$ with cost $c_{yt} = 0$ and capacity $u_{yt} = 1$. Now we can set the demand of the additional node t to $|X|$. An integral solution of the minimum cost flow problem on this extended network again corresponds to a solution of the asymmetric assignment problem: if the flow on one edge $(x,y) \in X \times Y$ is 1, then x is assigned to y.

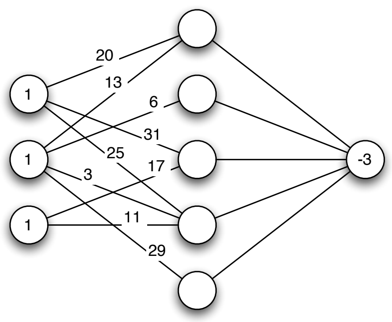


Figure 2.2: Minimum cost flow formulation for the asymmetric assignment problem

In an optimal integral flow for this network, according to the complementary slackness conditions, the price of the additional terminal node t has to be less or equal to the prices of assigned nodes and greater or equal to the prices of unassigned nodes.

This additional node, however, is only an auxiliary construction to reduce the asymmetric assignment problem to the minimum cost flow problem. The canonical LP relaxation of this problem does not require the terminal node. We will see, however, that the condition we just stated, namely that the prices of unassigned nodes must not be above the prices of assigned nodes, holds for this formulation as well.

The linear program for the asymmetric assignment problem is not much different from the LP for symmetric assignment. The only difference is that we do not require every node of the right hand side to be assigned to a left hand side node:

$$\text{Minimise} \sum_{(i,j)\in E} c_{ij}x_{ij} \tag{2.23}$$

subject to

$$\sum_{j:(i,j)\in E} x_{ij} = 1 \qquad \text{for all } i \in X, \tag{2.24}$$

$$\sum_{i:(i,j)\in E} x_{ij} \leq 1 \qquad \text{for all } j \in Y, \tag{2.25}$$

$$x_{ij} \geq 0 \qquad \text{for all } (i,j) \in E. \tag{2.26}$$

Consequently, the according dual program is very similar to the dual for the symmetric assignment problem as well:

$$\text{Maximise} \sum_{i\in X} \pi(i) - \sum_{j\in Y} \pi(j) \tag{2.27}$$

subject to

$$\pi(i) - \pi(j) \leq c_{ij} \qquad \text{for all } (i,j) \in E, \tag{2.28}$$

$$\pi(j) \geq 0 \qquad \text{for all } j \in Y. \tag{2.29}$$

The only difference between this dual program and the dual for symmetric assignment is that the price variables $\pi(i)$ have to be greater than 0. However, there is another important difference. Since, as opposed to the symmetric assignment problem, (2.25) is an inequality constraint, we have a corresponding additional complementary slackness condition:

$$\Big(\sum_{i:(i,j)\in E} x_{ij} - 1\Big)\pi(j) = 0 \qquad \text{for all } j \in Y \tag{2.30}$$

or, in words, the price $\pi(j)$ of every unmatched node $j \in Y$ has to be zero.

These conditions on the node prices seem to be stricter than the previously mentioned condition for the minimum cost flow version of the problem, where assigned nodes simply have higher prices than unassigned ones. We can, however, easily observe that the two problem formulations are essentially equivalent. Assume that we have an optimal primal solution and suitable dual values for a minimum cost flow instance of the problem. We can set the prices of the unassigned nodes to the lowest price value of the assigned nodes. Moreover, we can shift the price values $\pi(i)$ together by an arbitrary amount without changing the objective function value or feasibility of the dual solution. In particular, we can now shift the price values such that the lowest price value of the assigned nodes and the prices of the unassigned nodes are zero. That way, we can transform any optimal dual solution of the minimum cost flow reduction to an optimal dual solution for our LP.

We will see later that the small difference in the structure of the symmetric and the asymmetric assignment problem actually has an impact on our algorithms: some algorithms can solve the symmetric assignment problem, but they fail on the asymmetric assignment problem.

2.4 Assignment with Subset Constraints

Given a weighted bipartite graph $(X \cup Y, E)$ with $|X| < |Y|$, and a partition of Y into disjoint subsets $Y_1, Y_2, \ldots, Y_s$. Each of the subsets Y_i is assigned a capacity l_i which denotes the maximal number of matched vertices allowed in this subset. The problem is now to find a minimum cost maximal matching.

Obviously, the capacities must sum up to a number greater than the number of elements which are to be assigned: $\sum_{i=1}^{s} l_i \geq |X|$. Otherwise, the problem would be infeasible, since not every element of X can be assigned to an element of Y without violating some of the capacity constraints. It is also easy to observe that for an instance of the symmetric assignment problem ($|X| = |Y|$), the capacity constraints are met by every feasible solution; therefore we can assume that the underlying assignment problem is asymmetric.

As we will see later, it is important to distinguish two cases: $\sum_{i=1}^{s} l_i = |X|$ and $\sum_{i=1}^{s} l_i > |X|$. In the first case, the capacities l_i for each subset are strict requirements: exactly l_i of the nodes in Y_i have to be matched; we talk of *strict capacity constraints*. In the second case, the capacities l_i are merely upper bounds for the number of matched nodes in Y_i; we call these *loose capacity constraints*.

2.4.1 Strict Constraints

We will now take a closer look at the problem with $\sum_{i=1}^{s} l_i = |X|$. This problem will be the main objective of our work. Like the regular assignment problem, this problem can be reduced to the minimum-cost flow problem and be solved using generic flow algorithms. We augment the bipartite graph by a third layer $Z = z_1, z_2, \ldots, z_s$ of vertices, corresponding to the subsets $Y_1, Y_2, \ldots, Y_s$. For each $y \in Y_k$, we add an edge (y, z_k) with capacity 1 and cost 0 to the graph. Now, we assign demand -1 (one unit of supply) to each node $x \in X$, and demand l_k to the node $z_k \in Z$. Clearly, an optimal solution of this flow instance corresponds to an optimal assignment under the given subset capacity constraints: exactly l_k nodes of each subset Y_k will be matched.

Again, to gain deeper insight into the structure of the problem, we try to find a simpler LP relaxation than the generic minimum cost flow LP. The (canonical) linear program for the assignment problem with strict constraints is

$$\text{Minimise} \sum_{(i,j)\in E} c_{ij} x_{ij} \tag{2.31}$$

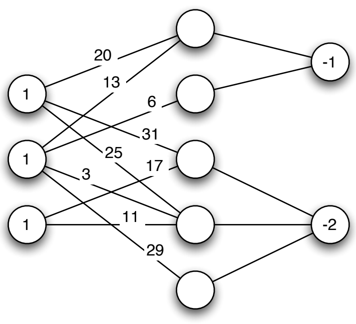


Figure 2.3: Minimum cost flow formulation of the assignment problem with strict capacity constraints

subject to

$$\sum_{j:(i,j)\in E} x_{ij} = 1 \qquad \text{for all } i \in X, \tag{2.32}$$

$$\sum_{i:(i,j)\in E} x_{ij} \leq 1 \qquad \text{for all } j \in Y, \tag{2.33}$$

$$x_{ij} \geq 0 \qquad \text{for all } (i,j) \in E, \tag{2.34}$$

$$\sum_{j\in Y_k} \sum_{i:(i,j)\in E} x_{ij} = l_k \qquad \text{for } 1 \leq k \leq s. \tag{2.35}$$

Clearly, the asymmetric assignment is a special case of this problem with $s = 1$. The corresponding dual program is

$$\text{Maximise} \sum_{i\in X} \pi(i) - \sum_{j\in Y} \pi(j) + \sum_{k\in Z} (|Y_k| - l_k)\pi(k) \tag{2.36}$$

subject to

$$\pi(i) - \pi(j) \leq c_{ij} \qquad \text{for all } (i,j) \in E, \tag{2.37}$$

$$\pi(k) \leq \pi(j) \qquad \text{for all } j \in Y_k, 1 \leq k \leq s. \tag{2.38}$$

For this problem, we cannot get rid of the additional price variables $\pi(k)$ for the additional demand nodes in Z. For the asymmetric assignment, we basically could set the price of the single additional node to 0 and shift all node prices accordingly. Now, we have several additional demand nodes, with possibly different prices. Therefore these variables have to appear in the dual program.

2.4.2 Loose Constraints

The assignment problem with loose constraints is a more general version of the problem with strict constraints. For the application in logistics, this problem is not important and therefore we will not look at it in too much detail later on; we will however state some basic properties of the problem here as well.

If we want to reduce the problem with loose capacity constraints ($\sum_{i=1}^{s} l_i \geq |X|$) to minimum cost flow, we again need an additional sink node t. For each $z_i \in Z$, we add an edge (z_i, t) with capacity $l(Y_i)$ and cost 0. Now, we assign supply 1 to each node $x \in X$, and supply $-|X|$ (or demand $|X|$) to the node t.

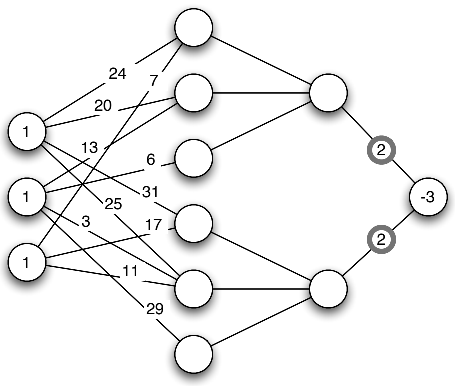


Figure 2.4: Minimum cost flow formulation of the assignment problem with loose capacity constraints

$$\text{Minimise} \sum_{(i,j)\in E} c_{ij} x_{ij} \tag{2.39}$$

subject to

$$\sum_{j:(i,j)\in E} x_{ij} = 1 \qquad \text{for all } i \in X, \tag{2.40}$$

$$\sum_{i:(i,j)\in E} x_{ij} \leq 1 \qquad \text{for all } j \in Y, \tag{2.41}$$

$$x_{ij} \geq 0 \qquad \text{for all } (i,j) \in E, \tag{2.42}$$

$$\sum_{j\in Y} \sum_{i:(i,j)\in E} x_{ij} \leq l_k \qquad \text{for } 1 \leq k \leq s. \tag{2.43}$$

The corresponding dual program is

$$\text{Maximise} \sum_{i \in X} \pi(i) - \sum_{j \in Y} \pi(j) + \sum_{k \in Z} (|Y_k| - l_k)\pi(k) \tag{2.44}$$

subject to

$$\pi(i) - \pi(j) \leq c_{ij} \qquad \text{for all } (i,j) \in E, \tag{2.45}$$

$$0 \leq \pi(k) \leq \pi(j) \qquad \text{for all } j \in Y_k, 1 \leq k \leq s. \tag{2.46}$$

However, we again have additional (complementary slackness) conditions on the reduced costs of the edges to the terminal node:

- If a subset Y_k has assigned the maximum number l_k of nodes, then the price $\pi(k)$ has to be greater or equal than $\pi(t)$.
- If a subset Y_k has assigned some nodes, but not the maximum number, the price $\pi(k)$ has to be equal to $\pi(t)$.
- If no items in a subset Y_k are assigned, the price $\pi(k)$ must be less or equal than $\pi(t)$.

This means that—similarly to the condition that assigned nodes must have higher prices than unassigned ones—full subsets must have a higher threshold between assigned and unassigned node prices than empty subsets.

Chapter 3

Basic Algorithms

In this chapter, we will present previously developed algorithms for the well-known minimum cost flow and assignment problems. We will show how the more specialised algorithms can be derived from the algorithms for the more general problems. First, we will present an algorithmic framework that all our algorithms depend on, the push/relabel method.

