The Column Subtraction Method for the Traveling Salesman Problem

by

Friedel Wolff

Dissertation

submitted in fulfilment
of the requirements for the degree

Master Of Science

in

Computer Science

in the

Faculty of Science

at the

Rand Afrikaans University

Supervisor: Prof. T.H.C. Smith

November 2004

Abstract

Keywords:

- Combinatorial optimization
- Parallel programming (Computer science)
- Branch and bound algorithms
- Traveling-salesman problem

The Set Partitioning Problem (SPP) and the Traveling Salesman Problem (TSP) are problems in discrete optimisation. The Column Subtraction Method (CSM) is presented as a method to solve these two problems. It was implemented to execute in uniprocessor and parallel computing environments.

The CSM [13] is an efficient branch-and-bound technique for solving the SPP. The CSM is also adapted to solve the TSP, and the efficiency of the method in solving the TSP is investigated. An implementation of the CSM for the TSP is also presented. The CSM starts by solving a relaxation of the problem. A search tree is constructed in which each node corresponds to a subproblem where certain variables are fixed at certain values. Several rules exist to limit the size of the search tree.

For the TSP the special structure of the relaxation (the Relaxed 2-Matching Problem, RMP2) is exploited and specialised primal and dual simplex methods are used. These methods can be implemented entirely with integer variables. A new method is also presented to calculate which nodes are connected by a given edge. The Union-Find algorithm is used as a scalable method for determining if a candidate solution is a valid tour. The CSM is tested for both the SPP and TSP in a uniprocessor configuration and on a cluster of workstations. Superlinear speedup is obtained in some instances.

Opsomming

Trefwoorde:

- Kombinatoriese optimering
- Parallelle programmering (Rekenaarwetenskap)
- Vertak-en-begrensalgoritme
- Handelsreisigerprobleem

Die Versamelingpartisieprobleem (Eng: Set Partitioning Problem, SPP) en die Handelsreisigerprobleem (Eng: Traveling Salesman Problem, TSP) is beide 0-1-probleme in diskrete optimering. Die Kolomaftrekkingsmetode (Eng: Column Subtraction Method, CSM) word aangebied as metode om hierdie probleme op te los. Dit is geïmplementeer om in enkelverwerker- en in parallelle omgewings te kan uitvoer.

Die CSM [13] is 'n vertak-en-begrensmetode wat met groot sukses gebruik is om die SPP op te los. Vir die SPP begin dit met die oplos van 'n LP-verslapping van die probleem. 'n Soekboom word geskep waarin elke nodus ooreenstem met 'n subprobleem waar sekere veranderlikes by spesifieke waardes vasgemaak word. Hierdie vasmaak van die veranderlikes word gedoen deur 'n wysiging van die regterkant van die beperkings van die Lineêre Programmeringsformulering van die probleem ($\mathbf{b}$ in die formulering $A\mathbf{x} = \mathbf{b}$). Kolomme van A wat ooreenstem met die veranderlikes wat by 1 vasgemaak word, word naamlik afgetrek van $\mathbf{b}$.

Hierdie verhandeling begin met ’n bespreking van die verbeterde CSM vir die SPP van Smith en Thompson [30]. Verskeie tegnieke bestaan om die grootte van die soekboom te beperk. Die verbeterde CSM is geïmplementeer vir ’n CRAY T3E superrekenaar. Vir hierdie verhandeling is die implementasie aangepas en daar word hier ook verduidelik wat nodig was om dit aan te pas sodat dit kan loop op ’n gewone boodskapstuurstelsel (MPI in hierdie geval). Uitvoertye vir hierdie implementasie dui aan dat die CSM ook suksesvol uitvoer op ’n werkstasietros - superlineêre versnelling word in sekere gevalle behaal.

Die CSM is aangepas om die TSP op te los. Vir die TSP begin die CSM met die oplos van ’n verslapping van die probleem, die Kontinue 2-Paringsprobleem (Eng: Relaxed 2-Matching Problem, RMP2). Die RMP2 het ’n spesiale struktuur wat uitgebuit kan word met ’n gespesialiseerde primaal-simpleksmetode. Hierdie simpleksmetode kan geheel en al met heelgetalveranderlikes geïmplementeer word. Weens hierdie simpleksmetode kan nie-basiese veranderlikes moontlik by hul bogrens (1) wees. ’n Soekboom word geskep waarin elke nodus ooreenstem met ’n subprobleem waar sekere veranderlikes by spesifieke waardes vasgemaak word. By elke nodus word ten minste een veranderlike vasgemaak by ’n veranderde waarde: ’n veranderlike wat by sy ondergrens (0) was word vasgemaak by 1, of ’n veranderlike wat by sy bogrens (1) was word vasgemaak by 0. Hierdie verandering van die veranderlikes word gedoen deur ’n wysiging van **b**. Kolomme van A wat ooreenstem met die veranderlikes wat by veranderde waardes vasgemaak word, word óf afgetrek van **b**, óf by **b** getel. Hierdie manipulasie kan tot duale onuitvoerbaarheid lei. Vir die heroptimering in die geval van duale onuitvoerbaarheid, word ook gebruik gemaak van ’n duaal-simpleksmetode wat die spesiale struktuur van die probleem uitbuit. Hierdie duaal-simpleksmetode is aangepas vir die CSM om ook geheel en al met heelgetalveranderlikes geïmplementeer te word. Daar is dus geen dryfpuntveranderlikes nodig vir die CSM vir die TSP nie. Dit het die voordeel dat numeriese onstabiliteit tydens die simpleksmetodes nie moontlik is nie.

Tydens implementasie is gebruik gemaak van sekere algoritmes wat spesifiek gekies is weens hulle geskiktheid vir groot probleme. 'n Berekening wat baie gebruik word tydens die CSM vir die TSP is om te bereken watter stede verbind word deur 'n gegewe pad. 'n Nuwe metode wat hierdie berekening in $O(1)$ uitvoer, word aangebied. 'n Toets wat gereeld uitgevoer moet word, is om te bepaal of die oplossing van 'n subprobleem by 'n nodus in die soekboom wel 'n geldige toer vorm. Vir hierdie toets is gebruik gemaak van die Vereniging-Vindalgoritme (Eng: Union-Find algorithm) wat 'n asimptotiese uitvoertyd het wat amper so goed is soos $\Theta(n)$ vir n stede.

Die implementasie is getoets op sommige van die probleme uit die TSPLIB - 'n biblioteek van Handelsreisigerprobleme. 'n Vertaler is ontwikkel om die lêerformaat van die TSPLIB te kan lees. Daar is gevind dat die CSM-soekboom baie groot is, selfs vir klein probleme. Sommige van die reëls wat gebruik is om die soekboom se grootte te beperk by die CSM vir die SPP was oneffektief by die TSP. Alhoewel die probleme met minder as 30 stede in redelike tyd opgelos kan word (enkele sekondes op 'n hedendaagse Pentium 4-rekenaar) het die uitvoertye aansienlik gegroei met die groter probleme.

Parallelisering van die CSM vir die TSP is getoets op 'n werkstasietros, en superlineêre versnelling is weer in sekere gevalle behaal. Die skema vir ladingbalansering wat gebruik is vir die SPP was soms egter oneffektief wanneer 16 of meer werkstasies gebruik is.

Contents

Chapter 1

Introduction

The Set Partitioning Problem (SPP) and the Traveling Salesman Problem (TSP) are problems in discrete optimisation. They can be defined with the aid of linear programming formulations. They have many real world applications and there has been continued effort to solve them in less time and to solve bigger problems of these types.

This dissertation discusses the Column Subtraction Method (CSM) as an efficient technique for solving the SPP. The CSM is also adapted to solve the TSP, and the efficiency of the method in solving the TSP is investigated. An implementation of the CSM for the TSP is also presented.

The use of parallel computing, both with multi-processor computers and clusters of workstations, has increased in recent years and are often used to solve big optimisation problems. In solving the biggest TSP to date, the majority of the work was done on a cluster of 96 dual processor Intel Xeon Linux workstations [4].

Parallelised versions of the CSM for both the SPP and the TSP are discussed and their efficiency is determined. Running times on different configurations are presented for comparison. A cluster of Linux workstations was used for this purpose.

1.1 The Set Partitioning Problem

The SPP is structurally related to the Set Covering and Set Packing Problems. The linear programming formulations of these three problems are very similar. The CSM was presented in [13] as a method for solving all three types of problems. Only the CSM for the SPP will be discussed in this dissertation.

The SPP will be formally defined in Chapter 2 along with the CSM for the SPP. This discussion of the CSM will serve as background for the implementation of the CSM for the TSP, to be discussed in Chapter 5.

1.2 The Traveling Salesman Problem

The TSP has long been studied as one of the hardest problems in combinatorial optimisation. It is often mentioned as an example of a class of problems that are hard to solve.

It can informally be formulated as follows: A salesman has to visit n cities, visiting each city exactly once, and also has to return to his starting city. Such a route is called a *tour*. Traveling on a path between two cities has an associated cost, which is supplied as part of the problem. The goal is to find the tour the salesman must travel in order to minimise the sum of all the costs associated with the tour which he travels.

Various ways of solving the TSP have been proposed. Leenen [21] gives a summary of the developments in this research field. For more detail, see [20]. Techniques have also been developed to find provably good tours which are not necessarily optimal. Such techniques are called heuristics. Most research on exact solutions for the TSP in recent years has been based on the Cutting-Plane method of Dantzig, Fulkerson, and Johnson [8].

Of particular interest is the Branch-and-Cut method by Padberg and Rinaldi [23] that begins by solving a relaxation of the TSP. Side constraints are continually added to reduce the search space that needs to be considered. The Concorde TSP solver [2] by Applegate, Bixby, Chvátal, Cook, and Helsgaun has to date solved the hardest problems and is based on the Branch-and-Cut method. The following table from [3] outlines some of the milestones in TSP computation:

Year	**Research Team**	**Size of Instance**	**TSPLIB Name**
1954	G. Dantzig, R. Fulkerson, and S. Johnson	42 cities	dantzig42
1971	M. Held and R.M. Karp	64 cities	dantzig42 + 22 random cities
1975	P.M. Camerini, L. Fratta, and F. Maffioli	100 cities	hk48 + two smaller instances
1977	M. Grötschel	120 cities	gr120
1980	H. Crowder and M.W. Padberg	318 cities	lin318
1987	M. Padberg and G. Rinaldi	532 cities	att532
1987	M. Grötschel and O. Holland	666 cities	gr666
1987	M. Padberg and G. Rinaldi	2 392 cities	pr2392
1994	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	7 397 cities	pla7397
1998	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	13 509 cities	usa13509
2001	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	15 112 cities	d15112
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K. Helsgaun	24 978 cities	sw24978

All of these problems are part of the TSPLIB [24] which is discussed in

Chapter 7.

To implement the CSM for the TSP, several changes had to be made. Contributions by Leenen [21] and Geldenhuys [9] on implementing the Branch-and-Cut method [23] were used to implement parts of the Column Subtraction Method.

1.3 Overview of this dissertation

The CSM for the SPP is presented in Chapter 2. It introduces the idea of column subtraction, and explains how the search space of the problem is covered by constructing a search tree. Fathoming rules that prune the search tree are discussed. The implementation of the CSM was also ported to a general message passing system, and this is also discussed. This chapter is based on the work of Smith and Thompson [30] and provides background knowledge of the CSM for Chapter 5 where the CSM for the TSP is presented.

From Chapter 2 the role of the primal and dual simplex method in the CSM will be clear. The relaxation used for the TSP has a special structure that allows for specialised versions of the primal and dual simplex method. These methods are discussed in Chapters 3 and 4 respectively. This is based on the work of Leenen [21] and Geldenhuys [9].

With the background presented in Chapters 2 to 4, the CSM for the TSP is presented in Chapter 5. Column subtraction (and addition) in the case of the TSP is discussed as well as the adaption of the fathoming rules.

Chapter 6 presents certain techniques that are needed for the implementation of some of the fathoming rules described in Chapter 5. The algorithms in this chapter focus on scalability.

The TSPLIB (mentioned earlier) is discussed in Chapter 7. Other details of implementation are discussed, such as the use of integer variables to represent fractional quantities. Results from executing the CSM for the TSP on a cluster of workstations are also presented and analysed.

1.4 Prerequisites for reading this dissertation

It is assumed that the reader is mostly familiar with the following topics and their associated terminology:

- Graph theory - the TSP is first defined as a graph theoretic problem, and terminology from graph theory is used throughout Chapters 3 to 7.
- Linear programming - relaxations of both the SPP and TSP are formulated as linear programs and linear programming terminology is used throughout. With the graph theoretic formulation of the TSP, the linear programming formulation can be presented.
- Pseudocode - many pieces of pseudocode are provided to formulate techniques and to illustrate implementation detail.

Chapter 2

The Column Subtraction Method for the Set Partitioning Problem

Harche and Thompson [13] presented the Column Subtraction Method (CSM) for solving large Set Covering, Set Packing and Set Partitioning Problems. It works by first solving a Linear Programming relaxation and then performing a branch-and-bound search. It has been shown that the CSM can effectively be parallelised. Good results were reported by Smith and Thompson [29] after implementing the CSM on a massively parallel Connection Machine, obtaining in some instances superlinear speedups. Smith and Thompson [30] presented several improvements to the original CSM for Set Partitioning Problems (SPP), resulting in speedups of several orders of magnitude.

This chapter serves as a background for the CSM for the Traveling Salesman Problem (see Chapter 5) and will cover the original CSM and the improvements presented in [30]. Only the application for the SPP is discussed here. The implementation by Smith and Thompson [30] made use of platform specific libraries on the CRAY T3E which they used. Through the rest of the chapter the process of porting this implementation to a general message passing system, MPI [11], is discussed.

2.1 The Set Partitioning Problem

The Set Partitioning Problem is a problem in combinatorial optimisation. Along with the Set Packing and Set Covering problems it has many applications, as mentioned in [17]:

> Many applications arise having the packing, partitioning and covering structure. Delivery and routing problems, scheduling problems and location problems often take on a set covering structure whereby one wishes to assure that every customer is served by some location, vehicle or person. Other applications include switching theory, the testing of VLSI circuits, and line balancing.

The SPP can be formulated as follows:

$$\begin{array}{rl} \textit{Minimise} & \mathbf{cx} \\ \textit{Subject to} & A\mathbf{x} = \mathbf{e} \\ & \mathbf{x} \in \{0,1\}^n, \end{array}$$

where $\mathbf{c}$ is an n-dimensional vector of positive costs, A is an $m \times n$ 0-1 matrix, and $\mathbf{e}$ is an m-dimensional vector of ones. When a set of variables provides an integer solution to the SPP, the columns corresponding to the positive variables add up to a vector of ones. This set of columns is said to *cover* the rows or to *provide a cover*.

2.2 The Column Subtraction Method

The CSM starts by solving a relaxation of the Integer Programming formulation given above. The objective function value of the optimal solution associated with the relaxation (zLP) provides a lower bound on the optimal objective function value for the SPP. If this solution to the relaxation contains only integer valued variables, it is also a solution to the SPP and the algorithm can be terminated. If the solution is not integer, one or more of the non-basic columns will form part of an optimal integer solution.

A heuristic is used to find an upper bound (zIP) on the objective function value of the Integer Programming problem. If the heuristic fails, zIP is set to ∞. A list L is constructed with the indices of all the variables that are non-basic in the optimal solution to the relaxation and that have a reduced cost less than $zIP - zLP$. These are the only variables that we need to consider for increasing their value to one, as increasing a non-basic variable outside this list will cause the objective function value to equal or exceed the cost of the current best known solution (zIP). L will interchangeably be used to refer to a list of variables as well as indices.

A search tree is constructed where at each node (except the root node) the CSM will fix some of these non-basic variables in L to a value of 1, while prohibiting others to assume positive values.

The first child of the root node is created by fixing the variable L_1 to a value of 1. The i-th child of the root node, $i > 1$, is created by fixing the variable L_i to a value of 1 while fixing L_1 to L_{i-1} to a value of 0. The j-th child of a node for which L_i was fixed at value 1, is created by additionally fixing L_{i+j} to value 1 and also fixing L_{i+1} to L_{i+j-1} to value 0. The set of variables fixed at 1 on the path from the root node to a node is the *partial partition* associated with that node. By constructing these partial partitions differently for each child, we ensure that we will cover the entire search space for the problem.

In some cases, however, it is not necessary to explicitly create all nodes in a branch if they cannot yield better solutions. The original CSM gave three rules that specify when a node in the search tree can be *fathomed* without creating any children. The improvements presented in [30] specified six rules over and above the original three. These nine rules will be numbered Rule 1 to 9.

When a column is fixed at 1 to become part of the partial partition for a node, it has to be orthogonal to those already present. If it is not, the node can be fathomed (**Rule 1**). With m constraints, this ensures that the depth of the search tree can never exceed m.

For each node an *Associated Linear Program* (ALP) can be formulated:

$$\begin{aligned} \textit{Minimise} \quad & \mathbf{cx} \\ \textit{Subject to} \quad & A\mathbf{x} = \mathbf{e}' \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

where $\mathbf{e}'$ is formed by subtracting the columns of A that corresponds to the partial partition, from $\mathbf{e}$. The optimal basis (B) for the SPP relaxation provides a dual-feasible basic solution for the ALP, with $B^{-1}\mathbf{e}'$ giving the values of the basic variables. The objective function value of this solution provides a lower bound on the cost of completing the node's associated partial partition. This lower bound also holds for all nodes below the current node in the search tree. The node can therefore be fathomed if the cost of completing the associated partial partition is greater than or equal to zIP (**Rule 2**).

If the solution is primal feasible and integer, a better solution to the SPP has been found and zIP can be updated. The node can also be fathomed as no children can provide a better solution (**Rule 3**). When zIP is updated, L can be shortened by removing all variables with reduced cost greater than or equal to zIP - zLP.

2.3 The improved Column Subtraction Method

Rules 2 and 3 of the original CSM made it necessary to use $\mathbf{x} = B^{-1}\mathbf{e}'$. Smith and Thompson [30] suggested that by using the revised simplex method and storing the basis B in product form, better accuracy can be achieved. They also presented six extra rules to further aid in pruning the search tree. These improvements resulted in speedups of several orders of magnitude.

A node can be fathomed without computing $B^{-1}\mathbf{e}'$ if one of these conditions hold:

- the cost of the partial partition alone is greater than or equal to zIP (**Rule 4**).
- the partial partition provides a complete cover ($\mathbf{e}' = \mathbf{0}$) (**Rule 5**).

If a node could not be fathomed by any of Rules 1 to 5, then the basic solution $B^{-1}\mathbf{e}'$ is either non-integer or primal-infeasible. If it is primal-infeasible, the dual simplex method can be applied to the ALP.

The size of the ALP can be reduced by eliminating from consideration all columns of A which are not orthogonal to the sum of the columns of the partial partition. Let A' be this reduced matrix. Furthermore, if a row of A' contains only a single 1 where $\mathbf{e}'$ has a 1, the column containing the 1 will necessarily be part of a solution. Therefore only a single child node need to be created in such a case.

If the row of A' only contains zeros where $\mathbf{e}'$ has a 1, the node can be fathomed as no solution is possible (**Rule 6**). If a node could still not be fathomed, the dual simplex algorithm is applied.

The node can now be fathomed if one of these conditions hold:

- primal feasibility could not be obtained (**Rule 7**)
- the resulting objective function value is greater than or equal to zIP (**Rule 8**)
- primal feasibility is obtained and the resulting solution is integer (**Rule 9**). In this case the list L is truncated as we did for Rule 3.

If none of these rules apply, the node cannot be fathomed and the search needs to continue down this branch of the search tree.

2.4 Implementation

An implementation of the improved CSM was programmed in the C programming language. It was also adapted to execute in a parallel programming environment. Initially, the i-th process would start processing the subtree rooted at the i-th child of the root node of the search tree. After exhausting the search of a specific subtree, a process will continue the search in a new subtree rooted at another child of the root node. Each of these children of the root node corresponds to an entry in the list L. This process continues until the end of the list L is reached. A master-slave approach is used where the master process supplies slave processes with unprocessed subtrees.

The following descriptions define some of the additional variables and functions used in the following sections:

rcsize the size of the list L

numProcs the number of processes in the parallel computer

rootNext this variable stores the number of the next subtree to process

zIPs an array where `zIPs[i]` stores the best zIP value discovered by process `i`

subtrees an array where `subtrees[i]` stores the number of the subtree that process `i` processes

reduce() a function that updates the list L by removing indices of columns with a reduced cost greater than or equal to zIP - zLP

EPS (ϵ) a small positive fractional number used for floating point comparisons

All entries of the zIPs array are initialised to the best known zIP value after solving the LP relaxation. The subtrees array is initialised to distribute the first set of subtrees to process, in sequential fashion.

```
for(j = 0; j < numProcs; j++) {
    zIPs[j] = zIP;
    subtrees[j] = j + 1;
}
```

The author added the following while adapting the program:

finalize_count the number of processes that have terminated locally

status a variable used to store information about the MPI function calls

The following definitions are used to tag communication

```
#define JOB_TAG     101
#define ZIP_TAG     102
```

2.5 Porting the CSM for the SPP

The implementation developed by Smith and Thompson [30] was tested on a CRAY T3E parallel computer. Excellent results were reported - with some problem instances they reported super linear speedup. It used a combination of message passing and shared memory. On the CRAY, shared memory is emulated with a custom library (`mpp/shmem.h`). As this architecture was not available to the author, the parts of the program that used this library had to be adapted to make use of platform independent message passing only. MPI was used on a cluster of Linux workstations.

The inter process communication that takes place in the program achieves the following goals:

- distributing the original problem information
- communicating the best (lowest) upper bound for the SPP of the upper bounds found by all nodes

- distributing the basis for the solution found by the master process (to ensure that all slaves continue with the same basis)
- work distribution (distribution of subtrees of the root node of the search tree)
- updating the best known solution
- termination detection
- profiling to measure the success of different fathoming rules

Work distribution, updating the best known solution and termination detection made use of shared memory. These are the parts that were reimplemented to make use of message passing only. In the next section these changes are discussed.

2.6 Work distribution

2.6.1 Using shared memory

At the slaves

A `subtrees` array, shared by all processes, stored the number of the subtree that each process was searching through. Once the search of the subtree has been exhausted, an idle slave process would insert a negative value into the shared array and wait for the value to become positive. A positive number would indicate the next subtree to process.

```
/* send idle indicator to process 0 */
rootNext = -1;
shmem_int_put(subtrees + myId, &rootNext, 1, 0);
/* wait for number of next subtree */
do
    shmem_int_get(&rootNext, subtrees + myId, 1, 0);
while (rootNext < 0);
```

At the master

The master process frequently iterates through the `subtrees` array. If it finds a negative value, it changes it to the number of the next subtree to be processed. The `subtrees` array is directly accessible by the master process and the master therefore need not use the special library functions (`shmem_...`). In the following code, the variable `j` starts at 1, which is the number of the first slave process. The master need not update the entry for itself (process 0), because it rather keeps track of subtrees by means of `rootNext`.

```
/* send a subtree number to each idle process */
for(j = 1; j < numProcs; j++)    {
    if(subtrees[j] < 0) {
        subtrees[j] = rootNext; /* now slave j can continue */
        rootNext++;      /* the next subtree to delegate */
    }
}
```

2.6.2 Using message passing

At the slaves

When becoming idle, a slave process sends a message to the master process asking for a new subtree to process. It makes use of a `JOB_TAG` to make it clear what the message is for. The process blocks until it receives the next subtree to process.

```
MPI_Send(&rcsize, 1, MPI_INT, 0, JOB_TAG, MPI_COMM_WORLD);
//now we wait for the reply from the master process
MPI_Recv(&rootNext, 1, MPI_INT, 0, JOB_TAG, MPI_COMM_WORLD, &status);
```

At the master

Because the master will not know if a slave has asked for a new subtree to process, it has to probe the message queue for messages with the relevant tag (JOB_TAG). If an appropriate message is in the message queue, it will receive the message (remove it from the message queue) and respond with the number of the next subtree to be processed.

Slaves send their current rcsize to enable the master process to determine whether the slave will definitely interpret the master's response as an indication of termination.

```
void send_jobs()
{
    int flag, other_rcsize;
    MPI_Iprobe(MPI_ANY_SOURCE, JOB_TAG,
            MPI_COMM_WORLD, &flag, &status);
    while (flag) {  //while there is a message in the queue
        //now we remove the message from the message queue
        MPI_Recv(&other_rcsize, 1, MPI_INT, MPI_ANY_SOURCE,
                JOB_TAG, MPI_COMM_WORLD, &status);
        //reply to the sender of the message
        MPI_Send(&rootNext, 1, MPI_INT, status.MPI_SOURCE,
                JOB_TAG, MPI_COMM_WORLD);

        //for termination detection:
        if (rootNext > other_rcsize) finalize_count++;
        rootNext++;

        //test again
        MPI_Iprobe(MPI_ANY_SOURCE, JOB_TAG,
                MPI_COMM_WORLD, &flag, &status);
    }
}
```

2.7 Updating the best known solution

2.7.1 Using shared memory

Each process has a shared array storing the best known objective function value per process. When process i finds a solution that was better than the best known to it, it updates the value in the i-th position in the array. It also updates the i-th position in the array of each of the other processes. When the other processes iterate through their local array, they compare each of the values with their previously known best and update it if necessary.

Code executed when finding a new objective function value:

```
for(i = 0; i < numProcs; i++)
    if(i != myId) shmem_int_put(zIPs + myId, &zIP, 1, i);
zIPs[myId] = zIP;
```

Code executed to update best known objective function value:

```
do {
    flag = 0;
    min = zIP;
    for(j = 0; j < numProcs; j++)
        if(zIPs[j] < min) min = zIPs[j];
    if(min < zIP) {
        flag = 1;
        zIP = min;
        rcsize = reduce(zIP - zLP - 1.0 + EPS);
    }
} while (flag);
```

2.7.2 Using message passing

When a process finds a solution that is better than the best known to it, it sends a message to all processes excluding itself. It sends the new objective function value and uses ZIP_TAG to indicate that the purpose of the message is to update the zIP variable.

Code executed when finding a new objective function value:

```
for (i = 0; i < numProcs; i++)
    if (i != myId) MPI_Send(&zIP, 1, MPI_INT, i,
                       ZIP_TAG, MPI_COMM_WORLD);
```

Code executed to update best known objective function value:

```
void update_zIP()
{
    int flag, min;
    MPI_Iprobe(MPI_ANY_SOURCE, ZIP_TAG,
            MPI_COMM_WORLD, &flag, &status);
    while (flag) {  //while there is a message in the queue
        // now we remove the message from the message queue
        MPI_Recv(&min, 1, MPI_INT, MPI_ANY_SOURCE,
            ZIP_TAG, MPI_COMM_WORLD, &status);
        if(min < zIP) {
            flag = 1;
            zIP = min;
            rcsize = reduce(zIP - zLP - 1.0 + EPS);
        }
        //test again
        MPI_Iprobe(MPI_ANY_SOURCE, ZIP_TAG,
                MPI_COMM_WORLD, &flag, &status);
    }
}
```

2.8 Termination detection

2.8.1 Using shared memory

As discussed above, a slave sets an entry in the `subtrees` array to a negative value to indicate that it has finished processing the subtree it was busy with. When the master process terminates locally, it iterates through the `subtrees` array. When it finds a negative value, it changes the value to a positive, but invalid value. The slave will realise that the invalid value indicates termination. By counting the number of slaves to which it replied, the master can detect global termination.

```
/* wait for all processes to finish */
do {
    count = 1;
    for(j = 1; j < numProcs; j++) {
        if(subtrees[j] < 0) { /* ``answer'' idle processes */
            subtrees[j] = rootNext;
            count++;
        } else if(subtrees[j] >= rootNext)
            count++
    }
} while (count < numProcs);
```

2.8.2 Using message passing

As discussed above, a slave would send a message with a `JOB_TAG` to the master to ask for the next subtree to process. If the returned message contains the number of a valid subtree to process, the slave continues. Otherwise the slave terminates locally. The master counts the number of invalid subtrees it sends out and detects global termination in that way. Note that the variable `finalize_count` might already be positive when the master terminates locally as the master could already detect local termination of slaves during work distribution.

```
finalize_count++;    //master process finished
rootNext = -1;       //to have definite indication of termination
while (finalize_count < numProcs) {
    MPI_Recv(&j, 1, MPI_INT, MPI_ANY_SOURCE, JOB_TAG,
                MPI_COMM_WORLD, &status);
    MPI_Send(&rootNext, 1, MPI_INT, status.MPI_SOURCE, JOB_TAG,
                MPI_COMM_WORLD);
    finalize_count++;
}
```

2.9 Results

This section reports on the results obtained with the CSM on a cluster of workstations. The program was tested with problems described in table 1 of Hoffman and Padberg [16]. The problems were pre-processed as suggested in [16].

The testing environment was a cluster of identical single processor workstations, each with the following configuration:

Hardware	
Processor	Pentium III 730MHz
Physical memory	128 MBytes
Interconnection	100 Mbits/s switched Ethernet
Software	
Operating System	Linux 2.4.20
File system	Network File System
	(file server not part of the cluster)
MPI Implementation	LAM/MPI 6.5.9
C compiler	GNU C Compiler 3.3

In the discussed implementation, solving the LP relaxation is not parallelised. By Amdahl's law (see [1]), if f is the fraction of the runtime on a single processor that cannot be parallelised, the maximum speedup attainable is $1/f$. Below, only those results where a speedup of more than 5 is possible, are reported. Execution time is the average of three runs and is measured in seconds. Utilisation is the ratio of actual processor time used, to available processor time.