3.1 Push/Relabel Algorithms

The most basic algorithm, from which all algorithms presented later are derived, is the *push/relabel* algorithm. It solves the maximum flow problem, which is in some sense a special case of the minimum cost flow problem in which all edges have cost zero. Instead of minimizing the cost, we try to maximise the flow from a dedicated source to a dedicated sink node.

The basic idea of the push/relabel algorithm is, like many combinatorial optimisation algorithms, based on LP-Duality theory. The algorithm falls into the family of primal-dual algorithms: At every step the algorithm maintains a set of corresponding primal and dual variables. These variables together satisfy the complementary slackness conditions; however, the primal solution is unfeasible during the execution of the algorithm, while only the dual solution is always feasible. The algorithm now modifies the primal and dual variables accordingly, approaching a feasible primal solution. Once it has found one, we know that this solution is optimal by the complementary slackness conditions.

In the problem domain of network flow, the primal solution consists of flow values for the edges of the network, called a *pseudoflow*. As opposed to a flow, these values lie in the capacity bounds of the edges, but they do not necessarily satisfy the flow conservation constraints, i.e. the flow values do not add up correctly to the supply/demand on the nodes of the network (and therefore do not form a feasible solution). We talk of *imbalance* of flow on the nodes. If the imbalance is non-negative (there is *excess* of flow) on all nodes except the source and the sink node, the pseudoflow is called a *preflow*.

The characteristic schema of this algorithm consists of maintaining and adapting feasible dual variables, which can be regarded as distance labels for each node, and in pushing flow over the edges (i.e., increasing the value of the flow variable), depending on the complementary slackness conditions imposed by the distance labels, until the flow conservation constraints are met at each node of the network. These two basic operations, relabeling the price variables and pushing flow over edges, gave the algorithm its name. For a comprehensive discussion of the algorithm, see [AMO93].

3.2 Cost Scaling

The minimum cost flow problem, in comparison to the maximum flow problem, additionally involves costs per unit of flow for each edge. First algorithms for this problem had exponential running time; for all known strongly polynomial (polynomial in the number n of nodes and the number m of edges) time algorithms, the concept of *scaling* is vital. It was introduced in 1972 by Edmonds and Karp [EK72], who introduced the first weakly polynomial algorithm for minimum cost flow. Weakly polynomial means polynomial in n (the number of nodes), m (the number of edges), $\log U$ (the logarithm of the largest supply/demand/capacity), and $\log C$ (the logarithm of the largest cost). Scaling means that the algorithm works toward the optimal solution through a number of iterations with increasing accuracy of the numeric parameters. Since the solution found in one iteration makes it easier to proceed to the next iteration, scaling can improve the running time opposed to just using one iteration with the required accuracy. See [AMO93, pp. 68–70] for more details.

The scaling algorithms we will now take a closer look at are *cost scaling* algorithms. In these algorithms, the maximum error in regard to the cost of each edge is decreased in every iteration. We also speak of *approximate optimality*. There are other types of scaling algorithms, for example capacity scaling algorithms [AMO93, pp. 360–362], which scale the error on the capacity constraint. The algorithm of

Edmonds and Karp [EK72] used this technique. For the assignment problem, this approach does not provide any advantage, since the capacity for each edge in the bipartite graph is 1.

3.3 Approximate Optimality

In Subsection 2.1.2, we stated an equivalent condition for optimality with the complementary slackness conditions; we will use this formulation to derive our definition for approximate optimality or ϵ-optimality. This is a relaxation of complementary slackness which allows an error of ϵ at each edge for the reduced costs. The notion was first introduced by Bertsekas in his auction algorithm for the assignment problem (see Section 3.6).

Definition 2. *A (pseudo-)flow x is ϵ-optimal with respect to a collection of price values π, if for each edge (i, j) the following holds:*

$$\textit{If } c_{ij}^{\pi} > \epsilon, \textit{ then } x_{ij} = 0. \tag{3.1}$$

$$\textit{If } -\epsilon \le c_{ij}^{\pi} \le \epsilon, \textit{ then } 0 \le x_{ij} \le u_{ij}. \tag{3.2}$$

$$\textit{If } c_{ij}^{\pi} < -\epsilon, \textit{ then } x_{ij} = u_{ij}. \tag{3.3}$$

Obviously, these conditions reduce to the complementary slackness conditions if we set ϵ to 0. We are, however, not interested only in approximate solutions, but we want to achieve exact optimality. Here the integrality of costs comes into the play: it can be shown that for $\epsilon < \frac{1}{n}$, any ϵ-optimal solution is also an optimal solution, if the costs of the edges are integral [AMO93, p. 363].

3.4 Generic Cost Scaling Algorithm (Goldberg/Tarjan)

We will now present an algorithm for minimum cost circulation proposed by Goldberg and Tarjan [GT90]. The same algorithm can also be applied to the minimum cost flow problem, as described by Goldberg in [Gol97]. The algorithm ist listed as Algorithm 1.

As already mentioned before, to simplify notation of the algorithm, we assume that for each edge (i, j), there also is a back edge (j, i) with reversed cost $c_{ji} = -c_{ij}$ and flow $x_{ji} = -x_{ij}$. This only serves the purpose that we can push flow over an edge in both directions in our algorithm without having to write distinct cases for forward and reverse directions of each edge. We also assume an upper capacity bound $u_{ji} = 0$ and a lower capacity bound $l_{ji} = -u_{ij}$ for these back edges, which follows naturally from $x_{ji} = -x_{ij}$.

Initially, we set ϵ to the largest cost value C of all edges. Now any circulation is ϵ-optimal; the easiest way to obtain one is to assign flow 0 to each edge. Then we call *refine* repeatedly (Algorithm 2), where ϵ is reduced while keeping the circulation ϵ-optimal. When ϵ gets smaller than $1/n$, we have found an optimal circulation, and the algorithm terminates.

Algorithm 1 Cost scaling algorithm for minimum cost flow

```
1: $\epsilon \leftarrow C = \max_{(i,j)\in E}\{c_{ij}\}$;
2: **for all** $v \in V$ **do**
3:     $\pi(v) \leftarrow 0$;
4: **end for**
5: let $x$ be any circulation;
6: **while** $\epsilon \geq 1/n$ **do**
7:     *refine*$(\epsilon, x, \pi)$;
8: **end while**
9: return $x$;
```

Refine first divides ϵ by a constant factor α (in practice, any value between 2 and 16 provided only marginal differences in running time). Then we create an ϵ-optimal pseudoflow by saturating all edges with negative reduced costs. We also look at the reverse edges, which means that if $c^{\pi}_{ji} < 0$ (and therefore $c^{\pi}_{ij} > 0$), we set the flow x_{ji} to $u_{ji} = 0$ (and $x_{ij} = 0$).

Now we apply push/relabel operations (Algorithm 3) to this pseudoflow until no node has excess, which means that the pseudoflow is a circulation. Since push/relabel maintains ϵ-optimality of the pseudoflow, the resulting circulation is also ϵ-optimal.

On each active node, either a push or a relabel is applicable; therefore we combined both operations into one. If some edge has negative reduced costs and is not saturated, we say that this edge is *admissible*, and we push as much flow as possible over that edge.

If there is no admissible edge, we change the price label of i to the smallest value needed to make at least one edge admissible.

A push which sets the flow x_{ij} to the maximal capacity u_{ij} of the edge is called a

Algorithm 2 *refine*(ϵ, x, π);

```
1: $\epsilon \leftarrow \epsilon/\alpha$;
2: for all $(i, j) \in E$ do
3:     if $c^{\pi}_{ij} < 0$ then
4:         $x_{ij} \leftarrow u_{ij}$;
5:     end if
6: end for
7: while $\exists$ an active node $i$ (a node with excess on the flow $e(i) > 0$) do
8:     $push/relabel(i)$;
9: end while
```

Algorithm 3 *push/relabel*(i)

```
1: if $\exists$ an admissible edge $(i, j)$ (with $x_{ij} < u_{ij}$ and $c^{\pi}_{ij} < 0$) then
2:     send $\delta = \min\{e(i), u_{ij} - x_{ij}\}$ units of flow from $i$ to $j$:
       $x_{ij} \leftarrow x_{ij} + \delta$; {push}
3: else
4:     $\pi(i) \leftarrow \min_{(i,j)\in E \,\wedge\, x_{ij}<u_{ij}}\{\pi(j) + c_{ij} + \epsilon\}$; {relabel}
5: end if
```

saturating push. It can be shown that at most $\mathcal{O}(nm)$ saturating pushes, $\mathcal{O}(n^2m)$ nonsaturating pushes, and $\mathcal{O}(n^2)$ relabel operations take place during an execution of refine. The refine subroutine is called $\mathcal{O}(\log(nC))$ times in the main loop. Thus, the total running time for the cost scaling algorithm is $\mathcal{O}(n^2m\log(nC))$. The running time can be improved to $\mathcal{O}(n^3\log(nC))$ with a special rule on the selection of active nodes (*wave algorithm*). For a detailed proof we refer to [GT90].

An important observation for our application is that on a unit capacity network all pushes are saturating, yielding a running time of $\mathcal{O}(nm\log(nC))$.

3.5 Bipush (Ahuja/Orlin)

In [AOST94], Ahuja et al. propose an algorithm that solves the min-cost circulation problem on unbalanced ($n_1 \ll n_2$) bipartite networks $G = (X \cup Y, E)$ in time $\mathcal{O}((n_1m + n_1^3)\log(n_1C))$, which is only dependent on the number $n_1 = |X|$ of nodes in the smaller subset and not on the total number of nodes.

This algorithm is basically the cost-scaling algorithm by Goldberg and Tarjan with additional rules about the choice of active nodes for push/relabel in the refine routine. The refinement loop (Algorithm 4) is divided into two phases: first, push/relabel is applied to active nodes on the right hand side Y only, until all nodes in Y have become inactive. Now all excess is on the nodes on the left hand side X. In the second phase, *bipush/relabel* (Algorithm 5) is applied to active nodes on the left hand side X. This is a modified version of *push/relabel* which always keeps excess on the same subset of nodes by applying two subsequent pushes if a node in Y would become active to push back the excess to X. That way, the algorithm first brings the excess of the pseudoflow to the smaller subset of the nodes and then always keeps the excess on that subset.

Since the longest cycle contains at most $2n_1$ vertices, the refinement step only has to be repeated until $\epsilon < \frac{1}{2n_1}$.

Algorithm 4 *bipartite-refine*(ϵ, x, π);

```
1: $\epsilon \leftarrow \epsilon/\alpha$;
2: for all $(i,j) \in E$ do
3:   if $c^{\pi}_{ij} < 0$ then
4:     $x_{ij} \leftarrow u_{ij}$;
5:   end if
6: end for
7: while $\exists$ an active node $j \in Y$ do
8:   $push/relabel(j)$;
9: end while
10: while $\exists$ an active node $i \in X$ do
11:   $bipush/relabel(i)$;
12: end while
```

3.6 Auction Algorithm

The auction algorithm, first proposed by Bertsekas [Ber81], is an algorithm for solving the assignment problem. It works in a very similar manner to the cost-scaling push/relabel algorithm applied to the assignment problem. In fact, in a paper by Goldberg and Kennedy [GK95], the auction algorithm is described as a heuristic applied to the push/relabel algorithm to improve practical performance. Bertsekas also showed the equivalence of the auction algorithm and the push/relabel algorithm [Ber94], meaning that each method can be derived from the other. The algorithm achieves the same running time of $\mathcal{O}(nm\log(nC))$ as the push/relabel algorithm. Ahuja and Orlin [OA92] and Goldberg and Kennedy [GK95] showed later

Algorithm 5 *bipush/relabel*(i)

```
1: if ∃ an admissible edge $(i,j)$ (with $x_{ij} < u_{ij}$ and $c^{\pi}_{ij} < 0$) then
2:   if $e(j) < 0$ then
3:     send $\delta = \min(e(i), e(j), u_{ij} - x_{ij})$ units of flow from $i$ to $j$; {push}
4:   else if ∃ an admissible edge $(j,k)$ then
5:     send $\delta = \min(e(i), u_{ij} - x_{ij}, u_{jk} - x_{jk})$ units of along the path $i - j - k$; {push}
6:   else
7:     $\pi(j) \leftarrow \min_{(j,k)\in E \wedge x_{jk}<u_{jk}}\{\pi(k) + c_{jk} + \epsilon\}$; {relabel}
8:   end if
9: else
10:  $\pi(i) \leftarrow \min_{(i,j)\in E \wedge x_{ij}<u_{ij}}\{\pi(j) + c_{ij} + \epsilon\}$; {relabel}
11: end if
```

that the running time can be improved to $\mathcal{O}(\sqrt{n}m\log(nC))$, although this is merely a theoretical improvement which has not shown any advantage in implementations. A thorough average case analysis of the auction algorithm has not been achieved yet. Bertsekas states however [BCT93] that computational experiments with uniformly distributed edge costs lead to the assumption that the average running time could be around $\mathcal{O}(m\log(n))$ or $\mathcal{O}(m\log(n)\log(nC))$.

The algorithm, as described by Bertsekas, solves the maximum cost assignment problem. For this problem, the analogy with a real auction is more evident than for the minimum cost assignment problem. However, for the latter problem it allows an intuitive explanation as well, so we stick to this notion here. The maximisation problem is easy to transform into a minimum cost assignment problem by negating the costs.

The auction algorithm can be described as a real auction: the elements in X (persons) make bids for their most beneficial elements in Y (objects), raising the prices of these elements. The most beneficial object j for a person i is the object with the least sum of cost c_{ij} and price $\pi(j)$. An object is awarded to the person with the best bid.

Again, we can ensure optimality of a solution by complementay slackness conditions. Intuitively, the complementary slackness conditions can be stated as follows: if each person is awarded the object which is most beneficial to him (with respect to prices for the objects, the dual variables), everyone is happy and we know that the cost of the assignment is minimal.

To ensure that the algorithm does not cycle, prices have to increase by a minimum amount of ϵ, like in a real auction. This means that if for a person two (or more) objects have the same benefit, the price of one object has to increase (by ϵ). Note that after that, the other object is more profitable by an amount of ϵ, because its price did not increase.

Bertsekas proposed the use of approximated complementay slackness conditions (ϵ-optimality): instead of requiring that every person is awarded the best object, for each assigned person the profit of the awarded object has to be at most ϵ less than the profit of any other object:

$$c_{ij} + \pi(j) \geq \max_{k:(i,k)\in E}\{c_{ik} + \pi(k)\} - \epsilon \qquad \forall (i,j) \in E \tag{3.4}$$

This means that for n persons, the total cost for the assignment is at most $n \cdot \epsilon$ more than the cost of an optimal assignment. If the costs are integral, the total cost of any assignment is integral as well. This means that for $\epsilon < \frac{1}{n}$, the assignment found by the auction is obviously optimal: the total cost is less than 1 more than the optimum and it is integral, hence it must be equal to the optimum.

Bertsekas also was the first to propose the use of ϵ-scaling to improve the running time of the algorithm for certain difficult instances of the problem, where a group of persons competes for a smaller number of objects. If ϵ is too small, this constellation can lead to "price wars", where a large number of small price increments on the respective objects is required before some of the persons find another object which is more profitable than the objects competed for. By executing the algorithm several times with decreasing values for ϵ, starting with the highest cost C, these price wars can be avoided.

A noteworthy property of the auction algorithm is that it is highly parallelizable. Several persons can bid for several objects at a time (bidding phase), and the objects that have received bids are then assigned to the highest bidders and the prices are increased accordingly (assignment phase). The focus of our project was on serial algorithms, so we only looked at the version of the algorithm where just one unassigned person at a time can bid. This version is called *Gauss-Seidel version* by Bertsekas [Ber88] because of its resemblance to the Gauss-Seidel method for solving linear equation systems.

3.7 Auction as a Flow Algorithm (double-push)

We will now describe how this algorithm fits into the minimum cost flow context. We look at the weighted bipartite graph $G = (X \cup Y, E)$ again; the elements in X correspond to the persons, while the elements in Y correspond to the objects in the auction algorithm.

The prices of the objects are the same as the prices of the nodes in Y: these are the dual variables for the LP. Note that in the auction algorithm, as opposed to the flow algorithm, there are only prices for the objects, but not for the persons, whereas in the flow algorithm every node has a price. The reason is that the prices for persons are somewhat redundant: for matched persons, they depend on the price of the matched object and the cost of this pairing; for unmatched persons, they are not needed for finding the most profitable object. Consequently, there exists a simplified formulation of the dual linear program without the price variables for persons, as we have seen in Section 2.2. In our implementation, only the prices for nodes in Y are used; some variants of the algorithm however use prices for the nodes in X too, as we will see later.

Anyway, we can set the price value $\pi(x)$ of a person x assigned to an object y to $\pi(y) + c_{xy} - \epsilon$, so that the reduced cost c^{π}_{xy} of the respective edge is equal to ϵ and the ϵ-optimality conditions hold for the edge (x, y), while for all other outgoing edges of x, even the strict complementary slackness conditions hold. This allows us to use a stricter version of the ϵ-optimality conditions than the one we presented in Definition 2. While that definition allowed an error of ϵ on all edges (compared to strict complementary slackness conditions), in the so-called *asymmetric* definition of ϵ-optimality, on the unmatched edges strict complementary slackness conditions are required to hold. This has the effect that the complementary slackness conditions are only violated on the $\mathcal{O}(n)$ matched edges instead of up to $\mathcal{O}(m)$ edges in the symmetric definition.

Why is this important? We want to apply ϵ-scaling, which means that we run the auction algorithm several times with decreasing ϵ. While ϵ-optimality may hold for an edge in one iteration, it may be violated in the following iteration. For that reason, in the generic push/relabel algorithm we had to initialise the pseudoflow in each iteration of refine to maintain ϵ-optimality with regard to the new ϵ. We can now take advantage of the fact that, according to the asymmetric ϵ-optimality, all unmatched edges already satisfy the complementary slackness conditions. We can therefore simply start with an empty flow; all previously matched edges (x, y)

have reduced cost $c^{\pi}_{xy} = \epsilon$; if these edges are unmatched now, the complementary slackness conditions hold for these as well.

A bidding step in the auction algorithm is the equivalent to a combined push and relabel operation, called *double-push* in [GK95]. This operation is similar to the bipush operation proposed in [AOST94], which was mentioned in Section 3.5. There are, however, some small differences between the two operations. When a person x bids for an object y, flow is pushed along the edge (x, y) in the flow algorithm. After that, the node x is relabeled (optionally, as already mentioned) to the highest possible value with regard to the ϵ-optimality conditions. If y has already been assigned to another person x', the node y now has positive excess and flow is pushed back to x'. Finally, the node y is relabeled, again to the highest possible value. The algorithm is listed as Algorithm 6.

Algorithm 6 Cost scaling (auction) algorithm for assignment

```
1: $\epsilon \leftarrow C = \max_{(i,j)\in E}\{c_{ij}\}$;
2: for all $v \in V$ do
3:     $\pi(v) \leftarrow 0$;
4: end for
5: while $\epsilon \geq 1/n$ do
6:     refine($\epsilon, x, \pi$);
7: end while
8: return $x$;
```

Algorithm 7 *refine-auction*(ϵ, x, π);

```
1: $\epsilon \leftarrow \epsilon/\alpha$;
2: for all $(i, j) \in E$ do
3:     $x_{ij} \leftarrow 0$;
4: end for
5: while $\exists$ an unassigned node $i \in X$ do
6:     double-push($i$);
7: end while
```

The auction algorithm by [Ber88] and the double-push method of [GK95] are essentially equivalent, however some of the definitions in the papers differ slightly. In [Ber88], the price values for the persons are negated compared to our definitions; in [GK95], all price values are negative. The definitions we gave here follow the book of Ahuja et al. [AMO93].

Another difference can be found in the definition of ϵ-optimality. In [Ber88], the

Algorithm 8 *double-push*(i)

```
1: let $(i, j)$ and $(i, k)$ be edges with smallest and second-smallest reduced costs;
2: $x_{ij} \leftarrow 1$; {push}
3: if $\exists$ another edge $(h, j) \neq (i, j)$ with $x_{hj} = 1$ then
4:     $x_{hj} \leftarrow 0$; {push}
5: end if
6: $\pi(j) \leftarrow \pi(k) + c_{ik} - c_{ij} + \epsilon$; {relabel}
```

prices for nodes in X are higher by an amont of ϵ compared to our definitions; this means that the complementary slackness conditions hold for the unmatched edges instead of the matched edges. The latter definition is not convenient in the context of minimum cost flows, for example for showing the above property; however it is obviously equivalent if the price values for persons are not used in the algorithm anyways. Moreover, motivated by the notation used in the proofs given in the paper, 2ϵ is used instead of ϵ everywhere in the algorithm. It is again easy to see that these different definitions are equivalent.

Compared to the generic scaling push/relabel algorithm, the auction algorithm makes use of the special structure of the assignment problem. It therefore differs from this algorithm in several ways:

- It uses a different definition of ϵ-optimality, which allows for an easier initialisation step in each refinement iteration.
- Similarly to the bipush operation from Section 3.5, excess is always kept at the left hand side of the graph by performing a push away from a right hand side node as soon as it has positive excess.
- Instead of relabeling nodes as little as possible before a push, nodes are relabeled as much as possible after a push.

Kennedy [GK95], who presents the algorithm as a network flow algorithm derived from the push/relabel algorithm, claims that the different rules for relabeling (which he calls "aggressive relabeling") provide a better running time in computational experiments and even make their global relabeling technique—which improves the theoretical performance of the algorithm to $\mathcal{O}(\sqrt{n}m\log(nC))$—unnecessary in practice.

3.8 Forward/Reverse auction

In [BCT93], Bertsekas, Castañon, and Tsaknakis proposed a modified version of the algorithm, in which not only the persons bid for objects by raising their prices, but also objects compete for persons by lowering their prices. The goal is to avoid the "price wars" under certain circumstances and therefore make it possible to use the algorithm without ϵ-scaling in many cases. Another important aspect is that for asymmetric assignment problems, the original auction algorithm can only be used without ϵ-scaling, since the initialisation step may not produce a valid (ϵ-optimal) configuration. Using reverse auction, this restriction can be circumvented. The modified refinement step is listed as Algorithm 9.

A reverse auction step works basically the same way as a forward auction step, but with reversed signs. Note that for the forward/reverse algorithm, prices are needed for all nodes in the network (persons and objects). The price values of persons can be interpreted as the maximal amount a person is willing to spend for an object.

In a reverse bidding step (Algorithm 10), an unassigned object lowers its price as much as possible without violating ϵ-optimality constraints to attract an unassigned person or to find a person who is willing to exchange his currently held object. Accordingly, the amount of money that person is willing to spend (the price label) is decreased.