		Processors				
Problem		1	2	4	8	16
Hp10757	avg time	390.850	125.875	41.542	33.874	22.744
	speedup		3.11	9.41	11.54	17.19
	utilisation		1.000	0.999	0.979	0.860
Hp43749	avg time	16.378	4.922	4.387	4.403	4.866
	speedup		3.33	3.73	3.72	3.37
	utilisation		1.000	0.999	0.998	0.946
nw04	avg time	36.735	18.187	4.740	3.781	4.165
	speedup		2.02	7.75	9.72	8.82
	utilisation		1.000	0.997	0.960	0.868

Superlinear speedup is observed in many of these cases, although the speedup is not as big as those achieved by Smith and Thompson [30] on the CRAY T3E. For more information about speedup when solving Integer Programming problems with a cluster of workstations, see [6] and [12]. For information on anomalies in parallel processing, see [19].

2.10 Conclusion

The CSM is an effective method for solving Set Partitioning Problems. Testing the CSM on a cluster of workstations confirmed the results reported in

[30], namely that superlinear speedups are obtained in some instances.

Chapter 3

The Relaxed 2-Matching Problem

A basic introduction to the TSP was given in Chapter 1. In this chapter a mathematical formulation is given. The formulation involves decision variables and constraints on those variables and includes an objective function which is minimised. Most techniques for solving the TSP make use of some *relaxation* (see [21] for details on different relaxations). A relaxation is a similar problem with fewer constraints, that is easier to solve and that can help to guide the search for the optimal solution. The choice of relaxation for this thesis, the Relaxed 2-Matching Problem (RMP2), is presented here as well as the techniques to solve it.

The TSP can be formulated using a complete, undirected, weighted graph. Suppose G is a complete graph with node set $N = \{1, 2, ..., n\}$ and edge set $E = \{1, 2, ..., m\}$ where an edge $e = (i, j)$ in E connects two nodes i and j in N and has a cost c_e. Each node in the graph represents a city that the salesman has to visit. Each edge represents the connection between two cities. The edge weight represents the cost associated with traveling between the two cities connected by the edge. As the graph is complete, there are $m = n(n-1)/2$ edges in the graph.

Definitions

A (simple) *path* is a sequence of distinct edges such that any two successive edges in the sequence are incident to a common node, but at most two edges in the path are incident to any node. A subgraph is *connected* if it contains a path between any two nodes in the subgraph. A *cycle* is a path with at least three edges in which the first and last edge have a common node. A *1-tree* is a connected subgraph containing exactly one cycle. A *2-matching* is a subgraph containing exactly two edges incident to each node. A *tour* of G is a cycle with n edges while a *subtour* of G is a cycle with less than n edges. (A tour is both a 2-matching and a 1-tree.) The cost of a subgraph is the total cost of the edges in the subgraph. The *TSP* is the problem of finding the minimum cost tour in G.

3.1 Formulation

3.1.1 The TSP

For each edge $e \in E$ a 0-1 variable x_e is introduced. The value indicates inclusion (1) or exclusion (0) from the current solution. For each node in N a constraint is added that limits the chosen edges incident to that node to two edges (representing the 2-matching). This corresponds to the two edges that connect that node to the rest of the graph (the “roads” the salesman uses for entering and leaving the city). Considering the definition of the TSP given in the previous section, the TSP can be formulated mathematically as follows:

$$
\begin{array}{lll}
\textit{Minimise} & \mathbf{cx} & \textit{(cost)} \\
\textit{Subject to} & A\mathbf{x} = \mathbf{b} & \textit{(2-matching constraints)} \\
 & \mathbf{x} \in X & \textit{(1-tree constraints)} \\
 & \mathbf{x} \in \{0,1\}^m & \textit{(integer constraints and bounds)},
\end{array}
$$

where $\mathbf{c}$ is an m-dimensional vector of positive costs, A is the $n \times m$ node-edge incidence matrix of the graph G, $\mathbf{b}$ is an n-dimensional vector of twos and X is a polyhedron of which the extreme points are the 1-trees in the graph G (see Held and Karp [14]). The two ones in $\mathbf{a}_e$ ($\mathbf{a}_e$ is the e-th column of A) represent the two nodes connected by edge e.

Strictly speaking, only the optimal tour is a solution for the TSP. In this dissertation, however, any subgraph that satisfies the constraints in this formulation (a tour) will informally be called a solution. In searching for the optimal tour, we use an upper and lower bound on the objective function value to guide our search. Details on how the bounds are used will follow later. An upper bound can be computed using a heuristic and can be updated as we find increasingly cheaper tours. To find a lower bound, we use a relaxation of the TSP formulation given above.

3.1.2 The 2-Matching Problem

A possible relaxation is to relax the 1-tree constraints. The 2-Matching Problem (MP2) is formulated as an integer program (IP) without the 1-tree constraints:

$$\begin{array}{ll} \textit{Minimise} & \mathbf{cx} \\ \textit{Subject to} & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \in \{0,1\}^m, \end{array}$$

where $\mathbf{c}$, A, and $\mathbf{b}$ have the same meanings as before.

A solution to the MP2 (a 2-matching, as defined earlier) might, however, not be a solution to the TSP, as a solution to this IP could contain subtours (two or more disjoint cycles). As a cycle contains at least three nodes, this can only happen where a graph has six or more nodes. Solutions containing subtours have to be eliminated, as they don not solve the TSP. Solving the MP2 gives a lower bound on the cost of the optimal solution to the TSP, but the MP2 remains a difficult problem to solve.

3.1.3 The Relaxed 2-Matching problem

The Relaxed 2-Matching problem (RMP2) is a relaxation of the MP2 (and therefore also the TSP) where the integrality constraints on the variables are relaxed to allow arbitrary values within the interval [0, 1]. The formulation for this relaxation is as follows:

$$\begin{array}{rl} \textit{Minimise} & \mathbf{cx} \\ \textit{Subject to} & A\mathbf{x} = \mathbf{b} \\ & \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}, \end{array}$$

where $\mathbf{c}$, A, and $\mathbf{b}$ have the same meanings as before.

Naturally, a solution to the RMP2 will also not necessarily be a tour (but could be). The lower bound obtained from solving the RMP2 could also be lower than that obtained from the MP2. The RMP2 is however much easier to solve than the MP2 because the simplex method can be used. The RMP2 is the relaxation that will be used in this dissertation. The rest of this chapter continues with material necessary for solving the RMP2.

3.2 Solving the Relaxed 2-Matching problem

Solving the RMP2 is the first step taken to solve the TSP. The cost of the optimal solution to the RMP2 is used as a lower bound on the objective function value of the TSP. The standard simplex method can be used to solve the RMP2, but the RMP2 has a special structure that allows for a more efficient solution.

A generalised network problem is a special Linear Programming case where each column of the A matrix has at most two non-zero entries (see Kennington and Helgason [18], chapter 5). Each of the columns of the matrix A in the definitions above contain two ones and $n-2$ zeros. The RMP2 can therefore be seen as a generalised network problem. This special structure has been exploited to implement a very efficient simplex algorithm for the

RMP2 (see Smith et al [28]).

The method involves the same basic steps as the standard simplex algorithm: selecting an entering variable, doing a ratio test to select a leaving variable, and pivoting. The following sections explain some of the techniques used to implement these steps very efficiently for the RMP2. For a complete discussion see Meyer [22].

Definitions

A *basis structure* (B, L, U) partitions the edge set E such that the $n \times n$ matrix formed by the columns $\mathbf{a}_e$, $\forall e \in B$ is a basis for A. A *basis graph* is a spanning subgraph of G with edge set B. The edges in B are called the *basic edges* and the variables in $\mathbf{x}$ that correspond to the basic edges are the *basic variables* and are denoted by $\mathbf{x}_B$. Similarly, the costs of the basic edges are denoted by $\mathbf{c}_B$.

$L \cup U$ contains the edges corresponding to the non-basic variables. Because of the upper bounds that are imposed on the variables, the implementation of the simplex method used here, allows non-basic variables to assume non-zero values. Variables corresponding to edges in L are non-basic at the lower bound 0 and variables corresponding to edges in U are non-basic at the upper bound 1.

The *basic solution* corresponding to a basis structure (B, L, U) is obtained by setting $x_e = 0$ for all $e \in L$ and $x_e = 1$ for all $e \in U$, and solving

$$B\mathbf{x}_B = \mathbf{b} - \sum_{j \in L} x_e \mathbf{a}_e - \sum_{e \in U} x_e \mathbf{a}_e \tag{3.1}$$

for $\mathbf{x}_B$. Here B is additionally used to refer to the basis.

3.2.1 Properties of the basis

The 1-tree structure is important when discussing the RMP2. It's properties are used in sections to come for certain calculations and algorithms. It was noted earlier that a tour is both a 2-matching and a 1-tree.

When a variable has a value of 1, it means that the edge the variable corresponds to, is included in the tour (or current solution to the relaxation). This of course applies to non-basic variables as well. Therefore, the basis graph will not necessarily contain all the edges included in the current solution (with corresponding variables at value 1), and also, could contain edges that are not part of the current solution (with corresponding variables at value 0).

With the RMP2, the variables' integrality restrictions are relaxed. Therefore, it is possible for a variable to assume a value between zero and one. In such a case the meaning of the variable value is lost and the particular solution will not be a solution to the TSP. In the algorithm discussed here, this can only happen with basic variables.

It should therefore be noted that the basis graph for the RMP2 consists of one or more 1-trees (see Proposition 5.9 in [18]). A basis graph consisting of more than one 1-tree, could still correspond to a solution to the TSP when one or more basic edges are not part of the tour and some non-basic edges are part of the tour. In a solution to the TSP the edges that correspond to all the variables with a value of one (basic and non-basic) will form both a tour and a 1-tree.

Leenen [21] points out that the cycle on any 1-tree in a basis graph contains an odd number of nodes. This is an important attribute when we calculate the dual values in a following section.

Solving for $\mathbf{x}_B$ in 3.1, shows the role of B^{-1} in calculating $\mathbf{x}_B$. Leenen also indicated that the odd number of nodes in the cycle of a 1-tree has the effect that each element of B^{-1} is an integer divided by 2. This important result causes values in $\mathbf{x}$ to be one of $\{0, \frac{1}{2}, 1\}$.

When a basic variable $x_e = \frac{1}{2}$, it imposes restrictions on the values of the variables corresponding to edges incident to the same node as e in the 1-tree. To satisfy the equations of the RMP2, these variables which correspond to edges in the cycle incident to the same node as e have to be fractional too.

For each of these edges that are incident to the same node as e, the same argument can be applied. Because of the structure of the basis graph, $x_e = \frac{1}{2}$ is only possible if e is part of a cycle in the basis graph. In such a case $x_e = \frac{1}{2}$ for all e in that particular cycle. If x_e were to be fractional outside the cycle of the 1-tree, the equations of the RMP2 would only be satisfiable up to the leaf nodes of the 1-tree, at which point it would not be possible anymore. At each equation of the RMP2 involving a fractional x_e, at least one other variable has to be fractional too. The edges corresponding to these fractional basic variables are incident to a common node. The equation of the RMP2 corresponding to the second node adjacent to this common node will cause another variable to be fractional. This process can be repeated up to the leaf nodes of the 1-tree, but at the leaf nodes the equation will not be able to force another variable to be fractional.

This is a very important result: as $2x_e$ is an integer, it makes it possible to use integers to store variable values by storing $2x_e$ $\forall e \in B$. Computation is faster than with floating point variables. Integers possibly also consume less memory.

In a following section where dual values and reduced costs are defined, we will see that they too can be represented with integers.

3.2.2 Representing the basis graph

The basis graph is represented by the *augmented predecessor index method* (see Glover et al [10]). For each node indices are held for a parent, a sibling and a child node as well as the distance from the cycle. For Figure 3.1, the following table indicates the values at each node:

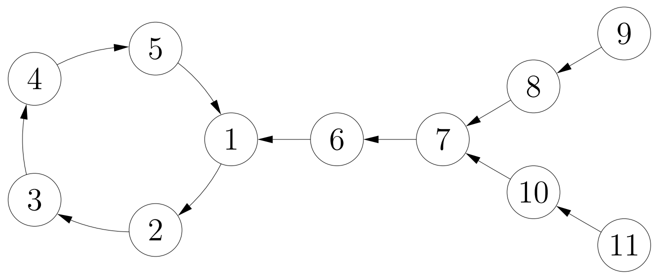

Figure 3.1: A 1-tree with arrows indicating parent relationships

Node	**Parent**	**Sibling**	**Child**	**Distance**
1	2	-	6	0
2	3	-	1	0
3	4	-	2	0
4	5	-	3	0
5	1	-	4	0
6	1	5	7	1
7	6	-	10	2
8	7	-	9	3
9	8	-	-	4
10	7	8	11	3
11	10	-	-	4

For several calculations the basis graph needs to be traversed (usually partially), with the needed traversal pattern usually being from a node towards the cycle of the 1-tree, using the path formed by following parent indices.

3.2.3 Dual values and reduced costs

Just as with the standard simplex method, the reduced costs are used to select the entering variable. In the standard simplex method the calculation of the reduced costs involves expensive matrix multiplication. In the RMP2, our knowledge of the special structure of the A matrix allows us to vastly simplify the computation of the reduced costs. To this end, we define the n-vector $\mathbf{u}$.

The *dual values* $u_j \ \forall j \in N$ are defined as the solution to the set of *dual value equations*:

$$u_i + u_j = c_e \quad \forall e = (i,j) \in B.$$

The dual values are used to define the *reduced costs*

$$v_e = c_e - u_i - u_j \quad \forall e = (i,j) \in E \tag{3.2}$$

or alternatively

$$v_e = c_e - \mathbf{u}\mathbf{a}_e \quad \forall e \in E$$

The reduced costs for basic variables are, of course, 0:

$$\begin{aligned} v_e &= c_e - (u_i + u_j) \\ &= c_e - c_e \\ &= 0 \end{aligned}$$

The dual values can easily be calculated by solving the set of equations in a way that will be illustrated with a concrete example:

Firstly we solve for the dual values of the nodes on the cycle of a 1-tree in the basis graph. For example, in a 1-tree with three nodes a, b and c that follow consecutively on each other on the cycle, we use the dual value equations for edges $(a,b), (b,c)$ and (c,a). We alternately add and subtract the dual value equations for all the edges on the cycle in the order in which they are connected, which allows us to solve for one of the dual values:

$$\begin{aligned} (u_a + u_b) - (u_b + u_c) + (u_c + u_a) &= c_{ab} - c_{bc} + c_{ca} \\ 2u_a &= c_{ab} - c_{bc} + c_{ca} \end{aligned}$$

Because there is always an odd number of nodes (and therefore edges) in a cycle of a 1-tree in the basis graph of the RMP2, a summation of the dual value equations such as the one above, will always simplify to allow one to solve for a single dual value.

We can now use the dual value obtained (u_a in the example above) to solve for another dual value by substitution in a dual value equation that contains both dual values (for example $u_a + u_b = c_{ab}$), and in this fashion the dual values for all nodes on the 1-tree can be solved. If there is more than one 1-tree, the same technique can be used to calculate the dual values on all the 1-trees.

When all dual values are computed, the calculation of reduced costs is straight forward. It will be shown later that even the dual values need not be recalculated after each pivot, but that they can be updated efficiently.