Algorithm 9 *refine-forward-reverse*(ϵ, x, π);

```
1: $\epsilon \leftarrow \epsilon/\alpha$;
2: for all $(i, j) \in E$ do
3:     $x_{ij} \leftarrow 0$;
4: end for
5: repeat
6:     while $\exists$ an unassigned node $i \in X$ and
       the number of assigned nodes did not increase do
7:         double-push$(i)$;
8:     end while
9:     while $\exists$ an unassigned node $j \in Y$ and
       the number of assigned nodes did not increase do
10:        reverse-double-push$(j)$;
11:    end while
12: until all nodes are assigned
```

Algorithm 10 *reverse-double-push*(i)

```
1: let $(i,j)$ and $(i,k)$ be edges with smallest and second-smallest reduced costs;
2: $x_{ij} \leftarrow 1$; {push}
3: if $\exists$ another edge $(h,j) \neq (i,j)$ with $x_{hj} = 1$ then
4:     $x_{hj} \leftarrow 0$; {push}
5: end if
6: $\pi(j) \leftarrow \pi(k) + c_{ik} - c_{ij} + \epsilon$; {relabel}
```

Unfortunately, the authors do not provide a proof that forward/reverse auction does not increase the theoretical time bound for the auction algorithm. However, in practice the running time of the forward/reverse algorithm compares favourably to the running time of the purely forward algorithm.

3.9 Other Algorithms for Minimum Cost Flow and Assignment

There is another major class of algorithms for the minimum cost flow and assignment problems: the *shortest augmenting path* algorithms. These algorithms are based on searching for shortest paths in the *residual network* (the network consisting of unsaturated edges). A detailed description can again be found in [AMO93]. We did not investigate the use of these algorithms, since they seem to less efficient than push/relabel or auction algorithms in general. The most competitive algorithm for the assignment problem was developed by Jonker and Volgenant [JV87]. In [Cas93], Castañon compares this algorithm with an implementation of the forward/reverse auction algorithm, with favourable results for the latter.

It is worth noting, however, that shortest augmenting path algorithms tackle certain configurations more efficiently than push/relabel or auction algorithms. Therefore, Ahuja and Orlin [OA92] propose a hybrid algorithm, which in each scaling iteration first starts with an auction and then, if no complete assignment has been found after a certain amount of bidding steps, switches to a shortest augmenting path method. This hybrid algorithm matches the best known running time bound of $\mathcal{O}(\sqrt{n}m\log(nC))$.

One of the oldest algorithms for the assignment problem, the Hungarian method [Kuh55], works with shortest augmenting paths as well. In its original version, it was proposed for a complete bipartite graph, but it can also be applied to sparse graphs efficiently (see [AMO93]). It still has the best strongly polynomial time bound for the assignment problem ($\mathcal{O}(nm + n^2 \log n)$ if using Dijkstra's algorithm with Fibonacci heaps for finding shortest paths). In practical usage, however, it is not competitive compared to recent algorithms.

Chapter 4

Asymmetric Assignment

In the preceding chapter we have seen a basic algorithmic technique for the assignment problem with the auction algorithm. We will now focus on the asymmetric assignment problem, where $|X| < |Y|$, and try to modify the auction algorithm so that we can use it for this problem as well.

4.1 Auction Algorithm for Asymmetric Assignment

Unfortunately, the auction algorithm cannot easily be adopted to the asymmetric assignment problem. The main difficulty is that the correctness of this algorithm depends on the fact that before each iteration of the auction, a valid (ϵ-optimal) assignment of the variables has to be given. An empty assignment, however, as we have used for the symmetric assignment problem, is not always feasible for the asymmetric assignment problem.

Remember that the minimum cost flow version of the asymmetric assignment problem has an additional terminal node, t, to which all nodes in Y are connected by an edge with cost 0 and capacity 1. The symmetric assignment problem can also be formulated that way (with one single source node and one sink node); however the edges from the source and to the sink have always to be matched. For asymmetric assignment, on the other hand, part of the problem is to determine which edges have to be matched. Let us require ϵ-optimality for the edges of the bipartite graph, and strict optimality for these auxiliary zero cost edges. In terms of the price function, for an unmatched right hand side node y we require $\pi(y) \leq \lambda$, and for a matched node we need $\pi(y) \geq \lambda$, where $\lambda = \pi(t)$ is the price of the additional node t. In other words, the prices of the matched nodes have to be less or equal to the prices of the unmatched nodes. Running the algorithm unchanged does not ensure that this requirement is met: if after an iteration the price of an object is too high, it might be left unassigned in the next iteration, even if it would be desirable for some person. In the symmetric case, every object gets assigned, so this situation does not occur.

One possible way to avoid this problem is to not use ϵ-scaling, which can, however, lead to "price wars" (long sequences of only small price increments for a small number of objects competed for by a slightly larger group of persons), as pointed out by Bertsekas [BCT93]. Therefore, we look for further modifications to the auction algorithm which again allow the use of ϵ-scaling.

4.2 Virtual Nodes

A straightforward and commonly used way to reduce the asymmetric assignment problem to the symmetric problem is to add an additional $|Y| - |X|$ nodes (which we will call *virtual nodes*) to the left hand side, each of which are connected to every node on the right hand side with zero cost. Especially for sparse bipartite graphs, this means that quite a lot of additional edges are added to the graph; however, since all edges going out of a virtual node have zero cost, implementation of the double-push operation can be simplified substantially for these nodes. Instead of computing reduced costs, we only need to regard the price values of the right hand side nodes. We push flow over the edge to the node with the lowest price value. To find this node, we can use a heap for the right hand side nodes with the price values as keys. The edge on which flow is going to be pushed is always the edge to the node with the lowest price, which is found using the heap. Note that these zero cost edges do not have to be saved in explicit form.

4.3 Using a Heap for Prices

As already mentioned, the prices of the matched right hand side nodes need to be greater or equal to the prices of the unmatched right hand side nodes. Basically, the virtual nodes approach achieves exactly this condition: Flow from a virtual node is always pushed along an edge with lowest reduced cost. This is the edge to the right hand side node with the lowest price. Therefore, the unmatched nodes (or rather, the nodes matched with virtual nodes) are the nodes with the lowest prices after a refine is completed.

We will now present a more direct approach for satisfying this condition. Every time we match and relabel a right hand side node, we check if the condition is violated by some node, i.e. if there is some unmatched node with a higher price than a matched node. If this is the case, we push back flow on the respective edge again. As already mentioned, pushing back flow is always possible without violating ϵ-optimality.

To check the mentioned condition, we maintain a heap with the $n_1 = |X|$ right hand side nodes with highest prices. Every time we relabel a node that is not already in the heap, we insert it and then remove the node with the lowest price in the heap. If this node is matched, we push flow back over the matched edge.

The modified version of *double-push* is listed here as Algorithm 11 and simply replaces the double-push in the refinement step (Algorithm 7) of the auction algorithm.

Algorithm 11 *double-push-heap*(i)

```
1: let $(i, j)$ and $(i, k)$ be edges with smallest and second-smallest reduced costs;
2: $x_{ij} \leftarrow 1$; {push}
3: if node $j$ is not in the heap $\mathcal{H}$ then
4:    add $j$ to the heap $\mathcal{H}$ with key $\pi(j)$
5:    if more than $|X|$ elements are in the heap $\mathcal{H}$ then
6:       $g$ = remove-min$(\mathcal{H})$
7:       if $g$ is assigned to a node $f \in X$ then
8:          $x_{fg} \leftarrow 0$; {push}
9:       end if
10:   end if
11: end if
12: if $\exists$ another edge $(h, j) \neq (i, j)$ with $x_{hj} = 1$ then
13:    $x_{hj} \leftarrow 0$; {push}
14: end if
15: $\pi(j) \leftarrow \pi(k) + c_{ik} - c_{ij} + \epsilon$; {relabel}
```

Using a binary heap, we need $\mathcal{O}(\log(n_1))$ time to relabel a node, while accessing the smallest label in the heap takes $\mathcal{O}(1)$. Consequently, the running time bound increases to $\mathcal{O}(nm\log(nC)\log(n_1))$. However, for the sizes of the instances we tested, the additional factor did not affect the performance of the algorithm compared to other algorithms for solving the asymmetric assignment problem.

4.4 Relabeling Unmatched Nodes

An additional heuristic has been developed to improve the running time of the algorithm. It follows the same strategy as the "aggressive relabeling" in the double-push operation; namely, every node is relabeled to the highest possible value. After a refine iteration, some of the unmatched nodes may not have been relabeled at all, whereas all of the matched nodes have been relabeled at least once. In following iterations of refine, this can lead to flow being pushed to a previously unmatched node and afterwards pushed back again several times until the price of the unmatched node has increased enough, even if this node stays unmatched during the refine procedure afterwards.

To avoid these unnecessary steps, after an iteration of refine has finished, we try to increase the price values of unmatched nodes as much as possible, namely to the lowest price of the matched nodes. The respective value is accessible in constant time through the heap.

Experimental comparisons with an implementation not using this heuristic show a clear improvement in running time and number of double-push operations, especially for highly asymmetric instances ($n_1 \ll n_2$). See also the computational results in Chapter 7.

4.5 Reverse Auction

Another approach is to use reverse auction (see Section 3.8) to make sure that all unmatched nodes have higher prices than the matched nodes. This technique was presented by Bertsekas et al. in [BCT93]. This algorithm works in two phases. In the first phase, it finds a feasible assignment using forward auction. It then run reverse auction in the second phase, until the prices of the unmatched nodes have decreased below the threshold λ, which is the smallest price of the assigned nodes. Since node prices are only relative values, changing all values by the same amount gives essentially the same configuration. This means that we can as well set the threshold λ between unmatched and matched nodes to 0, as we have already explained in Section 2.3.

An improved version (Algorithm 12) was proposed in [BC92]. It incorporates the combined forward/reverse auction algorithm described in Section 3.8 instead of the forward auction algorithm. In contrast to Algorithm 9, reverse auction steps (reverse

Algorithm 12 *refine-forward-reverse-asymmetric*(ϵ, x, π);

```
1: $\epsilon \leftarrow \epsilon/\alpha$;
2: **for all** $(i,j) \in E$ **do**
3:    $x_{ij} \leftarrow 0$;
4: **end for**
5: **repeat**
6:    **while** $\exists$ an unassigned node $i \in X$ and
      the number of assigned nodes did not increase **do**
7:       *double-push*$(i)$;
8:    **end while**
9:    **while** $\exists$ an unassigned node $j \in Y$ with price $\pi(j) > 0$ and
      the number of assigned nodes did not increase **do**
10:      *reverse-double-push*$(j)$;
11:   **end while**
12: **until** all nodes in $X$ are assigned
13: **while** $\exists$ an unassigned node $j \in Y$ with price $\pi(j) > 0$ **do**
14:    *reverse-double-push*$(j)$;
15: **end while**
```

Algorithm 13 *reverse-double-push*(i)

```
1: let $(i,j)$ and $(k,j)$ be edges with smallest and second-smallest reduced costs;
2: **if** $c_{ij} - \pi(i) < 0$ **then**
3:    $\pi(j) \leftarrow 0$;
4: **else**
5:    $\pi(j) \leftarrow \pi(k) + c_{kj}$; {relabel}
6:    $\pi(i) \leftarrow \pi(i) + c_{ij} - \epsilon$; {relabel}
7:    $x_{ij} \leftarrow 1$; {push}
8:    **if** $\exists$ another edge $(h,j) \neq (i,j)$ with $x_{hj} = 1$ **then**
9:       $x_{hj} \leftarrow 0$; {push}
10:   **end if**
11: **end if**
```

double pushes) are only applied to nodes in Y with price $\pi(j) > 0$. Moreover, after a full assignment is found, the prices of the leftover unassigned nodes with $\pi(j) > 0$ have to be fixed with reverse auction steps, possibly modifying the assignment again.

Since the prices are required to be nonnegative, a reverse bidding step does not necessarily result in the bidding object being assigned to a person: even with a price of 0, the object might still be unprofitable for all persons.

This algorithm relates closely to the LP formulation of the problem that we gave in Section 2.3: node prices are required to be greater or equal than 0, and there is no explicit price variable for an additional sink node (which is equivalent to setting this price to 0).