In the previous section it was indicated that variable values can be stored as integers. The dual values and reduced costs can also be stored as integers, but for an unrelated reason: if the edge costs are given as integers, the first dual value calculated on the cycle of a 1-tree (u_a in the example) is the sum of (integer) costs divided by two. $2u_a$ is therefore an integer. It can similarly be shown to be true for all dual values and also for the reduced costs. We will use the same technique used with the variable values to store dual values and reduced costs as integers.

3.2.4 An initial basic feasible solution

The first step in solving the RMP2 is to construct an initial basic feasible solution. Any tour can provide the necessary starting basis structure, but an initial tour that is closer to the optimum solution will reduce the number of pivots necessary to reach optimality.

A heuristic is therefore used to find an initial solution: nodes 1, 2 and 3 are included in a small initial cycle. Each of the nodes $4, 5, ..., n$ are then sequentially inserted into the cycle in the cheapest way. If n is odd, the resultant tour can be used as a starting basis structure. If n is even, the basis structure needs to altered to ensure it has all the properties mentioned in the previous sections.

In such a case, the structure is altered around three consecutive nodes in the tour, say a, b and c. The edges (a, b) and (b, c) are currently included in the basis structure. If the cycle is shortened by inserting (a, c), a cycle is obtained with an odd number of nodes. Now one edge has to be removed, as B can only contain n edges. (b, c) can therefore be made non-basic by moving it to U, and then a valid basis structure is obtained. All basic variables will have initial values of one, except x_{ac} that will start out with a value of zero.

After obtaining a starting basic feasible solution, the simplex method can be used to move from this solution to increasingly better feasible solutions. Definitions for primal and dual feasibility are needed in order to describe when it is possible to proceed to a better basic feasible solution.

3.2.5 Primal and dual feasibility

For any RMP2 solution and basis structure (B, L, U), we have

$$\begin{aligned}
A\mathbf{x} &= B\mathbf{x}_B + \sum_{e\in L} x_e\mathbf{a}_e + \sum_{e\in U} x_j\mathbf{a}_e = \mathbf{b} \\
\mathbf{x}_B &= B^{-1}(\mathbf{b} - \sum_{e\in L} x_e\mathbf{a}_e - \sum_{e\in U} x_e\mathbf{a}_e) \qquad (3.3)\\
&= B^{-1}\mathbf{b} - \sum_{e\in L} x_e\mathbf{y}_e - \sum_{e\in U} x_e\mathbf{y}_e \\
&= B^{-1}\mathbf{b} - \sum_{e\in U} \mathbf{y}_e - \sum_{e\in L} x_e\mathbf{y}_e - \sum_{e\in U}(x_e - 1)\mathbf{y}_e \\
&= \mathbf{d} - \sum_{e\in L} x_e\mathbf{y}_e - \sum_{e\in U}(x_e - 1)\mathbf{y}_e \qquad (3.4)
\end{aligned}$$

where $\mathbf{y}_e = B^{-1}\mathbf{a}_e$ and $\mathbf{d} = B^{-1}\mathbf{b} - \sum_{e\in U}\mathbf{y}_e$. In the basic solution corresponding to the basis structure (B, L, U), x_e is set to 0 for all $e \in L$ and x_e is set to 1 for all $e \in U$ and, therefore, $\mathbf{d}$ contains the values of the basic variables.

The objective function z can be written as

$$\begin{aligned} z &= \mathbf{cx} \\ &= \mathbf{c}_B\mathbf{x}_B + \sum_{e\in L\cup U} c_e x_e \end{aligned}$$

Substituting $\mathbf{x}_B$ using equation (3.3):

$$\begin{aligned} z &= \mathbf{c}_B(B^{-1}(\mathbf{b} - \sum_{e\in L} x_e\mathbf{a}_e - \sum_{e\in U} x_e\mathbf{a}_e)) \ + \sum_{e\in L} c_e x_e + \sum_{e\in U} c_e x_e \\ &= \mathbf{c}_B\mathbf{d} - \mathbf{c}_B\sum_{e\in L} x_e B^{-1}\mathbf{a}_e - \mathbf{c}_B\sum_{e\in U}(x_e-1)B^{-1}\mathbf{a}_e \ + \sum_{e\in L} c_e x_e + \sum_{e\in U} c_e(x_e-1) + \sum_{e\in U} c_e \\ &= \mathbf{c}_B\mathbf{d} + \sum_{e\in L} c_e x_e - \mathbf{c}_B\sum_{e\in L} x_e B^{-1}\mathbf{a}_e \ + \sum_{e\in U} c_e(x_e-1) - \mathbf{c}_B\sum_{e\in U}(x_e-1)B^{-1}\mathbf{a}_e + \sum_{e\in U} c_e \\ &= \mathbf{c}_B\mathbf{d} + \sum_{e\in L} x_e(c_e - \mathbf{c}_B B^{-1}\mathbf{a}_e) \ + \sum_{e\in U}(x_e-1)(c_e - \mathbf{c}_B B^{-1}\mathbf{a}_e) + \sum_{e\in U} c_e \\ &= \mathbf{c}_B\mathbf{d} + \sum_{e\in L} x_e(c_e - \mathbf{ua}_e) \ + \sum_{e\in U}(x_e-1)(c_e - \mathbf{ua}_e) + \sum_{e\in U} c_e \\ &= \mathbf{c}_B\mathbf{d} + \sum_{e\in U} c_e + \sum_{e\in L} v_e x_e + \sum_{e\in U} v_e(x_e-1) \\ &= z_0 + \sum_{e\in L} v_e x_e + \sum_{e\in U} v_e(x_e-1) \end{aligned}$$

where $z_0 = \mathbf{c}_B\mathbf{d} + \sum_{e\in U} c_e$, $\mathbf{u} = \mathbf{c}_B B^{-1}$ and $v_e = c_e - \mathbf{ua}_e \ \forall e \in L\cup U$. The elements of the vectors $\mathbf{u}$ and $\mathbf{v}$ are respectively the *dual values* and the *reduced costs* defined earlier.

A basis structure (B, L, U) is *primal feasible* if $\mathbf{0} \leq \mathbf{x}_B \leq \mathbf{1}$ in the corresponding basic solution. A basis structure (B, L, U) is *dual feasible* if $(v_e \geq 0 \ \forall e \in L)$ and $(v_e \leq 0 \ \forall e \in U)$.

When a basis structure is both primal and dual feasible, the corresponding basic solution is optimal.

3.2.6 Selecting an entering variable

While a basis structure is not dual feasible, we can select a variable to enter the set of basic variables (hereafter referred to as entering the basis). Selection of an entering variable is mostly the same as with the standard simplex method: a dual infeasible non-basic variable is chosen to enter the basis.

With the RMP2, a basis structure can be dual infeasible when a variable in L has a negative reduced cost, or when a variable in U has a positive reduced cost. The absolute value of the reduced cost, $|v_e|$ for all dual infeasible variables e are compared and the largest value indicates the variable to enter the basis.

3.2.7 Selecting a leaving variable

Now that an entering variable has been selected, a variable must be chosen to leave the set of basic variables (hereafter referred to as leaving the basis) in such a way as to ensure that primal feasibility is maintained. To this end the ratio test is used, but in a very specialised way to exploit the special structure of the RMP2.

Suppose the variable corresponding to edge $e = (i, j)$ has been selected as entering variable. We can see the effect of changing the value of the variable x_e in equation (3.4). To perform the ratio test, we need to solve for $\mathbf{y}$ in $B\mathbf{y} = \mathbf{a}_e$, where B is the current basis. For detail on the ratio test for the RMP2, see Leenen [21]. The technique will only be illustrated here by means of a few graphical examples (see Figures 3.2 to 3.5 on pp. 35-36). The direction of edges in these graphs indicates parent relationships. The value displayed on an edge (a, b) is y_{ab}.

In all cases the basis graph is traversed from nodes i and j towards the cycle of the 1-tree(s) they are part of. The values of $\mathbf{y}$ corresponding to the first edge on these two paths (the two edges incident on i and j) are +1 for both (see [21]). From there on the values alternate between -1 and +1 on the path towards the cycle. If the two paths meet (i and j are part of

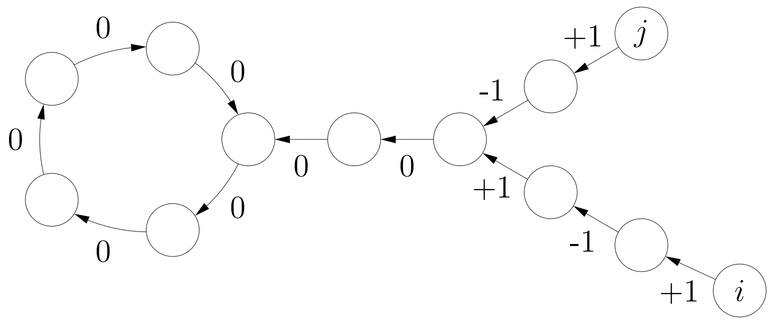

Figure 3.2: Calculating **y**: Two paths meet and cancel each other out

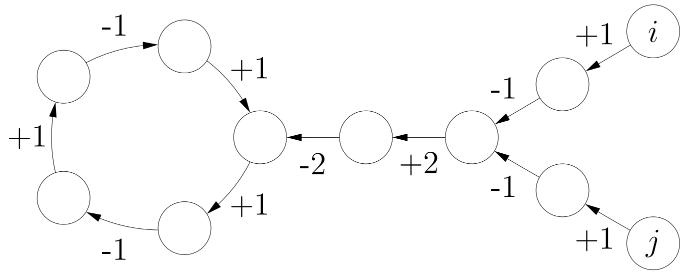

Figure 3.3: Calculating **y**: Non-zero values throughout a 1-tree

the same 1-tree), the results from there onwards are the cumulative effect of the calculations for i and j separately. The entries for **y** corresponding to the edges on the cycle have half of the value they would have in a path outside the cycle. Parent links are followed until the node where the cycle was entered into, is reached for the second time. Entries in **y** that correspond to edges that were never traversed are all 0.

The values for **y** are actually never stored, but are used for the ratio test as they are calculated. At each edge e in the traversal, d_e/y_e is calculated and compared with the minimum ratio found so far. The ratio test is stopped if a ratio of 0 is found, as that is the smallest attainable ratio.

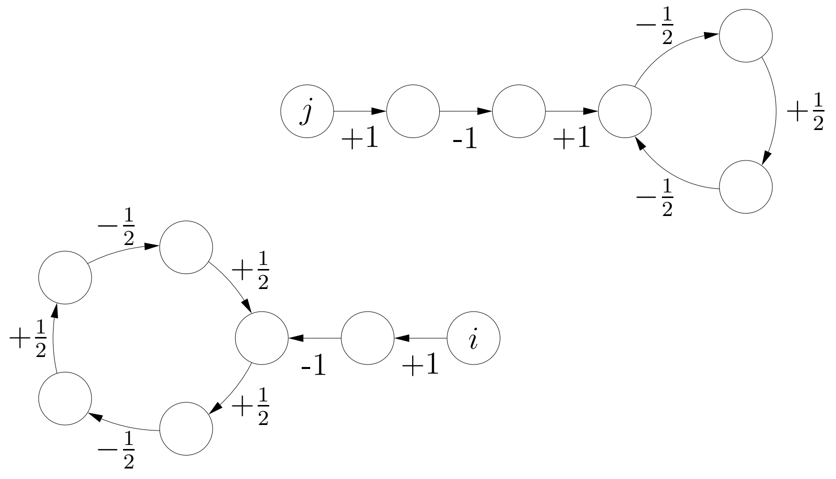

Figure 3.4: Calculating $\mathbf{y}$: i and j in separate 1-trees

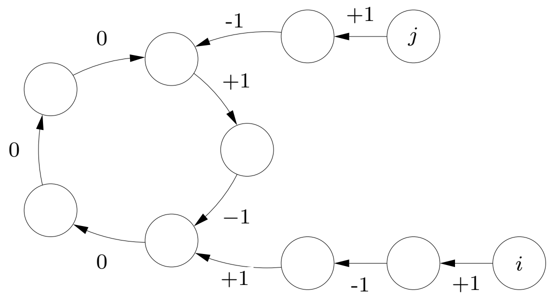

Figure 3.5: Calculating $\mathbf{y}$: Paths meeting the cycle in different places

3.2.8 Pivoting

Now that both the entering and leaving variables have been selected, we can proceed to change the basis. The minimum ratio can be added to the entering variable value (or subtracted from it if the entering variable was in U). The dual values need to be updated as well.

Not all dual values need to be updated; once again we will use traversal of the basis graph to aid us in updating the dual values efficiently. After the basis change with $e = (i, j)$ that entered the basis, the subgraph with edge set $B - \{e\}$ contains a maximal tree in the 1-tree that contained e.

Let us assume that i is the node furthest away from the cycle (ties are broken arbitrarily). The nodes of this tree can be separated into two sets:

- $P = \{k \mid k \text{ and } i \text{ are connected by a path of even length}\}$
- $Q = \{k \mid k \text{ and } i \text{ are connected by a path of odd length}\}$

The values for u_i after the change, u'_i, can be calculated as follows:

$$u'_i = \begin{cases} u_i + \Delta & \forall i \in P \\ u_i - \Delta & \forall i \in Q \\ u_i & \text{otherwise} \end{cases}$$

$$\text{where } \Delta = \begin{cases} r_e/2 & e \text{ is in a cycle of the basis graph} \\ r_e & \text{otherwise} \end{cases}$$

This can be proven (see [21] p. 15) by showing that the dual value equations are still satisfied after the change.

3.3 Conclusion

A formulation of the TSP was presented here along with the relaxation that is used to obtain a lower bound on the cost of the optimal TSP tour. The

RMP2 was formulated as a linear programming problem that can be solved by the simplex method.

The version of the simplex method discussed here, exploits properties that exist in the basis graph to solve the RMP2 very efficiently. The fact that the 1-trees in the basis graph always contain an odd number of nodes, makes it possible to store and manipulate variable values as integers. Instead of calculating the reduced costs by means of expensive matrix multiplication, they are calculated by first calculating the dual values, which can be computed easily by traversing the basis graph. Dual values and reduced costs can be stored as integers too.

Smith et al [28] reported speedups of several orders of magnitude using this simplex method, and their results suggested that it is asymptotically faster than the commercial solver with which they compared their results.

After solving the RMP2, we can proceed to solve the TSP. In the next chapter the dual simplex method, which is necessary for solving the TSP, is presented.

Chapter 4

The dual simplex method for the Relaxed 2-Matching Problem

In this chapter the dual simplex method for the Relaxed 2-matching Problem (RMP2) is discussed. The implementation of the dual simplex method discussed here, is based on the work of Leenen [21] and Geldenhuys [9]. As mentioned before, they used the dual simplex method as part of the Branch-and-Cut method. This implementation is part of the Column Subtraction Method and is, therefore, based on different assumptions.

Winston [31] gives three uses for the dual simplex method:

- Solving an LP for which a dual feasible basic solution is initially available
- Reoptimising when loosing primal feasibility after a constraint has been added to an LP
- Reoptimising when loosing primal feasibility after changing a right-hand side of an LP

The Branch-and-Cut method used by Leenen and Geldenhuys necessitated the consideration of additional constraints (aka side constraints). Primal infeasibility arose because of the extra constraints added to the RMP2. This necessarily influenced the implementation in, for example, the calculation of the basis and the storing of the variable values. In the implementation presented here, side constraints are not used, and therefore some of the restrictions of Leenen and Geldenhuys' implementation are not present. Primal infeasibility arises because of the right-hand side of the RMP2 changing. The structure that is exploited in the primal simplex method for the RMP2, can then be exploited in the dual simplex method as well.