There is another version of the algorithm which has an explicit value for the sink node or threshold between prices of assigned and unassigned nodes. In this version, prices are also allowed to be negative. The threshold λ is the n_1th-greatest price of the right hand side nodes Y. This requires a search over all n_2 nodes if the threshold changes. Computational comparisons [BC92] show that both variants have roughly the same running time on the tested instances.

Chapter 5

Constrained Assignment

In this chapter, we will describe various approaches to the constrained assignment problem introduced in Section 2.4. Due to the differences in structure of the two variants of the problem, the assignment problem with strict and loose capacity constraints, we will discuss both problems separately.

5.1 Using Minimum Cost Flow

The most obvious approach to solving the constrained assignment problem, as described in Section 2.4, is to reduce the problem to the minimum cost flow problem (Figure 5.1 on the following page). Applying a generic minimum cost flow algorithm on this network gives us a solution from which we can derive an optimal assignment.

As we have seen in the case of the assignment problem, the special structure of the input network can help to find substantially faster algorithms. Therefore we try to find faster algorithms than the generic flow algorithm for the constrained assignment problem.

5.2 Using Bipartite Network Circulation

In Section 3.5, we mentioned a variant of the cost scaling push/relabel algorithm that solves the min-cost circulation problem on unbalanced ($n_1 \ll n_2$) bipartite networks in time $\mathcal{O}((n_1 m + n_1^3) \log(n_1 C))$, which is only dependent on the number of edges and the number of nodes in the smaller subset, and not on the total number

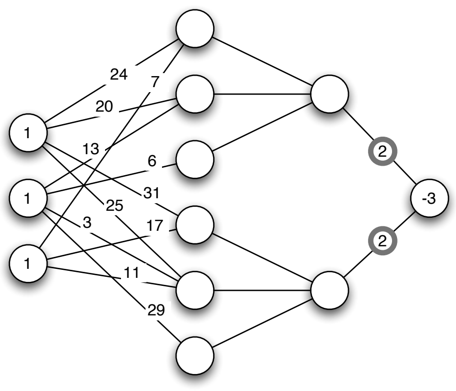


Figure 5.1: Minimum cost flow formulation of the assignment problem with capacity constraints

of nodes.

We can express the constrained assignment problem as a min-cost circulation problem on a bipartite network and thus make use of this algorithm. We start with the network for the minimum cost flow algorithm (Figure 5.1). For each node $x \in X$, we add an additional edge (t, x) to the network with capacity 1 and cost $-C$. The resulting network is bipartite, and a minimum cost circulation on this network corresponds to an optimal solution of the constrained assignment problem. The weight $-C$ is required to avoid that the empty circulation is optimal; alternatively, we could set the lower capacity bound of these edges to 1.

Since we have more nodes in Y than in X, this algorithm guarantees a better performance than the generic cost-scaling algorithm with $\mathcal{O}(n^3 \log(nC))$. However, we will not gain a significant advantage if we assume that n_2 is larger by a constant factor than n_1, which is the case in the warehousing application mentioned in the introduction.

5.3 Adapting the Auction Algorithm for Constrained Assignment

Since the auction algorithm provided a better running time than the push/relabel algorithm for minimum cost flow, we want to use this algorithm to find a solution for the constrained assignment problem. We transform the underlying asymmetric

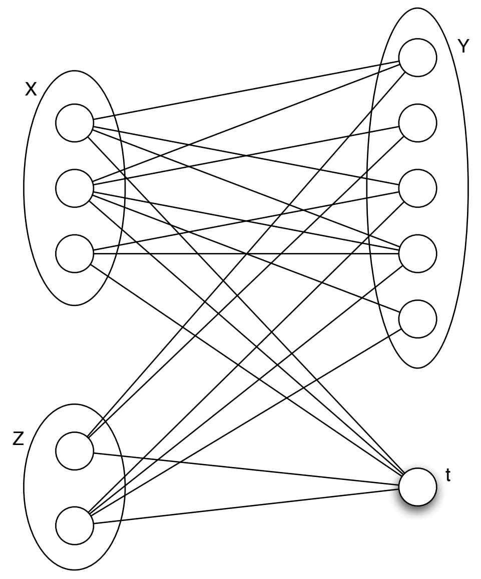


Figure 5.2: A bipartite network for solving the capacity-constrained assignment problem

assignment problem to a symmetric one by adding additional virtual nodes X'. We count the number of nodes in each subset Y_i assigned to a node in X (not a virtual node in X'). Now we have to make sure that no more than l_i nodes in Y_i are assigned.

One obvious approach to the problem is to simply push back flow along an edge if a capacity constraint is violated. Pushing back flow along an edge is always allowed, since the reduced cost of a matched edge is ϵ. Edges with reduced costs $[0, \epsilon]$ can be both matched and unmachted edges according to ϵ-optimality. Since we want the unmatched nodes to have the lowest prices, we push back at the node with the lowest price. That way, we keep a ϵ-optimal pseudoflow after each double-push operation.

Unfortunately, we cannot guarantee that this algorithm will terminate. Tests of an implementation of the described algorithm showed quickly that there are instances of the constrained assignment problem where the algorithm will cycle.

The reason is that an optimal solution to this problem does not require that *all* matched nodes have higher or equal prices than *all* unmatched nodes. We will now have a look at the problem with strict capacity constraints ($\sum_{i=1}^{s} l_i = |X|$).

5.4 Strict Constraints

If we consider the flow formulation of the problem, we can easily observe that all the edges (z_i, t) to the terminal node t with capacity l_i (the additional edges representing the capacity constraints) have to be saturated in a feasible solution of the problem, since the capacity constraints are strict. This means that the complementary slackness conditions require the price value for the terminal node $\pi(t)$ simply to be lower or equal to each of the values $\pi(z_i)$. As a consequence, we do not need to take care of these edges; we can assume that they are saturated all the time during execution of the algorithm.

To find an optimal solution, we only have to make sure that for each i, the prices of the matched nodes in Y_i are higher or lower than tho of the unmatched nodes in Y_i. This can be achieved in a way analogous to the asymmetric assignment, by using a heap for each subset Y_i in which the l_i nodes with highest prices are stored. Every time a node is relabeled which is not in the heap, it is added to the heap and the node with lowest price is removed and unmatched.

This means that the condition that matched nodes have higher prices than unmatched ones might be violated while the assignment is not complete; however, when the algorithm has found a complete assignment, in each subset no unassigned node can have a lower price than the minimum key in the heap. Furthermore, all nodes in the heap are assigned, since each node that gets assigned is also put into the heap, and each node that gets unassigned is removed from the heap. Thus, no unassigned node can stay in the heap at the end of a refinement step.

Note that the price threshold between matched and unmatched nodes may be different for each subset Y_i. In fact, if the optimum for the unconstrained problem is smaller than the optimum for the constrained problem, there will be some unmatched node with a higher price value than some matched node in another subset when the algorithm has found a solution.

The modified double-push operation is listed here as 14; it simply replaces the double-push in the refinement step (Algorithm 7) of the auction algorithm.

There is, however, one caveat for this algorithms regarding the heuristic presented in Section 4.4. As we will see in Chapter 7, this heuristic, which is very robust and useful for the asymmetric assignment algorithm, does not always work well for the constrained assignment algorithm. For a larger number of subsets Y_i, the algorithm tends to take considerably more time on some instances; the higher the number of

Algorithm 14 *double-push-constrained*(i)

```
1: let $(i,j)$ and $(i,k)$ be edges with smallest and second-smallest reduced costs;
2: $x_{ij} \leftarrow 1$; {push}
3: let $Y_k$ be the subset containing $j$ with capacity $l_k$
4: if node $j$ is not in the heap $\mathcal{H}_k$ for $Y_k$ then
5:    add $j$ to the heap $\mathcal{H}_k$ with key $\pi(j)$
6:    if more than $l_k$ elements are in the heap $\mathcal{H}_k$ then
7:       $g = \textit{remove-min}(\mathcal{H}_k)$
8:       if $g$ is assigned to a node $f \in X$ then
9:          $x_{fg} \leftarrow 0$; {push}
10:      end if
11:   end if
12: end if
13: if $\exists$ another edge $(h,j) \neq (i,j)$ with $x_{hj} = 1$ then
14:    $x_{hj} \leftarrow 0$; {push}
15: end if
16: $\pi(j) \leftarrow \pi(k) + c_{ik} - c_{ij} + \epsilon$; {relabel}
```

subsets, the more often significantly longer running times occurr. This is caused by the relabeling to the maximal possible value, which makes the unassigned nodes undistinguishable from the assigned node with the lowest price.

A viable solution to this problem is to relabel the unassigned nodes to a value ϵ lower than the threshold (if their price does not decrease that way). Since this relabeling happens between two refinement steps, we can use the new ϵ here and gain better performance than with using the old ϵ. Using this approach, the performance of the algorithm is roughly the same as the performance of the auction algorithm with one heap for the unconstrained assignment.

5.5 Loose Constraints

If the capacity constraints for the subset are not strict ($\sum_{i=1}^{s} l_k > |X|$), solving the problem is more difficult: we cannot predict the exact number of matched nodes in each subset Y_k. Consequently, it does not suffice to ensure that all matched nodes in a subset Y_k have a price greater or equal to the l_k-th greatest price of all nodes in Y_k, as we have done before for the strict constraints.

One possible way to solve the problem is to always keep the prices of unassigned nodes in Y_k below λ_k and the prices of assigned nodes above or equal to λ_k, and to keep only assigned nodes in the heap. However, this requires finding the unassigned node with highest cost as well: if we want to assign node i to node j, we need to check if the price of the newly assigned node j is lower than the new highest price of the remaining unassigned nodes. This can again be done by searching in all n_2 nodes of Y or by organising the unassigned nodes in a heap.

There is another difficulty which did not arise in the previous case. As described in Section 2.4, we have additional conditions on the thresholds λ_k: if the capacity of a subset Y_k is fully used, the threshold must be higher or equal than that of a subset which has spare capacity. Conversely, if no node in a subset is matched, the threshold must be less or equal to the threshold of a subset with matched nodes. For all subsets with some matched nodes and spare capacity, the threshold must be equal. If we find out that at some point these conditions are violated, we can reestablish them by unassigning nodes in a subset, which increases the threshold.

As we have seen, a specialized algorithm for loose constraints cannot make much use of a simpler structure of the problem as the generic minimum cost flow problem, as it is the case for strict constraints. As we mentioned initially, however, for our practical application this problem is not important. To achieve an equal distribution of products to the warehouses, we can specify the exact number of products that have to be placed in every warehouse.

Chapter 6

Implementation

We are now ready to give an overview of the implementation issues for the algorithms described earlier. All implementations were done in C++. As a starting point, we were able to use the implementation of an auction algorithm for asymmetric assignment using virtual nodes, which was created as part of Werner Unterhofer's diploma thesis [Unt03]. The algorithms that we implemented share as much code and structure as possible to allow for a better comparison; even if there already was another implementation available, as it was the case for the forward/reverse auction algorithm, we choose to reimplement the algorithm to cancel out several differences in the implementation, such as use of Fortran instead of C++, integers instead of floating-point values, or undocumented heuristics that were used in the third-party code.

We therefore start with some aspects which apply to all implementations, before continuing with specific issues of each algorithm.

6.1 Common Aspects

Although the described algorithms require integer costs, and therefore obviously integral optimal dual solutions exist as well, the use of ϵ-optimality conditions requires fractional price values. One possible way to do the calculation with integers anyway is to multiply the costs and the starting ϵ by n_1 and run the algorithm until ϵ reaches 1, always rounding ϵ to an integral value in each scaling step. That way, the prices have integral values as well. This approach was used in Bersekas' Fortran implementation of the auction algorithm.