Given a dual feasible basic solution, the dual simplex method obtains primal feasibility. The basic method is to find a primal infeasible basic variable that will leave the basis, select a nonbasic variable to enter the basis in such a way that dual feasibility is maintained and to pivot. These steps will be discussed in this chapter.

4.1 Selecting the leaving variable

For each primal infeasible basic variable x_{Bj}, a deviation D_j (from primal feasibility) is computed: For a variable x_{Bj} with negative value d_j, the deviation D_j is simply $-d_j$ (the magnitude of the negative value) and for a variable x_j with value d_j greater than one, $D_j = d_j - 1$. A primal infeasible basic variable x_{Br}, with greatest deviation D_r, is chosen to leave the set of basic variables. The index r denotes the index in B of the leaving variable in the following sections.

4.2 Selecting the entering variable

In the previous section we saw how a leaving variable is selected. An entering variable must be chosen to replace x_{Br} in the set of basic variables. Because of the upper bounds on the variables, an adapted form of the ratio test is used to select the entering variable.

In the simplex method without upper bounds on variables, a basic variable is primal infeasible when it is negative. To restore primal feasibility in such a case, the value of an infeasible variable is increased. In such cases, the ratio test considers the ratio v_e/y_{re} for all non-basic edges e, where y_{re} is the component of the vector $\mathbf{y}_e = B^{-1}\mathbf{a}_e$ which corresponds to x_{Br} (the leaving basic variable). With a dual feasible basis, the reduced costs ($\mathbf{v}$) are all non-negative. To maintain dual feasibility the entering variable x_k is selected in such a way that $v_k/y_{rk} = max\{v_e/y_{re} \mid y_{re} < 0\}$. This is necessary and sufficient to maintain dual feasibility.

However, in the RMP2 we have 1 as upper bound on all variables, and the basic variable x_{Br} may be primal infeasible either because $d_r < 0$ or $d_r > 1$. To achieve primal feasibility, a basic variable must increase if it is negative, or decrease if it exceeds its upper bound. Furthermore, the normal requirement of just ensuring that reduced costs remain non-negative has to be extended according to the definition of dual feasibility (see p. 33). As the infeasible basic variable might have to decrease to obtain primal feasibility, and as the reduced costs of non-basic variables at the upper bound (1) has to remain non-positive, the set of variables that is considered for the ratio test is constructed differently with the RMP2 than with an LP without upper bounds on variables.

From equation (3.4) we can see the change in a single variable x_{Br}. The equation

$$x_{Br} = d_r - \sum_{e \in L} x_e y_{re} - \sum_{e \in U} (x_e - 1) y_{re}$$

shows how a change in non-basic variables influences the basic variables. Using this we can construct the set of non-basic variables that can be considered for entering the basis. If the entering non-basic variable x_k is in L, it will increase. If it is in U, it will decrease.

- If x_{Br} *has to increase* ($d_r < 0$), a non-basic variable x_e, $e \in L$ can only be considered for entering the basis if $y_{re} < 0$, because $x_e y_{re}$ is subtracted from x_{Br}. For $e \in U$, y_{re} must be positive to be able to increase x_{Br} as $(x_e - 1)y_{re}$ is subtracted from x_{Br}.

The set of indices of variables that can be considered for entering the basis when $d_r < 0$, is $S_1 = \{e \in L \mid y_{re} < 0\} \cup \{e \in U \mid y_{re} > 0\}$.

- If x_{Br} *has to decrease* ($d_r > 1$), a non-basic variable x_e, $e \in L$ can only be considered for entering the basis if $y_{re} > 0$, because $x_e y_{re}$ is subtracted from x_{Br}. For $e \in U$, y_{re} must be negative to be able to decrease x_{Br} as $(x_e - 1)y_{re}$ is subtracted from x_{Br}.

  The set of indices of variables that can be considered for entering the basis when $d_r > 1$, is $S_2 = \{e \in L \mid y_{re} > 0\} \cup \{e \in U \mid y_{re} < 0\}$.

4.3 Ratio test

The values for v_e after pivoting, v'_e can be calculated as follows:

$$v'_e = v_e - y_{re}(v_k/y_{rk})$$

where k is the index of the entering column.

To maintain dual feasibility, we must ensure that $v_e \geq 0 \; \forall e \in L$ and $v_e \leq 0 \; \forall e \in U$ after the change. To this end, knowledge of the sign of v_k/y_{rk} is important. If $k \in L$, $v_k \geq 0$ because of the dual feasibility requirement. If $k \in U$, $v_k \leq 0$. The sign of y_{rk} depends on the value of x_{Br}, as outlined above.

To ensure that dual feasibility is maintained, the following cases have to be considered:

- If $d_r < 0$, then $v_k/y_{rk} \leq 0$
  - For all $e \in L$: if $y_{re} \geq 0$, v_e cannot decrease and will therefore remain non-negative, else v_e can only remain non-negative if $v_e/y_{re} \leq v_k/y_{rk}$.
  - For all $e \in U$: if $y_{re} \leq 0$, v_e cannot increase and will therefore remain non-positive, else v_e can only remain non-positive if $v_e/y_{re} \leq v_k/y_{rk}$.

Therefore, when $d_r < 0$, $v_k/y_{rk} = max\{v_e/y_{re} \mid e \in S_1\}$.

- If $d_r > 1$, then $v_k/y_{rk} \geq 0$
  - For all $e \in L$: if $y_{re} \leq 0$, v_e cannot decrease and will therefore remain non-negative, else v_e can only remain non-negative if $v_e/y_{re} \geq v_k/y_{rk}$.
  - For all $e \in U$: if $y_{re} \geq 0$, v_e cannot increase and will therefore remain non-positive, else v_e can only remain non-positive if $v_e/y_{re} \geq v_k/y_{rk}$.

  Therefore when $d_r > 1$, $v_k/y_{rk} = min\{v_e/y_{re} \mid e \in S_2\}$.

For each non-basic variable x_e that can be considered for entering the basis, the ratio v_e/y_{re} has to be considered. Either the maximum or the minimum ratio will indicate the entering variable, depending on whether x_{Br} must increase or decrease. Geldenhuys [9, p.15] suggested a slightly more streamlined form which combines the two cases. This also makes it possible to simplify the two cases given above for constructing the set of non-basic variables that can be considered for entering the basis.

The set $S = S_1 \cup S_2$ is the set of all non-basic variables that can be considered for entering the basis. The process of choosing the entering edge k from the set S can be summarised as follows:

```
if ($d_r < 0$)
  then $\delta \leftarrow -1$
  else $\delta \leftarrow +1$
end if
$S \leftarrow \{e \in L \mid \delta y_{re} > 0\} \cup \{e \in U \mid \delta y_{re} < 0\}$
if ($S = \emptyset$)
  then //no column suitable to enter the basis
       //primal feasibility cannot be achieved
       stop
end if
Select $k \in S$ such that $v_k/\delta y_{rk} \leq v_e/\delta y_{re}\ \forall e \in S$
```

The value y_{re} can be calculated as outlined in Chapter 3, but that would mean a traversal of the basis graph for each ratio that is calculated. In the next section a more efficient method is given.

β values

In Chapter 3 an efficient method was discussed to calculate the entries of a column vector $\mathbf{y}$ that corresponds to the entering edge e. These entries were used in the ratio test for the primal simplex method. For the ratio test in the dual simplex method, we need entries of a row vector that contains the r-th entries of all vectors $\mathbf{y}_e = B^{-1}\mathbf{a}_e,\ \forall e \in L \cup U$. The denominator y_{re} used in the ratio test is therefore the e-th component of the row vector needed for the ratio test, with r corresponding to the leaving variable x_{Br}.

Therefore

$$y_{re} = \mathbf{e}_r B^{-1}\mathbf{a}_e$$

where $\mathbf{e}_r$ is the r-th unit row vector.

The repetitive traversal of the basis graph for the calculation of y_{re} for each non-basic edge e, can be eliminated by first calculating the row vector $\beta = \mathbf{e}_r B^{-1}$ (the r-th row of B^{-1}). Then, for each calculation of each y_{re}, we simply add the two entries in β corresponding to the two ones in $\mathbf{a}_e$. Since $\beta B = \mathbf{e}_r$ and each column of B contains exactly two ones, it is possible to solve for the entries in β. Each of the following equations corresponds to a column in B (and an edge in the basis graph) and the corresponding entry in $\mathbf{e}_r$.

If (a, b) is the edge corresponding to x_{Br},

$$\begin{aligned} \beta_a + \beta_b &= 1 \\ \beta_i + \beta_j &= 0 \textit{ for all other basic edges (i,j)} \end{aligned}$$

The structure of a 1-tree helps us again to solve these equations efficiently. These equations can be solved in the same way as the dual value equations in section 3.2.3 with a "cost" of 1 for basic edge (a, b) and a "cost" of 0 for all other basic edges. Entries in β can only be non-zero where entries correspond to nodes in the 1-tree that contains the edge (a, b). For calculation of the non-zero components, two different cases can be identified:

- If the edge (a, b) forms part of the cycle of a 1-tree, the β equations for all edges on the cycle can alternately be added and subtracted to solve for one β value (say β_a) : $2\beta_a = 1 - 0 + 0....$ From the equation for (a, b) it follows then that $\beta_a = \beta_b = \frac{1}{2}$. To solve the equations corresponding to each remaining edge $(i, j) \neq (a, b)$ on the cycle, β values corresponding to consecutive nodes on the cycle of the 1-tree have to alternate between $-\frac{1}{2}$ and $\frac{1}{2}$. All β values for the rest of the 1-tree can be calculated by substitution in a β equation containing one unknown quantity. β values at consecutive nodes connected by an edge e on a path away from the cycle alternate between $\frac{1}{2}$ and $-\frac{1}{2}$ to add up to 0 for each basic edge e except for (a, b).
- Otherwise, (a, b) is outside the cycle of a 1-tree with (say) node a closest to the cycle. The β values are non-zero only for the nodes in the subtree that contains b. $\beta_b = 1$ and β values corresponding to nodes that are connected by basic edges alternate between -1 and 1 to solve the equations defining β.

When all entries in β have been calculated, y_{re} can be calculated for any edge $e = (i, j) \in L \cup U$ with $y_{re} = \beta_i + \beta_j$.

4.4 Pivoting

Now that the leaving and entering variables are selected, we must proceed to change the basis. As side constraints were used in the dual simplex implementation of Leenen [21] and Geldenhuys [9], updating the basic variables and pivoting was much more involved than in the primal simplex method for the RMP2. As side constraints are not considered here, the basis uses exactly the same structure than that of the primal simplex method. This makes it possible to use the same algorithm for updating the basis structure and the dual values that were used for the primal simplex method.

Denote by Δ the amount by which the entering variable x_k is changed. In an LP without upper bounds on the variables, Δ would simply be d_r/y_{rk}. Here d_r reflects the change in value for the leaving variable.

As upper bounds are considered here in the RMP2, d_r does not necessarily reflect the change in value for the leaving variable: if x_{Br} was primal infeasible because $d_r > 1$, x_{Br} will become non-basic at its upper bound, 1. The change in value for x_{Br} when $d_r > 1$ is therefore $(d_r - 1)$. Δ is therefore defined as d_r/y_{rk} if $d_r < 0$, and as $(d_r - 1)/y_{rk}$ when $d_r > 1$ to correspond to the change in the value of x_{Br}.

Change in a non-basic variable affects the objective function value as follows:

$$z = z_0 + \sum_{e \in L} v_e x_e + \sum_{e \in U} v_e (x_e - 1)$$

We will be changing the entering variable x_k by Δ. Therefore, the change in z is $v_k \Delta$ regardless of whether k is in L or in U.

The technique used in the primal simplex method to update the dual values after the basis change, can be used again here, without any change.

The complete dual simplex algorithm is now given:

```
while ($(B, L, U)$ is primal infeasible) do
        Calculate the deviation $D_j$ for each primal infeasible variable
        Select r such that $D_r = max\{D_j \mid d_j < 0$ or $d_j > 1\}$
        if ($d_r < 0$)
           then $\delta \leftarrow -1$
            else $\delta \leftarrow +1$
        end if
        $S = \{e \in L \mid \delta y_{re} > 0\} \cup \{s \in U \mid \delta y_{re} < 0\}$
        if ($S = \emptyset$)
           then  stop
        end if
        Select $k \in S$ such that $v_k/\delta y_{rk} = min\{v_e/\delta y_{re} \mid \forall e \in S\}$
        if ($\delta < 0$)
           then $\Delta \leftarrow d_r/y_{rk}$
            else $\Delta \leftarrow (d_r - 1)/y_{rk}$
        end if
        $x_k \leftarrow x_k + \Delta$
        $x_{B_j} \leftarrow d_j - \Delta y_{jk}\ \forall j \in B$
        $z \leftarrow z + v_k\Delta$
        if ($\delta < 0$)
           then if ($k \in L$)
                    then $L \leftarrow L + \{B_r\} - \{k\}$
                     else $U \leftarrow U - \{k\}$
                          $L \leftarrow L + \{B_r\}$
                 end if
            else
                 if ($k \in L$)
                    then $U \leftarrow U + \{B_r\}$
                          $L \leftarrow L - \{k\}$
                     else $U \leftarrow U - \{k\} + \{B_r\}$
                 end if
        end if
        $B \leftarrow B + \{k\} - \{B_r\}$
        Update the dual values
end
```

4.5 Implementation

It was illustrated in Chapter 3 that variables assume one of the values in $\{0, \frac{1}{2}, 1\}$ during the primal simplex method. The addition of side constraints to the RMP2 as part of the dual simplex method in the work of Leenen [21] and Geldenhuys [9] caused variables to assume any real value. This necessitated the use of floating point variables to store the values of the variables. The variables were converted from integers to floating point values after completion of the primal simplex method.

As there are no side constraints in the implementation discussed in Chapter 5, the basis structure (B, L, U) has similar attributes throughout the dual simplex method compared with what it had during the primal simplex method. Each element in the inverse of the basis remains an integer divided by 2. As primal feasibility is not maintained here, the range of a basic variable is however now not limited to $\{0, \frac{1}{2}, 1\}$ as during the primal simplex method, but the value of each remains an integer divided by 2. During implementation, this can be exploited throughout the dual simplex method. Since $2x_{Bj}$ is an integer, basic variables can be stored as integer values by simply storing $2x_{Bj}$ instead of x_{Bj}. Non-basic variables, as with the primal simplex method, assume only the values of either their upper bound (0) or lower bound (1).

These properties of the values of the variables have the further advantage that the same representation can be used for both the primal and dual simplex methods, without the need to convert after completion of the primal simplex method. The use of floating point variables is therefore eliminated entirely. Computation is faster than with floating point numbers. The numerical instability that Geldenhuys and Leenen experienced is also eliminated because of the perfectly accurate presentation.