In our implementations, we used floating-point values with double precision (IEEE 754). On today's computer architectures, the performance impact for using floating point arithmetic as opposed to using integer arithmetic is negligible. It should be noted, however, that rounding errors can be prohibitive to the use of too large costs: not only can the computed result deviate from the optimum solution, but if the value of ϵ is too small compared to the price values, the bidding steps may not increase the prices due to rounding, and the algorithm will not terminate.

Regarding the choice of the scaling factor α, Goldberg and Kennedy [GK95] state that running times do not vary by a factor of more than 2 for any value between 4 and 40, with some values preferable for certain problem classes and some for others. We choose a fixed value of $\alpha = 10$ for our algorithms.

Most of the algorithms (apart from the forward/reverse algorithm) do not specify in which way the active nodes are chosen. Two strategies have been compared in [Unt03]: storing active nodes in a queue (FIFO) or in a stack (LIFO). The results showed that none of both strategies obtain a significantly better running time. The LIFO data structure obtained marginally better running time on spares graphs and is also less memory-consuming, therefore it was used in our implementation.

The left hand side and right hand side nodes are stored in two separate arrays; the edges are saved in adjacency lists of the tail nodes of the edges (the nodes in $|X|$). Since nodes are accessed by their index in the node array, we can access them in constant time and store additional information for nodes needed by the algorithm in arrays of the same size instead of the data structure for the nodes themselves. Direct access to the weight of an edge specified by two given nodes is not needed by the algorithm. This avoids the need for complicated data structures for access to the nodes such as search trees or hash tables, which are often needed in data structure libraries for graph algorithms which do not use indexed nodes.

6.2 Auction with Heap

The implementation of the auction algorithm for asymmetric assignment, as described in Section 4.3, uses a binary heap of size $n_1 + 1$ with price values as keys to access the price threshold between assigned and unassigned nodes. Although we only need n_1 nodes with the highest prices, the capacity of the heap has to be $n_1 + 1$, since every time a node is relabeled which is not already in the heap, we add it to the heap and remove the node with the lowest price again. This means that we temporarily have to manage $n_1 + 1$ nodes in the heap.

In contrast, the implementation presented in [Unt03] stores all nodes in the heap. This might affect the running time of the algorithm if the size of Y is much larger than the size of X (see computational results in Chapter 7).

6.3 Forward/Reverse Auction

The implementation of the forward/reverse auction algorithm (Section 3.8) does not require a heap at all; however, since bidding steps are performed on both the left hand side (X) and right hand side (Y) nodes, price variables are required for all nodes, and moreover, two adjacency lists are required: one for edges leaving each node in X, as in the other implementations, and one for incoming edges of each node in Y. Consequently, memory usage of this implementation is roughly twice as high as that of the other algorithms, considering the fact that the adjacency list is the main factor in terms of memory consumption.

6.4 Auction for Constrained Assignment with Heaps

This implementation of the algorithm described in Section 5.4 extends the implementation of the auction algorithm with a heap mentioned earlier. As described before, several heaps are used for the subsets Y_i; however, each node in Y is assigned to one subset Y_i and can therefore only be saved in the heap associated to that subset.

This makes the implementation quite straightforward. For the special case where there is only one subset $Y_i = Y$, execution of the algorithm is basically the same as the heap algorithm. The main difference lies in the relabeling of unmachted nodes between refinement steps. As explained in Section 5.4, unmatched nodes are relabeled to a value ϵ less than the smallest label of the matched nodes. As the computational results in Chapter 7 show, this has no noticeable impact on performance of the algorithm.

Chapter 7

Computational Results

We will now present experimental comparisons of the implementations presented in the previous chapter. Additionally, we take into account the previous implementation of an auction algorithm with virtual nodes, developed by Werner Unterhofer and described in [Unt03], since this code served as a basis for our implementations.

Running time of the algorithms was measured using the `getrusage` command, which allows to only count the time actually used by a process. This is especially important on multi-tasked systems where several processes are running at the same time, and background processes can impact stop-watch running time measurement. The data collected by means of the `getrusage` command is mostly independent from the processor load caused by other processes. Still, it should be noted that running time results of the algorithms could vary by a factor of up to 10% on the same input, so we should not expect more accuracy than that in our test.

The tests were performed on an Apple Powerbook G4 with a 1 GHz Motorola 7455 processor and 1 GB of RAM. Attention was paid to not let the tests use virtual memory, i.e. that all of the process data could be held completely in RAM, to avoid a performance hit due to the use of the significantly slower hard disk memory.

The tests were performed on pseudo-random graphs which were generated on the fly by the test process. These pseudo-random graphs are determined by a seed value, with which the pseudo-random number generator was initialised. That way, identical input graphs could be used for the different algorithms, without the need to save the input data explicitly to hard disk.

To achieve approximate average-case figures, 5 random graphs with the same input parameters and different seed values for the random number generator were tested, and the median running time was taken as a result. This method proved to be sufficiently robust to provide results with accuracy in the order of magnitude of the used testing method.

7.1 Edge Density

First, we investigate how the edge density of a graph influences the running time of the algorithms. We generate several random graphs with fixed node sets $|X| = 1000$ and $|Y| = 2000$. The maximal edge weight is chosen as $C = 10^6$. Because of the way the graphs are generated randomly, we choose the edge probabilities p, ranging from 0.1 to 1 in steps of 0.1, as input parameter instead of the number of edges E. Note that for a particular edge probabilities p and fixed node numbers $|X|$ and $|Y|$, the expected number of edges $\mathbb{E}(|E|)$ is also fixed.

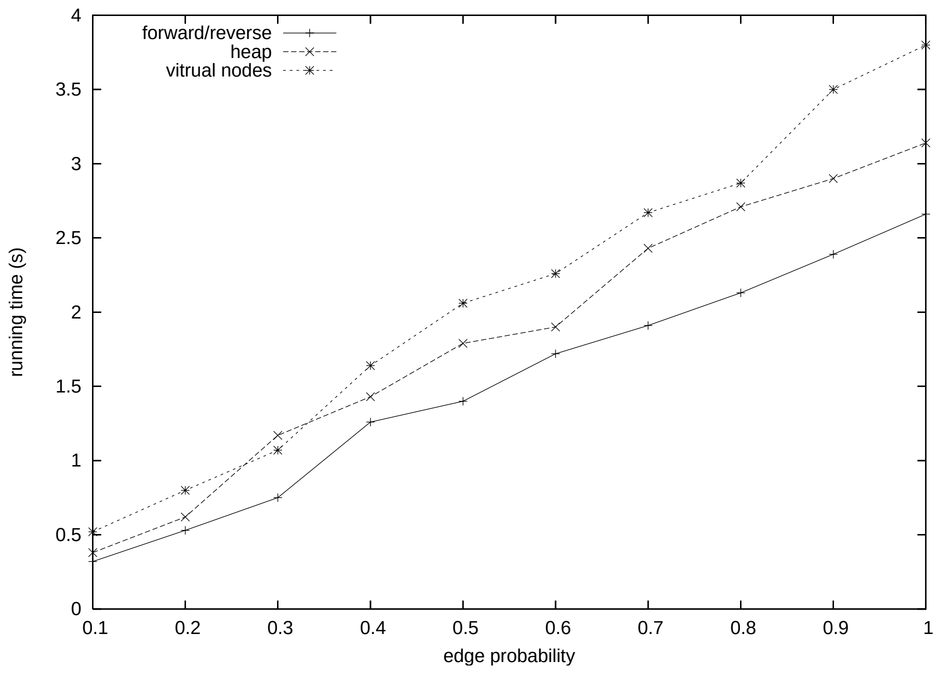
The running times are listed and graphically represented in Figure 7.1 on the facing page. For all algorithms, the running time increases roughly linearly with increasing edge probability. This is just the result one would expect under the assumption that our algorithms have similar running time behaviour as the basic auction algorithm, which needs $\mathcal{O}(nm\log(nC))$ steps and thus is linear in the number of edges as well.

The virtual node implementation and the heap implementation are almost on par. This comes at no surprise as these algorithms are quite similar, the main difference being the number of nodes managed in the heap. For these instances, the virtual node implementation always keeps $|Y|$ nodes in the heap, which is always twice as much as the heap implementation ($|X|$) here. This is obviously too little a difference to show up in the run time results.

The forward/reverse algorithm, in turn, shows a slightly better performance than the heap implementation; Running times are about 20% faster than those of the other algorithms for all probabilities. This result is comparable to the performance gain observed in [BCT93].

7.2 Asymmetry Between Vertex subsets

Next, we will have a look at how well the algorithms can handle increasingly asymmetric input graphs. We generate graphs with a fixed number of left hand side nodes $|X| = 1000$, and a fixed expected number of edges, $\mathbb{E}(|E|) = 10^6$, and an increasing



$\lvert X\rvert = 1000$, $\lvert Y\rvert = 2000$, $C = 10^6$			
Edge probability p	virtual nodes	heap ordered	forward/reverse
0.1	0.5	0.4	0.3
0.2	0.8	0.6	0.5
0.3	1.1	1.2	0.7
0.4	1.6	1.4	1.3
0.5	2.1	1.8	1.4
0.6	2.3	1.9	1.7
0.7	2.7	2.4	1.9
0.8	2.9	2.7	2.1
0.9	3.5	2.9	2.4
1.0	3.8	3.1	2.7

Figure 7.1: Running times (in seconds) for different edge probabilities p

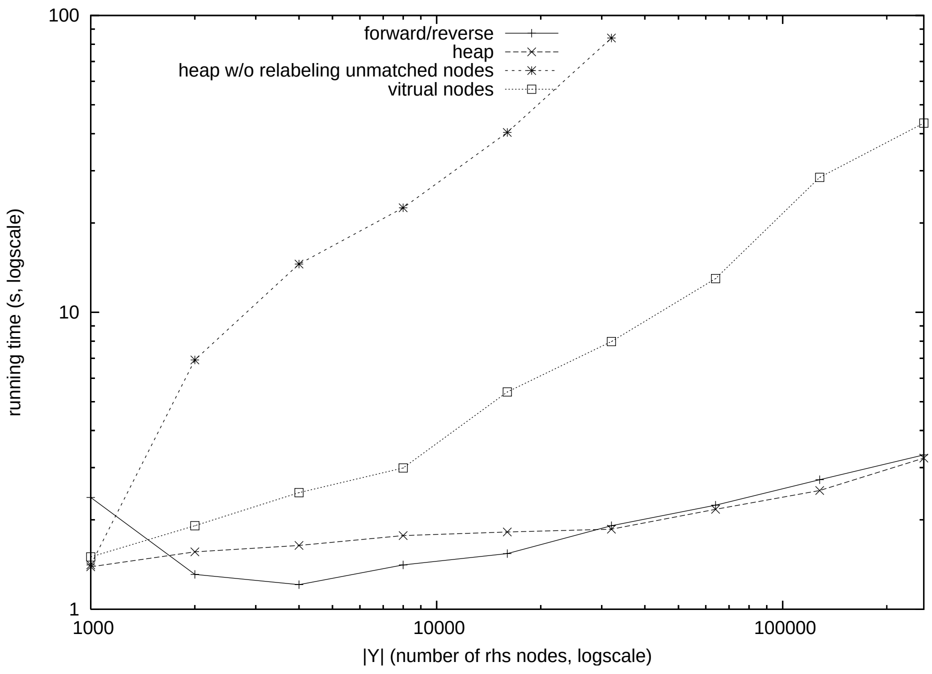
number of right hand side nodes $|Y|$. To get the desired expectation for $|E|$, we set the edge probability to $p = \frac{\mathbb{E}(|E|)}{|X|\cdot|Y|}$.

In this test, we also include the heap implementation without the heuristic presented in Section 4.4, which relabels unmatched nodes to the highest possible value after a refinement step is completed, to show the improvement gained by this technique. As we can easily see, the running time quickly gets much worse without this heuristic, so we only show results up to $|Y| = 32000$. The results are shown in Figure 7.2.

As expected, this test shows a significant difference in performance between the virtual nodes and the heap implementation if the size of $|Y|$ is much larger than the size of X, since the virtual node implementation has to manage much more nodes in its heap than the heap implementation.

The forward/reverse algorithm and the heap algorithm are roughly on par in this test. An notable result is that the forward/reverse algorithm performs worse on symmetric assignment problems than on asymmetric problems: the running time for $|X| = |Y| = 1000$ is clearly slower for this algorithm than for $|Y| = 2000$ (and worse than the other algorithms on the same data), whereas all other algorithms perform best on symmetric instances. The best result for this algorithm is $|X| = 1000$ and $|Y| = 4000$. Obviously the reverse bidding can show its advantage best if there are more "bidders" than objects in the auction.

For the heap implementation, the relabeling of unmatched nodes is very important for good performance on highly asymmetric instances, as can be seen in this test. While in the symmetric case this heuristic is not applicable and therefore both variants show the same performance, for asymmetric instances it takes a considerable amount of time to adapt the node prices of the unmatched nodes. This comes at no surprise: the prices are increasing at each relabel by a minimum amount of ϵ, and nodes which are not matched during a refinement step are not relabeled, and consequently have a relatively lower price than in the previous refine iteration. Therefore these nodes are more attractive in the next iteration; it might take several bidding increments with the new smaller ϵ to increase the price enough that the nodes will not get assigned. The more unassigned nodes there are after a refinement step—which is dependent on the degree of asymmetry in the assignment problem—the bigger the impact on performance caused by this problem. For that reason, we always used the heap algorithm with this heuristic enabled in the other tests.



$\lvert X\rvert = 1000$, $\mathbb{E}(\lvert E\rvert) = 10^6$, $C = 10^6$				
$\lvert Y\rvert$	virtual nodes	heap ordered	forward/reverse	(heap w/o relabel)
1000	1.5	1.4	2.4	1.4
2000	1.9	1.6	1.3	6.9
4000	2.5	1.7	1.2	14.
8000	3.0	1.8	1.4	22.
16000	5.4	1.8	1.5	40.
32000	8.0	1.9	1.9	84.
64000	12.	2.2	2.2	
128000	28.	2.5	2.7	
256000	43.	3.2	3.3	

Figure 7.2: Running times (in seconds) for different sizes of Y

7.3 Graphs with Different Vertex Set Sizes

We now investigate the influence of an increasing total number of vertices on a slightly unbalanced ($|Y| = 2|X|$) input with a fixed expected number of edges $\mathbb{E}(|E|) = 10^6$. Again, we set the edge probability to $p = \frac{\mathbb{E}(|E|)}{|X| \cdot |Y|}$ to get the right number of edges in our random graphs. The results can be seen in Figure 7.3.

The heap algorithm and the forward/reverse algorithm show a slightly more than linear increase in running time with an increasing number of nodes. This is the running time behaviour that the bounds $\mathcal{O}(nm \log(nC))$ for the basic auction algorithm suggests. For the virtual node implementation, however, the running time increases more rapidly, especially for larger values of $|X|$.

On small instances the heap algorithm and the virtual nodes algorithm both show similar performance, with the heap algorithm being slightly faster. This result goes in line with our earlier results, where these two algorithms performed similar for only slightly asymmetric instances.

The forward/reverse algorithm always has a noticeable advantage over the heap algorithm, with up to 40% faster running times. The difference for the smaller instances, however, is less significant, as we already have seen in our previous test.

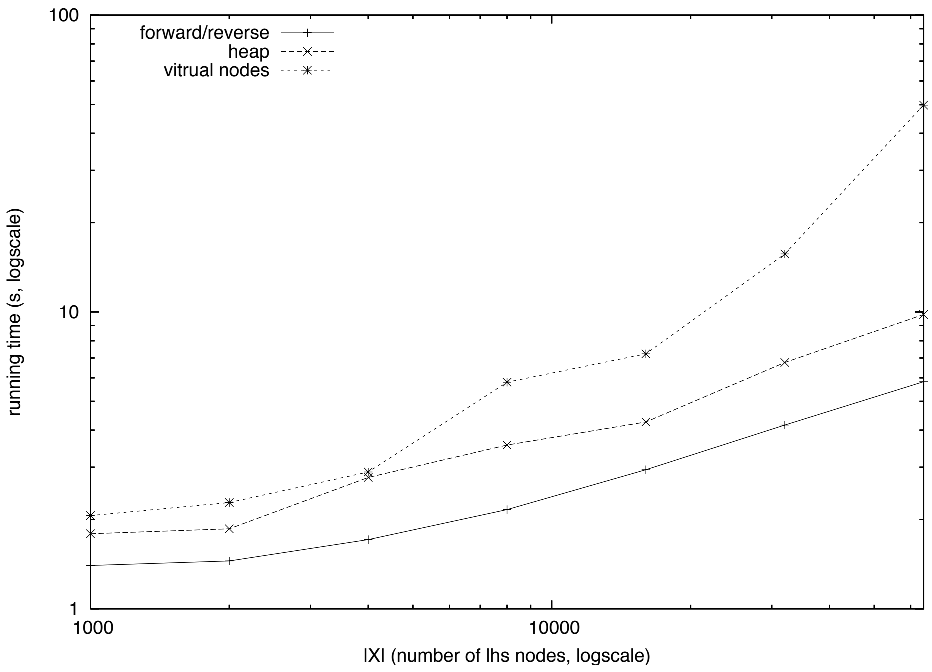
The test was only performed up to $|X| = 64000$, because for larger values the probability that some node in X would have less than two outgoing edges becomes too high. Remember that node that has only one outgoing edge does not have to be taken into account by the algorithm.

7.4 Graphs with Increasing Maximal Costs

As a final run time test for the asymmetric assignment algorithms, we look into different maximal weights for the assignment instances. The randomly generated graphs have fixed numbers of nodes $|X| = 1000$ and $|Y| = 2000$, and a fixed number of edges $|E| = 2 \cdot 10^6$ (edge probability $p = 1.0$). See Figure 7.4 for the results.

This test shows a very interesting property of the heap and virtual nodes algorithms. They have a peak in their running times at $C = 1000$, while the forward/reverse algorithm shows no irregularity at this point.

To find out what is causing this effect, we look at the number of bidding steps (push operations) and the cost of the intermediate solution after the refinement



$\lvert Y\rvert = 2\lvert X\rvert$, $\mathbb{E}(\lvert E\rvert) = 10^6$, $C = 10^6$				
$\lvert X\rvert$	$\lvert Y\rvert$	virtual nodes	heap ordered	forward/reverse
1000	2000	2.1	1.8	1.4
2000	4000	2.3	1.9	1.5
4000	8000	2.9	2.8	1.7
8000	16000	5.8	3.6	2.2
16000	32000	7.2	4.3	2.9
32000	64000	15.	6.8	4.2
64000	128000	49.	9.8	5.8

Figure 7.3: Running times (in seconds) for different numbers of nodes

steps of the algorithms. Here we see that most of the bidding steps take place in refinement iterations where the algorithm has actually already found an optimal solution. For higher maximal weights, it takes more iterations until an intermediate solution has optimal weight, but after that only very few more bidding steps than actually required are performed.

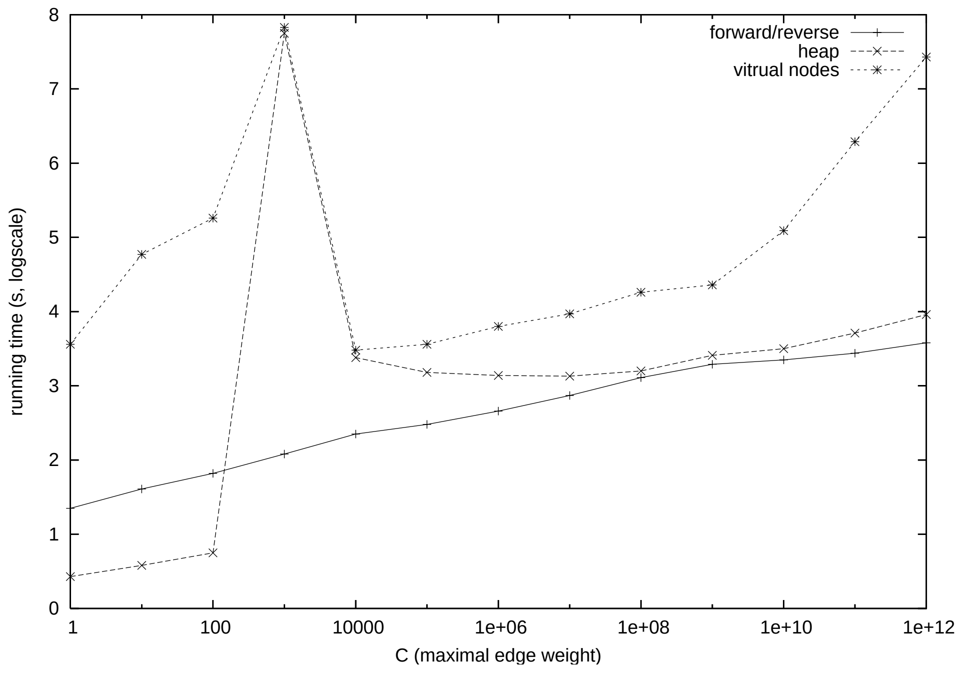
The problem we observe here is that while the primal solution is already optimal, the dual solution does not lead directly to the same solution after reducing ϵ and have to be adapted in every refinement step. The prices are too similar for the algorithm to identify the edges of the optimal solution by their reduced costs after a refinement step, and they are too different that enough different optimal solutions of the problem would exist, which would simplify the computation.

The forward/reverse algorithm avoids this problem by adapting the node prices in both directions alternately, and therefore propagating the changes to the price values faster than in the pure forward auction algorithm.

For the smaller maximal weights, optimal solutions are in most cases very easy to find, because in most cases the solution found in the first iteration is already optimal and can be found with the minimum number of bidding operations, because for every node $i \in X$ there is still an unassigned node in $j \in Y$ such that the cost c_{ij} for the edge (i, j) is the cheapest of all outgoing edges of i. Consequently, there are many different optimal solutions.