4.6 Conclusion

The dual simplex method for the RMP2 was presented here. Primal feasibility that is lost because of changing the right-hand side vector ($\mathbf{b}$) is restored.

The techniques presented here are simpler than those of Geldenhuys [9] and make it possible to implement these techniques entirely with the use of integer variables. As similar attributes are present during the dual simplex method compared with the primal simplex method, certain operations discussed as part of the primal simplex method can be used without change in the dual simplex method.

In the following chapter the CSM for the TSP is presented, which manipulates the right-hand side vector ($\mathbf{b}$). When primal infeasibility arises, the dual simplex method presented in this chapter is used to restore primal feasibility.

Chapter 5

The Column Subtraction Method for the Traveling Salesman Problem

In Chapter 2 the Column Subtraction Method (CSM) for solving Set Partitioning Problems (SPP) was introduced. It was shown to be an effective method for solving SPPs and also achieved excellent speedup in a parallel processing environment. This chapter presents the CSM as a method for solving the Traveling Salesman Problem (TSP). The algorithms discussed in Chapters 3 and 4 are used to implement efficient primal and dual simplex methods.

The CSM has to be adapted for the TSP since the relaxation used for the TSP (the RMP2) differs from the LP relaxation used for the SPP in three ways:

- The right hand side of the relaxation is a vector of twos (not ones).
- Upper bounds are used and therefore non-basic variables can assume the value of their upper bound (1).
- Even when all the variables are integer, they do not necessarily define a solution for the TSP, as they might define subtours.

The basic process for solving the TSP with the CSM is similar to that which was discussed in Chapter 2. A relaxation of the TSP (the RMP2) is first solved to find an optimal basis structure (B, L, U) (see Chapter 3). If the optimal basic solution does not correspond to a tour at least one of the edges in L must be in the optimal tour or at least one of the edges in U must not be part of the optimal tour. A search tree is constructed in which at each node certain non-basic variables are either fixed at the value they had in the optimal solution to the RMP2, or fixed at an altered value. Fixing a variable at an altered value entails either that an edge in L is fixed into the optimal tour for the subproblem (the variable is fixed at altered value 1), or an edge in U is fixed out of the optimal tour for the subproblem (the variable is fixed at altered value 0).

Fixing a variable at an altered value can result in primal infeasibility, and in such cases the dual simplex method can be applied to restore primal feasibility. The resulting basic solution can then be tested to see if it defines a solution to the TSP.

The improved CSM by Smith and Thompson [30] specifies nine rules under which a node can be fathomed (see Chapter 2). Some of these rules are tied to the attributes of the SPP and cannot be used directly in the TSP. This chapter describes which of the nine rules can be used in the TSP and how some of them were adapted for the attributes that are present in the TSP. This chapter also discusses some of the techniques that were used to implement the CSM for the TSP. In closing, some ideas for future improvements are mentioned.

Definitions

A few definitions follow that describe the extent to which the constraints of the RMP2 are satisfied by a set of variables:

- When a set of variables provides an integer solution to the RMP2, the columns corresponding to the positive variables add up to a vector of twos. This set of columns is said to *cover* the rows or to *provide a cover.*

- When a set of columns add up to a vector that only contains values from $0, 1, 2$, the set of columns is said to *partially cover* the rows or to *provide a partial cover.*

- When a set of columns add up to a vector containing values of which one or more exceed 2, the set of columns is said to *overcover* the rows or to *provide an overcover.*

5.1 Constructing the CSM search tree

At each node of the search tree one or more variables that are non-basic in the optimal solution to the relaxation are either fixed at their upper bound (1) or their lower bound (0).

Just as in the case of the SPP, an upper and a lower bound on the optimal objective function value are kept, denoted by zIP and zLP respectively. As indicated below, the lists can be trimmed to exclude all variables with reduced cost greater than or equal to $zIP - zLP$, because they cannot become part of a tour with a cost less than zIP.

The change in cost from zLP, the cost of the optimal solution to the RMP2, when changing the value of a non-basic variable, is seen in the following:

$$z = zLP + \sum_{e \in L} v_e x_e + \sum_{e \in U} v_e (x_e - 1)$$

If $x_f, f \in L$ increases to 1, we have

$$z = zLP + v_f.$$

Therefore, for variables in L, only those with reduced cost less than $zIP - zLP$ can possibly equal 1 in a tour with cost less than zIP.

Similarly, if $x_f, f \in U$ decreases to 0, we have

$$z = zLP - v_f.$$

Therefore, for variables in U, only those with reduced cost greater than $-(zIP - zLP)$ can possibly equal 0 in a tour with cost less than zIP.

The edges corresponding to the set of variables that are fixed at 1 are called the *partial tour* defined by the subproblem, because they define part of the optimal tour that is sought at a subproblem.

Whenever a tour is found with a lower objective function value than zIP, zIP is updated. The amount of non-basic variables that now need to be considered decreases too, because the gap between zIP and zLP decreases.

Different orderings for the non-basic variables are possible. Two of these are discussed next.

5.1.1 L followed by U as separate lists

Two sorted lists are constructed that correspond to the index sets L and U. L is sorted in non-decreasing order of reduced cost, and U is sorted in non-increasing order of reduced costs. L and U will interchangeably be used to refer to a set of variables, edges or columns that correspond to the index sets L and U. When constructing the search tree, the non-basic variables can be considered as part of one list which is formed by appending the list U to the list L. If $i > |L|$, the i-th variable is $U_{i-|L|}$. The use of non-basic variables in creating the search tree is different compared with the SPP, to allow the correct consideration of non-basic variables in U that are at their upper bound in the optimal solution to the relaxation.

To create the first child of the root node the variable L_1 is fixed at the altered value 1. For $1 < i \leq |L|$, the i-th child of the root node is created by fixing the variable L_i at the altered value 1 and fixing the variables L_1 to L_{i-1} at 0. For $i > |L|$, the i-th child of the root node is created by fixing variable $U_{i-|L|}$ at the altered value 0, fixing all variables in L at 0 and if $i > |L| + 1$, fixing variables U_1 to $U_{i-1-|L|}$ to 1.

The first child of a node for which the i-th variable was fixed at an altered value is created by additionally fixing the $(i+1)$-th variable at an altered value. For $1 < j$, the j-th child of a node for which the i-th variable was fixed at an altered value is created by additionally fixing the $(i+j)$-th variable at an altered value and fixing the $(i+1)$-th to $(i+j-1)$-th variables at their original values (0 for variables in L, and 1 for variables in U).

When a new tour is found and zIP is updated, the lists L and U can be trimmed to reflect the new gap.

5.1.2 All non-basic variables in one list

Another possibility is to compile a single list E with all non-basic variables. Such a list is then sorted according to non-increasing order of the absolute values of the reduced costs of the variables (to account for the fact that variables from L and U have different signs). The variables in U will therefore be interleaved between the variables in L.

To create the first child of the root node the variable E_1 is fixed at an altered value (1 for variables in L, 0 for variables in U). For $1 < i$, the i-th child of the root node is created by fixing the variable E_i at the altered value and fixing the variables E_1 to E_{i-1} at their original values (0 for variables in L, 1 for variables in U).

The first child of a node for which the i-th variable was fixed at an altered value is created by additionally fixing the $(i+1)$-th variable at an altered value. For $1 < j$, the j-th child of a node for which the i-th variable was fixed at an altered value is created by additionally fixing the $(i+j)$-th variable at an altered value and fixing the $(i+1)$-th to $(i+j-1)$-th variables at their original values (0 for variables in L, and 1 for variables in U).

When a new tour is found and zIP is updated, the list E can be trimmed to reflect the new gap. This technique was not tested computationaly.

5.2 Fixing a non-basic variable at an altered value

At each node in the search tree of the SPP, a set of non-basic columns was fixed into the optimal partition for a subproblem by fixing the corresponding variables at 1, and then re-optimising. This was done by subtracting all the columns of A that were fixed into the optimal partition of the subproblem, from the right hand side of the LP formulation. The process of fixing columns into the optimal partition and fixing the corresponding variables equal to 1 had to be adapted for use with the TSP as non-basic variables could possibly have a value of 1 (the variables in U). The use of upper bounds necessitates consideration of *adding* a column of A to the right hand side and fixing the variable equal to 0.

5.2.1 Fixing a variable in L at altered value 1

To fix a variable x_f, $f \in L$, at a value of 1, the corresponding column $\mathbf{a}_f$ of the matrix A is subtracted from the right hand vector of the equations ($\mathbf{b}$ in our formulation). The manipulations discussed next will show why column subtraction can possibly cause primal infeasibility.

As the vector $\mathbf{b}$ is not used in the implementation, but only $\mathbf{d}$, $\mathbf{d}$ has to be updated in such a manner as to reflect the change in $\mathbf{b}$. Suppose a column $\mathbf{a}_f$ corresponding to variable x_f, $f \in L$, and edge (u, v) is subtracted from $\mathbf{b}$. Subtracting $\mathbf{a}_f$ from $\mathbf{b}$ corresponds to fixing x_f at 1. Two entries in $\mathbf{b}$, b_u and b_v, are decreased by 1. This means that rows u and v need to be covered one time less because $\mathbf{a}_f$ will complete the cover. Since

$$\mathbf{d} = B^{-1}\mathbf{b} - \sum_{e \in U} \mathbf{y}_e$$

the values in $\mathbf{d}$ after the column subtraction, $\mathbf{d}'$, are as follows:

$$\mathbf{d}' = B^{-1}(\mathbf{b} - \mathbf{a}_f) - \sum_{e \in U} \mathbf{y}_e$$

$$\begin{aligned} &= B^{-1}\mathbf{b} - B^{-1}\mathbf{a}_f - \sum_{e \in U} \mathbf{y}_e \\ &= B^{-1}\mathbf{b} - \sum_{e \in U} \mathbf{y}_e - B^{-1}\mathbf{a}_f \\ &= \mathbf{d} - \mathbf{y}_f \end{aligned}$$

Therefore, only $\mathbf{y}_f = B^{-1}\mathbf{a}_f$ need to be subtracted from the current value of $\mathbf{d}$. The calculation of the vector $\mathbf{y}_f$ was discussed in Chapter 3 (p. 34).

During the Column Subtraction Method, more than one column can be subtracted from the right-hand side. Under such circumstances $\mathbf{d}$ is updated at each node in the search tree for each of the edges that is fixed into the optimal tour of the subproblem by subtracting $\mathbf{y}_f$ for each variable x_f, $f \in L$ fixed at 1.

5.2.2 Fixing a variable in U at altered value 0

To fix a variable x_f, $f \in U$, at a value of 0, the corresponding column $\mathbf{a}_f$ of the matrix A is added to the right hand vector of the equations ($\mathbf{b}$ in our formulation). This can also result in primal infeasibility.

The two entries in $\mathbf{b}$ that correspond to the nodes connected by edge $f = (u, v)$, b_u and b_v, are increased by 1. This means that rows u and v need to be covered one time more as $\mathbf{a}_f$ no longer contributes to the cover. The values in $\mathbf{d}$ after the column addition, $\mathbf{d}'$, are as follows:

$$\begin{aligned}\mathbf{d}' &= B^{-1}(\mathbf{b} + \mathbf{a}_f) - \sum_{e \in U} \mathbf{y}_e \\ &= B^{-1}\mathbf{b} + B^{-1}\mathbf{a}_f - \sum_{e \in U} \mathbf{y}_e \\ &= B^{-1}\mathbf{b} - \sum_{e \in U} \mathbf{y}_e + B^{-1}\mathbf{a}_f \\ &= \mathbf{d} + \mathbf{y}_f\end{aligned}$$

If S is a set that contains all variables with altered values,

$$\mathbf{d}' = \mathbf{d} - \sum_{e \in L \cap S} \mathbf{y}_e + \sum_{e \in U \cap S} \mathbf{y}_e$$

Updating $\mathbf{d}$ in this manner can result in values in $\mathbf{d}'$ becoming negative or greater than one. Therefore, primal feasibility can possibly be lost at this point.

Once primal feasibility has been lost, the dual simplex method discussed in Chapter 4 can be applied in an attempt to regain primal feasibility.

5.3 Fathoming rules

By far the greatest part of the time spent in solving the TSP with the CSM, is spent on traversing the search tree and applying the dual simplex method. It is therefore desirable to prune the search tree as much as possible, to reduce unnecessary calculations.

Almost all the rules for fathoming a node that were used for the SPP can be used for the TSP with no or minor changes. This section describes how the rules are applied to the TSP. The rules are presented in the order in which they are applied. The ordering attempts to do inexpensive tests before computationally expensive checks are attempted. The TSP exhibited behaviour different from that of the SPP with regard to the effectiveness of certain fathoming rules to fathom nodes as well as the computational efficiency of the rules. These rules will be discussed in the following sections.

5.3.1 The inexpensive fathoming rules

If certain conditions hold, it may be possible to detect that a node can be fathomed before a column is subtracted (or added) and therefore also before the dual simplex method may need to be applied. The checks described here are easy to carry out and can avoid expensive checks in some cases.

With the SPP a node could be fathomed if a column were fixed into the partial partition and the column were not orthogonal to the other columns in the partial partition. This was easy to implement as it was only necessary to check that the new column in the partial partition only covered rows that were not covered by other columns in the partial partition. Since the right hand side for the TSP is **b**, a vector of twos, this fathoming rule has to be adapted.

A different formulation of this rule for the TSP (which also applies to the SPP) is that a node can be fathomed if the column fixed to create the node causes an overcover (**Rule 1**). Therefore, if a row is only covered by a single fixed column in the TSP, the node of the search tree cannot yet be fathomed, as another edge corresponding to a column with a one in that row can also be fixed into the partial tour for that node.

If the cost of the partial tour equals or exceeds the current best known solution, the node can be fathomed (**Rule 4**). This rule is directly applicable to the TSP as in the case of the SPP.

With the SPP **Rule 5** considered the case where a partial partition is a complete cover and therefore provides a solution by itself. The node can be fathomed as no child can give a better solution. With the TSP, however, such partial covering defines a 2-matching, but not necessarily a tour. If it defines a tour, zIP can be updated and the node can be fathomed. If the partial covering does not define a tour, the node can be fathomed nonetheless because no more variables can be fixed at value 1, as that would cause an overcover. No more variables in U are left to fix at 0, otherwise the fixed columns would not be able to define a cover.

With the TSP this rule is unlikely to ever find a tour, but checking for it is not expensive, and it is useful as no child node can provide a solution to improve on the current best known solution.

5.3.2 Row-wise fathoming rules

The rules discussed in this section can also be applied before column subtraction or column addition takes place and can therefore possibly fathom a node before necessitating the dual simplex method. They are more expensive than the ones mentioned before as it is necessary to iterate over a large amount of variables to obtain the necessary information. Only edges remaining in L and U that are incident to a node for which less than two incident edges have been fixed into the partial tour, have to be considered.

With the SPP, where there was only one column left that could cover an uncovered row, it had to form part of the solution. Where there was no column left to cover an uncovered row, the node could be fathomed (**Rule 6**). These considerations are slightly more involved in the case of the TSP, because the right hand side contains twos, not ones, and therefore the technique has to be adapted.