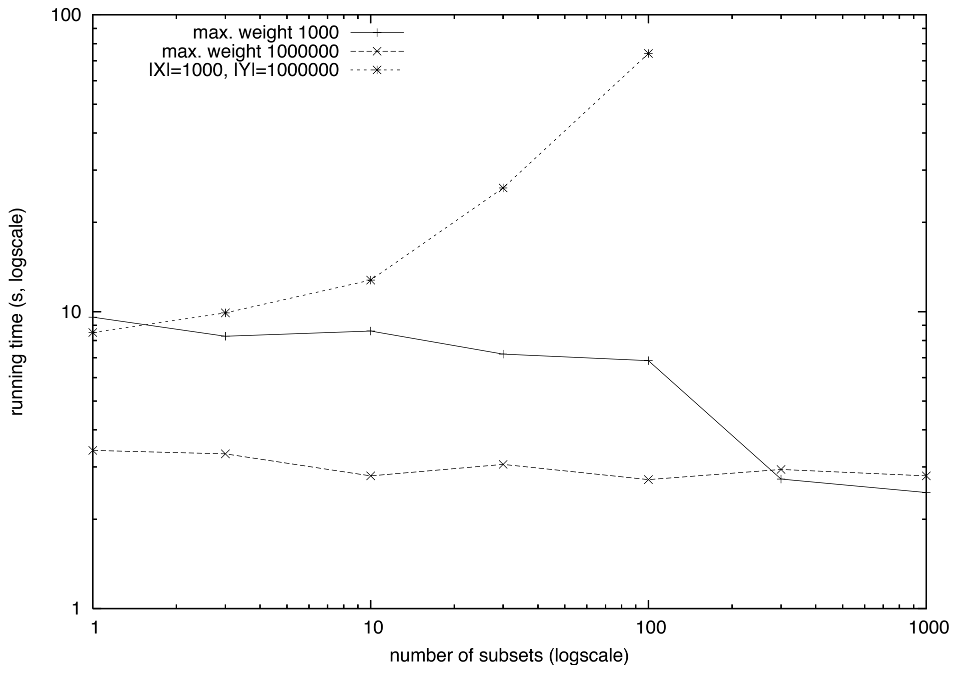
7.5 Constrained Assignment

After our detailed analysis of the running time for asymmetric assignment algorithms, we now take a look at our algorithm for the constrained assignment problem, which is based on the (purely forward) auction algorithm with a heap. We perform the test on random graphs with fixed node and edge numbers and maximal weights, and with different numbers s of subsets Y_i. 3 different types of graphs were tested: one with $|X| = 1000$, $|Y| = 2000$, $C = 1000$, and $p = 1.0$ (and therefore $|E| = 2 \cdot 10^6$), one with the same data except $C = 1000$, and one with $|X| = 1000$, $|Y| = 10^6$, $C = 10^6$, and $p = 0.001$ ($\mathbb{E}(|E|) = 10^6$). In each tested instance, the subsets Y_i all have the same number of elements (by a difference of at most 1). Since our previous tests showed that edge density and running time correlate linearly, we do these test only on complete graphs; short spot test show that the results on more sparse graphs are analogous. The results are shown in Figure 7.5 on page 58.



$\|X\| = 1000,\ \|Y\| = 2000,\ \|E\| = 2 \cdot 10^6$			
C	virtual nodes	heap ordered	forward/reverse
10^0	3.56	0.43	1.35
10^1	4.77	0.58	1.61
10^2	5.26	0.75	1.82
10^3	7.83	7.75	2.08
10^4	3.48	3.38	2.35
10^5	3.56	3.18	2.48
10^6	3.80	3.14	2.66
10^7	3.97	3.13	2.87
10^8	4.26	3.20	3.11
10^9	4.36	3.41	3.29
10^{10}	5.09	3.50	3.35
10^{11}	6.29	3.71	3.44
10^{12}	7.43	3.96	3.58

Figure 7.4: Running times (in seconds) for different maximal costs



$\lvert X\rvert = 1000$, $p = 1.0$			
s	$C = 1000$, $\lvert Y\rvert = 2000$	$C = 10^6$, $\lvert Y\rvert = 2000$	$C = 10^6$, $\lvert Y\rvert = 10^6$
1	9.8	3.4	8.5
3	8.3	3.3	9.9
10	8.6	2.8	12.
30	7.2	3.1	26.
100	6.8	2.7	74.
300	2.7	2.9	
1000	2.5	2.8	

Figure 7.5: Running times (in seconds) for different sizes of Y

For the graphs with maximal weight $C = 1000$, remember that we observed that the heap algorithm for asymmetric assignment had more difficulties than for higher maximal weights. Interestingly, if Y is partitioned into many small subsets Y_i in the constrained assignment problem, the problem gets easier to solve for our algorithm.

If we set the maximal weight to $C = 10^6$, on the other hand, running time is almost the same for all numbers s of subsets Y_i (only very slightly decreasing for larger numbers s). So we can conclude that for instances where $|X|$ and $|Y|$ are in the same order of magnitude, solving the constrained assignment problem does not take more time than solving the (asymmetric) assignment problem.

Only if $|Y|$ is much larger than $|X|$, as it is in our third test series, we can observe an increasing performance hit with higher numbers s of subsets Y_i.

Concluding, these results show that in most cases, and especially for our application in warehousing, where the number of products and the number of locations are in the same order of magnitude, our implementation is certainly competitive with the virtual node implementation developed in the preceding project.

Appendix A

Generating Sparse Random Bipartite Graphs

To test our algorithms, we needed a way to generate random graphs as input. The requirements for the graphs produced by the generator were the following:

- Input values are: the number of nodes on the left side $n_1 = |X|$ and on the right side $n_2 = |Y|$; the edge probability p; the maximal edge weight C.
- The probability that an edge (i, j) existed in the graph is p independently for all pairs $(i, j) \in X \times Y$.
- The weight c_{ij} of an edge (i, j) is an integer value between 1 and C randomly chosen with uniform distribution.

From the preceding project, a generator was available which met these criteria. However, it produced random graphs by iterating over all pairs $(i, j) \in X \times Y$ and then randomly inserting an edge into the graph with probability p (using a random number generator). Clearly, for small p, this method uses much more iterations ($\mathcal{O}(n_1 n_2)$) than edges are output ($\mathcal{O}(pn_1 n_2)$ in the average case). We wanted to generate the graphs in time $\mathcal{O}(m)$ without changing the properties of the random graphs.

Another approach is randomly choosing a pair $(i, j) \in X \times Y$ and adding the edge (i, j) to the graph if not already present. This requires, however, random access to the edges already present in the graph, using a hash table or another appropriate data structure, meaning that generating random graphs this way requires $\mathcal{O}(m)$

memory and does not run in time $\mathcal{O}(m)$ in the worst case. Moreover, simply randomly picking pn_1n_2 edges does not result in the desired edge probability p, since edges might be picked more than once.

A straightforward approach is to iterate over the vertices X in an outer loop, and when iterating over the vertices Y in the inner loop, randomly choose how many of the pairs are skipped by increasing the step width in the loop by a random variable.

However, on closer examination, an uniformly distributed chosen step width does not lead to the required random graphs. To obtain an expected number of pn_1n_2 edges, the step width would have to be $S_U = \left\lceil \frac{2Un_2}{p} \right\rceil$, where U is a random variable uniformly distributed between 0 and 1. The expected value of this random variable is $\mathbb{E}(S_U) = \left\lceil \frac{2\frac{1}{2}n_2}{p} \right\rceil = \lceil \frac{n_2}{p} \rceil$, meaning that $\lfloor \frac{p}{n_2} \rfloor$ of the n_2 vertices are chosen. But the maximal step width is $\frac{2n_2}{p}$ in this case, and not $n_2 + 1$ as in the case of the desired random graph (if all n_2 nodes are skipped). So a different random distribution for the step width is required.

For any i from 1 to n_2, the probability for a step width $S = i$ (density function) is $\Pr[S = i] = f_S(i) = (1-p)^{i-1}p$, as we can easily see: $i - 1$ nodes are skipped with a probability of $1 - p$ before a node is chosen with probability p. The corresponding distribution function $t \mapsto \Pr[S \leq t]$ is $F_S(t) = \sum_{i=1}^{t}(1-p)^{i-1}p = 1 - (1-p)^t$.

Since at most n_2 items are skipped, the maximal step width is n_2+1. The probability that all n_2 items are skipped is $(1-p)^{n_2}$.

Together, the required distribution function is

$$F_S(t) = \begin{cases} 1 - (1-p)^t & \text{for } t \leq n_2 \\ 1 & \text{for } t \geq n_2 + 1 \end{cases}$$

We can, of course, only generate uniformly distributed discrete values on a computer, using the `rand()` or `drand48()` functions, which returns a value in the range between 0 and `RAND_MAX` or between 0 and 1, respectively. Therefore we need a function g that can be applied to an (approximately) uniformly distributed random variable U to get a random value $S = g(U)$ with the desired random distribution $\Pr[S \leq t] = 1 - (1-p)^t$. Choosing the inverse of the required density function $S = F_S^{-1}(U) = \min\left(\frac{\log(1-U)}{\log(1-p)}, n_2 + 1\right)$ gives us a random variable with the desired properties:

$$F_S(t) = \Pr[U \leq F_S(t)] = \Pr[F_S^{-1}(U) \leq t] = \Pr[S \leq t]$$

Since we need discrete values in the range $1 \ldots n_2+1$, we have to round S to the next highest integer value; we also have to make sure that U is not equal to 0, since our minimal step width is 1. Since `drand48()` returns `double` values in the interval $[0, 1)$ (excluding 1), we can obtain the desired random value U by subtracting the value returned by `drand48()` from 1. Now the resulting value has the needed random distribution, and using this value as step width, we can generate our random graphs with constant memory usage in linear time.

List of Algorithms

1 Cost scaling algorithm for minimum cost flow . . . 22
2 *refine*(ϵ, x, π); . . . 23
3 *push/relabel*(i) . . . 23
4 *bipartite-refine*(ϵ, x, π); . . . 24
5 *bipush/relabel*(i) . . . 25
6 Cost scaling (auction) algorithm for assignment . . . 28
7 *refine-auction*(ϵ, x, π); . . . 28
8 *double-push*(i) . . . 29
9 *refine-forward-reverse*(ϵ, x, π); . . . 30
10 *reverse-double-push*(i) . . . 31
11 *double-push-heap*(i) . . . 35
12 *refine-forward-reverse-asymmetric*(ϵ, x, π); . . . 37
13 *reverse-double-push*(i) . . . 37
14 *double-push-constrained*(i) . . . 43

Bibliography

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows.* Prentice Hall Inc., Englewood Cliffs, NJ, 1993. Theory, algorithms, and applications.

[AOST94] Ravindra K. Ahuja, James B. Orlin, Clifford Stein, and Robert E. Tarjan. Improved algorithms for bipartite network flow. *SIAM J. Comput.*, 23(5):906–933, 1994.

[BC92] Dimitri P. Bertsekas and David A. Castañon. A forward/reverse auction algorithm for asymmetric assignment problems. *Comput. Optim. Appl.*, 1(3):277–297, 1992.

[BCT93] Dimitri P. Bertsekas, David A. Castañon, and Haralampos Tsaknakis. Reverse auction and the solution of inequality constrained assignment problems. *SIAM J. Optim.*, 3(2):268–297, 1993.

[Ber81] Dimitri P. Bertsekas. A new algorithm for the assignment problem. *Math. Programming*, 21(2):152–171, 1981.

[Ber88] D. P. Bertsekas. The auction algorithm: a distributed relaxation method for the assignment problem. *Ann. Oper. Res.*, 14(1-4):105–123, 1988.

[Ber94] Dimitri P. Bertsekas. Mathematical equivalence of the auction algorithm for assignment and the ϵ-relaxation (preflow-push) method for min cost flow. In *Large scale optimization (Gainesville, FL, 1993)*, pages 26–44. Kluwer Acad. Publ., Dordrecht, 1994.

[Cas93] David A. Castañon. Reverse auction algorithms for assignment problems. In D. S. Johnson and C. C. McGeoch, editors, *Network flows and matchings: First DIMACS implementation challenge*, pages 407–429, Providence, RI, 1993. DIMACS, American Math. Soc.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[GK95] Andrew V. Goldberg and Robert Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Programming*, 71(2, Ser. A):153–177, 1995.

[Gol97] Andrew V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22(1):1–29, 1997.

[GT90] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, 15(3):430–466, 1990.

[JV87] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.

[Kuh55] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.

[OA92] James B. Orlin and Ravindra K. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *Math. Programming*, 54(1, Ser. A):41–56, 1992.

[PS98] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Dover Publications Inc., Mineola, NY, 1998. Corrected reprint of the 1982 original.

[Unt03] Werner Unterhofer. Effiziente Berechnung von minimalen perfekten Matchings in gewichteten bipartiten Graphen. Diplomarbeit, TU München, Nov 2003.