When there is any row that cannot be covered entirely (twice) by the available columns, a solution is impossible. This condition includes both the cases where there is no column or only one column that can cover a row. This is also applicable where there is a column corresponding to a variable fixed at 1, already partially covering a row, and where there are no available columns to complete the cover for that row. In such a case only one column need to be found to complete the covering.

It is therefore not possible to apply the concept of a one-row directly to the TSP. However, there exist cases where a cover can only be achieved in one way. This could happen if only one column is needed to complete the cover in a row that is partially covered by a fixed column, or where two columns are still needed and only two candidates exist. This check is not present in the implementation presented as part of this study.

5.3.3 After subtracting (or adding) a column

If none of the previous fathoming rules could help to fathom the node, a column must be subtracted from or added to the right hand side. Similarly to the SPP, we formulate an associated LP (ALP) as follows:

$$\begin{array}{rl} \textit{Minimise} & \mathbf{cx} \\ \textit{Subject to} & A\mathbf{x} = \mathbf{b}' \\ & \mathbf{0} \leq \mathbf{x} \leq \mathbf{1} \end{array}$$

where $\mathbf{b}'$ is obtained from $\mathbf{b}$ by subtracting all the columns associated with edges in $L \cap S$ and adding all the columns associated with edges in $U \cup S$. As mentioned earlier, an upper and a lower bound on the optimal objective function value is kept, denoted by zIP and zLP respectively. The basis B in the optimal solution to the RMP2 defines a dual feasible basic solution for the ALP. The cost of this solution provides a lower bound on the cost of completing the node's partial tour. **Rule 2** specifies (as in the case of the SPP) that a node can be fathomed if the cost of the basic solution equals or exceeds zIP.

It was mentioned in the introduction that in the case of the TSP, if all variables are integer, they do not necessarily define a solution, as the subgraph with edge set corresponding to the positive variables may contain subtours. In the case of the SPP an integer solution meant that no children could produce solutions that are better than the current best known solution and a node could therefore be fathomed.

For the TSP this fathoming rule can be applied after column subtraction (or addition) before the dual simplex method is invoked, if the current solution is not only integer, but also forms a tour (**Rule 3**). Therefore, if the solution to the ALP at a node has all variables as integers, but subtours are present, the node cannot be fathomed as children nodes might still provide a solution with an objective function value better than the current best known value (zIP).

5.3.4 When the dual simplex cannot be avoided

If a node cannot be fathomed by one of the previous fathoming rules, the dual simplex algorithm has to be applied.

If primal feasibility cannot be obtained with the dual simplex method, the node can be fathomed (**Rule 7**). This applies exactly as in the case of the SPP.

During the dual simplex method, if the resulting objective function value for a subproblem is greater than or equal to zIP, the node can be fathomed (**Rule 8**). This applies exactly as in the case of the SPP.

In the case of the SPP, if primal feasibility is obtained and the resulting solution is integer, it can be considered as a solution. For the TSP it additionally has to form a tour to be considered a solution. If the solution is in fact a tour, the node can be fathomed as no child node can provide a better solution (**Rule 9**). In such a case zIP is updated. If the objective function value does not exceed zIP, and the solution is not integer or not a tour, the node cannot be fathomed.

5.4 Refinements and future research

The CSM for the TSP, as presented here, is based on the CSM for the SPP (see Chapter 2). Mostly the same structure and fathoming rules could be used.

Further improvements to what was presented here are possible. Extra fathoming rules might be possible. For example: if the partial tour contains a subtour, a node can be fathomed as all children will also contain that subtour. To implement this rule, a method for detecting subtours in a subset of less than n edges, will have to be used.

In some cases (notably the bigger problems), tours found in the CSM search tree exhibited relative locality in the search tree, i.e. two or more solutions were found in reasonably short succession. It might be worthwhile to investigate the properties of the branches in which these tours are found, to perhaps first search in such branches.

Heuristics can also be used to speed up the CSM:

- Instead of blindly using the objective function value of the initial basis for the RMP2 as an initial value for zIP, zIP can possibly be lowered. Objective function values of the optimal solutions to the investigated problems were found to be within 10% of the optimal value of the RMP2 solution, zLP, with the exception of problem gr17 from the TSPLIB (discussed in Chapter 7). This will allow for consideration of fewer non-basic variables for constructing the CSM search tree, and will therefore limit the width of the search tree.
- The use of a better heuristic for the initial starting basis for the RMP2 could also provide a better initial value for zIP.
- Optimal solutions were mostly found at relatively shallow depth in the CSM search tree. A possible heuristic could therefore be to limit the depth of the search tree. A possible heuristic is to limit it to 25% of n. This heuristic will of course make fathoming rule 5 redundant.

5.5 Conclusion

The CSM was presented here as a method to solve the TSP. The primal simplex method discussed in Chapter 3 is used to solve a relaxation of the

TSP (namely the RMP2). A depth-first search tree is constructed where at each node a subproblem is created with certain variables fixed at 0 or 1. Fixing a variable at an altered value is achieved by subtracting (or adding) a column of A from (or to) the right hand side vector $\mathbf{b}$. The dual simplex method discussed in Chapter 4 is used if the subtraction (or addition) results in primal infeasibility.

In Chapter 6 certain techniques are presented that are used to implement some of the fathoming rules described in this chapter. The algorithms in Chapter 6 focus on scalability. Implementation details and running times are reported in Chapter 7. The CSM for the TSP lends itself to the same method of parallelisation than that described with regard to the SPP (see Chapter 2). The parallelisation and the running times on more than one processor are also presented in Chapter 7.

Chapter 6

Techniques used in the Column Subtraction Method for the Traveling Salesman Problem

In Chapter 5 the CSM for the TSP was presented. In this chapter some of the techniques that were used in the method are presented. Two calculations that are very often needed during the CSM for the TSP is to calculate which edge is incident on two given nodes, or to calculate which two nodes are connected by a given edge. For the discussion of these calculations the node and edge numbering scheme is presented, followed by the techniques used to perform these calculations.

In the section on fathoming rules (Chapter 5), the necessity of testing whether a basic solution defines a tour was frequently mentioned. (See rules 3, 5, and 9.) The method employed to perform this test is presented here.

As the methods and calculations presented here are performed often, the potential impact on the running time of an implementation can be huge. They should be efficient, and should ideally scale favourably to be useful in large problem instances.

6.1 Calculating node numbers and edge numbers

Throughout the implementation of the CSM for the TSP, it is very often necessary to calculate which edge connects two given nodes, or to calculate which two nodes are connected by a given edge. Profiling of the code indicated that functions to do these calculations are called more than any of the other functions in the program, and that a substantial amount of time is spent on these calculations (5% - 10%). It is therefore important that these functions are implemented efficiently. One possibility is to store all the information in memory, but this would consume a large amount of memory.

6.1.1 Node and edge numbering

Before presenting the methods used to do the above mentioned calculations, the numbering scheme that is employed will be presented. Both edges and nodes can either be numbered from 0 or numbered from 1. The choice influences the exact details of the formulas presented here. Only the numbering scheme from zero is presented. It is assumed that both nodes and edges are numbered from zero.

Say we have n nodes numbered from 0 to $(n-1)$ and m edges numbered from 0 to $(m-1)$. As G (see p. 22) is a fully connected, undirected graph, $m = n(n-1)/2$. The edge (0,1) connecting node 0 with node 1, is numbered 0. The edges incident on node 0 are numbered 0 through $n-2$. Edge (1,2) is numbered $n-1$. The edges $(1,i)$ with $i > 1$ are numbered $n-1$ through $(n-1)+(n-3)$.

Below is a table listing the edge numbers for a graph with $n = 6$. Numbers are only given for the upper triangle to indicate the numbering scheme.

The ordering in this numbering scheme corresponds to the left-to-right, top-to-bottom reading of the upper triangle of the square cost matrix. This is the format used for some of the problems in the TSPLIB [24] that is

Node numbers	**0**	**1**	**2**	**3**	**4**	**5**
0		0	1	2	3	4
1			5	6	7	8
2				9	10	11
3					12	13
4						14

Table 6.1: Edge numbers with $n = 6$ when numbering from zero

marked as being in `UPPER_ROW` format. Other formats used in the TSPLIB can of course be converted to this numbering scheme easily. The TSPLIB is discussed in Chapter 7.

6.1.2 Calculating an edge number from node numbers

Edge numbers can be calculated as follows:

In general, the edge $(x, x+1)$ is numbered

$$\begin{aligned} k &= (n-1) + (n-2) + (n-3) + ... + (n-x) \\ &= \sum_{i=1}^{x} (n-i) \end{aligned}$$

An edge (x, y) connecting two arbitrary nodes is numbered relative to the edge $(x, x+1)$ by adding $(y - x - 1)$. The edge number k for an arbitrary edge (x, y) is therefore

$$k = \sum_{i=1}^{x} (n-i) + y - x - 1 \qquad (6.1)$$

The summations in the above formulas can be done in constant time with a formula, and therefore the calculation of edge numbers can be done in $O(1)$ time. Geldenhuys [9] used a lookup table for the values of the summations minus 1 ($\sum_{i=1}^{x}(n-i)-1$) for $x \in \{0, 1, 2, ..., (n-1)\}$. Using a lookup table, the number of edge (x, y) can be found by using x as index in the lookup table, and merely adding the difference between y and x to the retrieved value. The lookup table can easily be constructed to work with both numbering schemes.

```
$base[0] \leftarrow -1$
$j \leftarrow 0$
$i \leftarrow n - 1$
**while** $(j < n - 1)$ **do**
        $base[j + 1] \leftarrow base[j] + i$
        $j \leftarrow j + 1$
        $i \leftarrow i - 1$
**end**
```

It is however important that $x < y$. Using the lookup table `base`, this calculation can be described as follows (this calculation is the same for both numbering schemes):

```
*integer* **function** *edgenumber*$(x, y)$
            **if** $(x < y)$
               **then** *return* $(base[x] + y - x)$
                **else** *return* $(base[y] + x - y)$
            **end if**
**end**
```

6.1.3 Calculating node numbers from an edge number - $O(log(n))$

During the dual simplex method (see Chapter 4) and the CSM (see Chapter 5) it is often necessary to know which nodes are connected by a given edge. Geldenhuys [9] used a binary search on the lookup table used in the previous section to find one of the node numbers (the one with the smallest number) and could then calculate the second node number using the edge number, the lookup table and the number of the first node. Given an edge k, the following algorithm can determine the nodes x and y that is connected by k. (The division in the following algorithms is, of course, integer division.)

```
function nodenumbers(k, x, y)
  $l \leftarrow 0$
  $r \leftarrow n - 1$
  while $(l < r)$ do
        $m \leftarrow (l + r)/2$
        if $(k > base[m])$
          then $l \leftarrow m + 1$
           else $r \leftarrow m$
        end if
  end
  $x \leftarrow l - 1$
  $y \leftarrow k - base[x] + x$
end
```

The search on the lookup table is $O(log(n))$. Next a new method is presented which can do the calculation in $O(1)$ time.

6.1.4 Calculating node numbers from an edge number - $O(1)$

A technique is presented here which is able to eliminate the search on the lookup table to calculate the node number x of an arbitrary edge (x, y) with $x < y$. It is assumed again that nodes and edges are numbered form zero. While it is possible to adapt the technique to work with numbering from one, the formulas presented in this section are affected by the decision, and the adaption is not straight forward.

First we develop the technique for an edge $(x, x + 1)$.

$$k = \sum_{i=1}^{x}(n - i)$$

$$\begin{aligned} k &= \sum_{i=1}^{x} n - \sum_{i=1}^{x} i \\ k &= xn - (x+1)x/2 \\ 2k &= 2xn - (x+1)x \end{aligned}$$

Now we have a quadratic equation in x:

$$0 = x^2 + (1-2n)x + 2k$$

The roots are

$$\begin{aligned} x &= \frac{2n - 1 \pm \sqrt{(2n-1)^2 - 8k}}{2} \\ x &= n - \frac{1 \pm \sqrt{(2n-1)^2 - 8k}}{2} \qquad (6.2) \end{aligned}$$

These roots will always be real: since k is an edge number, the maximum value it can attain is therefore less than $n(n-1)/2$ (when numbering from zero). The discriminant in the equation is thus:

$$\begin{aligned} &= (2n-1)^2 - 8k \\ &> (2n-1)^2 - 8n(n-1)/2 \\ &= 4n^2 - 4n + 1 - 4n^2 + 4n \\ &= 1 \end{aligned}$$

For the positive sign in the fraction in equation 6.2, the fraction has a value that is greater than 1. For the negative sign the fraction has a value that is negative. The greater of the two roots therefore yields a value for x that is greater than n - which is not a valid node number. Therefore the smaller root is the correct one:

$$x = n - \frac{1 + \sqrt{(2n-1)^2 - 8k}}{2} \qquad (6.3)$$

The difference between the edge numbers for an arbitrary edge (x, y) with

$x < y$, and that of the edge $(x, x+1)$, corresponds to the difference between their respective second components of the node pairs. The edge numbers will therefore differ by $y - (x+1)$. If the number k of edge $(x, x+1)$ were to increase by $n - x - 1$ or more, it would not be incident on x anymore. This fact enables us to analyse the effect on equation 6.3 of passing a value of k which corresponds to an edge (x, y) where $y \neq x + 1$:

An increase in k of exactly $n - x - 1$ will yield $x + 1$ in equation 6.3. An increase in k of less than $n - x - 1$ will cause (fractional) increase in x, but x will not reach or exceed $x + 1$. The floor of equation 6.3 will therefore calculate a correct value of x for an arbitrary value of k, $0 <= k < m$. This corresponds to taking the ceiling of the fractional part (that is subtracted).

From equation 6.1 we can then also calculate y:

$$\begin{aligned}
k &= \sum_{i=1}^{x}(n-i) + y - x - 1 \\
y &= x + k + 1 - (\sum_{i=1}^{x} n - \sum_{i=1}^{x} i) \\
y &= x + k + 1 - (xn - x(x+1)/2) \\
y &= x + k + 1 - x(n - (x+1)/2) \\
y &= x + k + 1 - x(2n - x - 1)/2
\end{aligned}$$

To summarise:

$$\begin{aligned}
k &= \sum_{i=1}^{x}(n-i) + y - x - 1 \\
x &= n - \left\lceil \frac{1 + \sqrt{(2n-1)^2 - 8k}}{2} \right\rceil \\
y &= x + k + 1 - x(2n - x - 1)/2
\end{aligned}$$

The time complexity of this new technique is $O(1)$ and should scale much better than the previous method. It does unfortunately involve floating point arithmetic for the calculation of the square root and of the ceiling. Quite a few integer calculations are necessary as well.

6.2 Testing if a solution is a tour

An often needed functionality in the CSM for the TSP is to test whether a basic solution defines a tour. It is therefore important that the test be performed efficiently. This section describes the data structures and algorithms used to perform this test.

When the basic solution at a node is both primal and dual feasible, it is known to be an optimal solution for the ALP. Additionally a TSP tour has the following attributes:

- All variables must be integer, i.e. 0 or 1
- The set of edges corresponding to the variables with value 1 must define a tour (there may be no subtours)

When testing whether a solution to the ALP defines a tour, its important to remember that edges corresponding to the following variables can form part of the tour:

- basic variables with value 1,
- non-basic variables originally in L that is fixed at value 1 by means of column subtraction,
- non-basic variables originally in L that entered U when applying the dual simplex method,
- non-basic variables originally in U and still present in U that have not been fixed at value 0 by means of column addition.

In the implementation presented here, the Union-Find algorithm (see [27], [5] and [7]) is used to determine if the current solution is a tour.

6.2.1 The Union-Find algorithm

In the way we use the Union-Find algorithm, it receives a set of edges as input. It determines whether the subgraph with this edge set is connected or not. If the subgraph is connected, it defines a tour.

A new graph G' with n nodes is considered which initially has no edges. Edges added to G' will always result in G' defining a set of trees. For each node in the graph, a parent index is kept in an array. These indices are used to traverse upward in a tree of nodes defined by these parent relationships. A parent relationship indicates that the corresponding two nodes in the TSP are connected. A subgraph in G' defined by any subset of these nodes are acyclic and therefore the whole graph defined by the nodes and parent relationships define a forest of trees. Initially all nodes are initialised to have no parent by setting all parent indeces to an invalid index, -1. (The choice of this number will be explained later.) Therefore, each node defines a tree with only one node.

Two important functions are necessary to implement the Union-Find algorithm:

- The `findroot(x)` function takes one node, x, as parameter. From node x it traverses the graph by following parent indices until it reaches the root node, i.e. a node without a parent. It returns the index of this root node. (If the parameter to the function does not have a parent, the node itself is returned.)

- The `union(a, b)` function receives two nodes a and b connected in G by an edge as parameters. For each of these nodes the `findroot()` function is called to obtain the roots of the trees to which they belong. One of these root nodes is assigned as the parent of the other root node and thus combines the two trees to become one tree.

The Union-Find algorithm can determine whether a solution is a tour by executing the union() operation for all node pairs connected by edges corresponding to variables with value 1 and then determining whether the edges form a connected subgraph. The code for these functions follows:

```
function $findroot(x)$
  while (x has a parent) do
        $x \leftarrow parent[x]$
  end
  return x
end
```

```
function $union(a, b)$
  $a \leftarrow findroot(a)$
  $b \leftarrow findroot(b)$
  if $(a = b)$
    then stop
  end if
  $parent[a] \leftarrow b$
end
```

After all union operations are carried out, the structure of the tree formed by the parent relationships among the nodes depend on the order in which the edges were supplied to the union operation. From the algorithm above, it is clear that even the ordering of the nodes a and b makes a difference to the direction of the parent-child relationship between a and b. This leaves the door open for a tall narrow tree to form, which will make the findroot operation more expensive than with a flatter tree.

For scalability this can be fixed by means of two improvements on the union-find algorithm.

Path compression

The parent relationships are only needed to be able to do the findroot operation. The exact relationships are therefore arbitrary, as long as the root of the tree that is formed remains the same. We can therefore arbitrarily change a node's parent index to that of another node higher up in the tree structure, or specifically, directly to that of the root. After such a change the time complexity for a findroot operation for that node is *O(1)* where before it was *O(h)* where h is the depth of that node.

Because the path that is followed to the root node is shortened this is called *path compression*. A suitable time to perform the path compression has to be sought. After traversing from a node to the root of the containing tree in a findroot operation, we can traverse over the whole path again and set each node's parent index to the root node that was found in the previous step.

An algorithm for `findroot()` with path compression is as follows:

```
function $findroot(x)$
  $xroot \leftarrow x$
  while (xroot has a parent) do
        $xroot \leftarrow parent[xroot]$
  end
  //Now we can do the path compression:
  while (x has a parent) do
        $temp\_parent \leftarrow parent[x]$
        $parent[x] \leftarrow xroot$
        $x \leftarrow temp\_parent$
  end
  return xroot
end
```

Path compression will ensure that paths that were traversed before will be compressed, but paths formed by union operations, could still cause long uncompressed paths. A possible solution is height balancing.

It was mentioned before that the ordering of the nodes a and b as parameters to the union operation determines the direction of the parent relationship. This ordering affects the length of the resulting paths, because no path compression is performed as part of the union operation. It is therefore advantageous to make a guided choice rather than an arbitrary one. One possibility is height balancing, where the choice minimises the height of the tree resulting from the union operation. The use of path compression in combination with height balancing diminishes some of the advantages provided by height balancing because the height of a tree might change after the `findroot()` operation.

Weight balancing

Another possibility is weight balancing, which is slightly more suited to the implementation discussed here. Weight balancing uses the number of nodes in each tree instead of the height and is therefore unaffected by path compression. Weight balancing makes the root of the tree with fewer nodes a child of the root of the tree with more nodes. For example, a union operation for the nodes a and b belonging to trees of size 10 and 15 respectively, will make a a child of b. It can be proven (see [5]) that a tree with n nodes constructed with weight balancing will have a depth of at most $\lfloor log(n) \rfloor$.

An algorithm for `weighted_union()` is as follows:

```
proc $weighted\_union(a, b)$
  $a \leftarrow findroot(a)$
  $b \leftarrow findroot(b)$
  if $(a = b)$
    then stop
  end if
  if (a's tree has less nodes than b's tree)
    then
          $parent[b] \leftarrow parent[b] + parent[a]$
          $parent[a] \leftarrow b$
     else
          $parent[a[\leftarrow parent[a] + parent[b]$
          $parent[b] \leftarrow a$
  end if
end
```

Apart from choosing the direction of the parent relationship more carefully, a further advantage of weight balancing lies in the implementation details - it will be easy for us to determine if all nodes are in the same tree. Instead of storing the weight of each tree separately, it can be stored as a negative number at the root of each of the trees in the place where the parent index is stored (there is of course no parent for a root node). This is done by letting non-negative numbers denote real parents and negative numbers indicate that the relevant node is the root of a tree, with the negative number indicating the negative of how many nodes are in the tree. When the root of a tree contains $-n$ it is an indication that there are n nodes in the tree and therefore that the edges that were provided to the Union-Find algorithm is a connected subgraph and therefore defines a tour. Therefore it is easy to determine whether a union operation caused G' to become connected.

For this reason `weighted_union()` is also used to determine whether a union operation causes G' to become connected. A more complete algorithm for `weighted_union()` that includes this check is now provided.

```
boolean function weighted_union(a, b)
        $a \leftarrow findroot(a)$
        $b \leftarrow findroot(b)$
        if ($a = b$)
          then return (parent[a] = -n)
        end if
        if ($parent[a] > parent[b]$)
          then //a's tree has less nodes than b's tree
                $parent[b] \leftarrow parent[b] + parent[a]$
                $parent[a] \leftarrow b$
                return (parent[b] = -n)
          else
                $parent[a] \leftarrow parent[a] + parent[b]$
                $parent[b] \leftarrow a$
                return (parent[a] = -n)
        end if
end
```

6.2.2 Implementing `istour()`

The `istour()` function invokes the `weighted_union()` function to determine whether the current basis structure at a node in the CSM search tree defines a tour. Edges corresponding to three types of variables could form part of the tour:

- Basic variables with a value of 1
- Variables in L that are fixed at an altered value
- Variables in U that are not fixed at an altered value

An algorithm for `istour()` is now provided. The type CSMnode refers to a node in the search tree used by the CSM. The function `csm_parent()` returns the parent node in the CSM search tree of the CSMnode given as parameter. Current U refers to the set of variables that are in U at the point this term is used. It is important to note that this includes variables originally in U that have since been temporarily fixed at 0.

```
boolean function istour(CSMnode current)
        initialise all nodes to have no parents
        for (all basic edges e) do
            if ($x_e = \frac{1}{2}$)
              then return false //definitely not a tour
            end if
            if ($x_e = 1$)
              then $(a, b) \leftarrow$ the nodes touched by e
                    if ($weighted\_union(a, b)$)
                      then return true
                    end if
            end if
        end
        while ($current \neq$ root node of CSM search tree) do
              if (current fixed a variable originally from L)
                then $(a, b) \leftarrow$ the nodes of the edge fixed in current
                      if ($weighted\_union(a, b)$)
                        then return true
                      end if
              end if
              $current \leftarrow csm\_parent(current)$
        end
        for (all edges e in current U) do
            if ($x_e$ is not fixed at 0)
              then $(a, b) \leftarrow$ the nodes touched by e
                    if ($weighted\_union(a, b)$)
                      then return true
                    end if
            end if
        end
        return false
end
```

Time complexity

The Union-Find algorithm without path compression or weight balancing has a worst case time complexity of $\Theta(n^2)$. Path compression and weight balancing improves the time complexity of the Union-Find algorithm. Analysis in [5] suggests that the time complexity when either one of these techniques are used, improves to $\Theta(nlog(n))$. The combined effect of the two techniques is an time complexity of $\Theta(nG(n))$, where $G(n)$ grows extremely slowly. The running time is therefore almost linear, but not quite. For a more complete analysis, see [27].

6.3 Conclusion

In this chapter efficient techniques were presented for use in the CSM for the TSP. Constant time techniques were presented to calculate which edge is incident on two given nodes, and to calculate which two nodes are connected by a given edge. The Union-Find algorithm has a time complexity almost as good as $\Theta(n)$ and is used to test if a basic solution defines a tour. As these algorithms scale well, they are suitable for use in large Traveling Salesman Problems.

Chapter 7

Implementation of the Column Subtraction Method for the Traveling Salesman Problem

In Chapter 5 the CSM for the TSP was presented. In Chapter 6 some of the techniques that were used in the method were presented. In this chapter some of the implementation details are presented along with execution times.

Problems from the TSPLIB were used to test the implementation. The TSPLIB is introduced in this chapter. The implementation details that are discussed include the technique to implement fractional calculations using integer variables, as well as the programming code level issues that arise because of this technique. The technique used for parallelisation is also presented. To test the parallel implementation, use was made of a cluster of workstations running the Message Passing Interface (MPI).

Several techniques can also be used to speed up the CSM for the TSP. Possible heuristics include techniques to limit the size of the CSM search tree in width and in height. Some of these techniques will be presented.

Execution times are reported for the uniprocessor case and for the program running on multiple processors. The speedup in the parallel cases are reported and compared with the results given in Chapter 2.

7.1 The TSPLIB

The TSPLIB [24] is a collection of TSP instances used in TSP research. It consists of more than 100 problems, ranging from small problems with less than 20 nodes, up to huge problems with more than 10 000 nodes. Optimal objective function values as well as optimal tours are available for some of the problems.

The problems of the TSPLIB are distributed in a standard format described in [26]. This format is also used to specify problem instances for related problems, like:

- Hamiltonian Cycle Problem
- Asymmetric Traveling Salesman Problem
- Sequential Ordering Problem
- Capacitated Vehicle Routing Problem

For the TSP, the edge costs can be specified in a number of ways. Most of the problems are specified in one of the following formats:

- Explicitly listed edge costs
- Edge costs are calculated as distances in Euclidean 2-space
- Edge costs are calculated as geographical distances

Edge costs are always integral, and the rounding is specified to use the `nint()` function. This is a FORTRAN function which is not present in C. The function `nint()` is however not described in the file format specification. In the TSPLIB FAQ [25] the following implementation is suggested as being equivalent:

```
int nint(double x)
{
    return (int)(x + 0.5);
}
```

For the implementation, a small compiler was developed for the TSPLIB format, which enabled seamless use of problems of all supported types. The compiler was implemented with the use of flex and bison.

7.2 Using integer variables for fractional quantities

It was already mentioned in Chapters 3 and 4 that variable values, reduced costs, and dual values can be stored as integers. This section will illustrate why it is worthwhile to handle the edge costs and objective function value in a similar way. Furthermore, this section will show how computations can be done within such an integer-only setup.

7.2.1 Edge costs and objective function value

Many of the quantities in the problem are already handled as integers. It will now be shown that all quantities discussed in the previous chapters can indeed be handled as integers.

As mentioned earlier, the problem instances from the TSPLIB work with integral edge costs. Edge costs can therefore easily be handled with integer variables. However, because we store dual values at double their real value, it simplifies calculations if edge costs are stored at double their real values too. Similarly, it is advantageous to store the objective function value at double its real value as well, and only dividing by two when output is required.

In its basic form, the objective function value, $z = \mathbf{cx}$. We know the edge costs in $\mathbf{c}$ are integral and the variable values in $\mathbf{x}$ are integers divided by two. z can therefore assume non-integral values when $\mathbf{x}$ contains non-integral values. However, if we store edge costs at double their real value, the result of the multiplication will always be integral and z can therefore be stored as an integer variable too.

We can therefore handle all the quantities that were discussed in the previous chapters as integers by storing double their real value where necessary, or where otherwise advantageous. This method is even used for intermediate quantities like the entries in the vectors $\mathbf{y}_e$

7.2.2 Arithmetic with doubled quantities

In the previous section it was indicated that all quantities in the implementation of the CSM for the TSP are handled with integer variables. In previous chapters, several basic arithmetic operations were defined on these quantities. It is therefore important to note how the operations are influenced by doubling of the values concerned.

For addition and subtraction, there are of course few consequences. The result of addition or subtraction of two doubled quantities is doubled as well. Care should be taken when comparing this result to constants (such as 1, which is implemented as 2 in the program). When constants are added or subtracted from doubled quantities, they should of course also be doubled.

For multiplication and division, the implications are slightly more subtle. Multiplying two doubled quantities, $c \leftarrow ab$, results in c storing *four* times the real value. Dividing two doubled quantities, $c \leftarrow \frac{a}{b}$, results in c storing the real value exactly. The intended use of the result of multiplication or division will determine to which answer the result must be scaled for correct use.

For division there is an extra implication: dividing two doubled quantities can result in a remainder which will be lost if not accounted for. Apart from handling the division as described in the previous paragraph, we have to know that the answer of the division can be represented without loss of information. It must therefore also be an integer divided by two.

For the implementation of the CSM for the TSP, this discussion on division is relevant in two cases. Both are in the dual simplex method (p. 46). The two quantities that are calculated are $v_e/\delta y_{re}$ and d_r/y_{rk} (the calculation of $(d_r-1)/y_{rk}$ is no different in this respect to d_r/y_{rk}). Both of these fractions have an entry of a vector $\mathbf{y}_e$ as the denominator. Firstly we will investigate the properties of this quantity.

The calculation of y_{rk} (or y_{re} in general) for the dual simplex method was discussed on p. 44. When all entries in β have been calculated, y_{re} can be calculated for any edge $e=(i,j)\in L\cup U$ with $y_{re}=\beta_i+\beta_j$. Possible values for entries in β were in $\{0,\pm\frac{1}{2},\pm 1\}$. An important observation can be made from the beta equations (see p. 44): for a specific basis structure, non-zero β values all have the same absolute value, i.e. non-zero β values are either all ± 1, or all $\pm\frac{1}{2}$. Possible values for y_{re} are thus $\{0,\pm\frac{1}{2},\pm 1,\pm 2\}$. A value of $\frac{3}{2}$ is not possible.

Let us consider division by each of these possible quantities:

- Dividing by $y_{re}=0$ is naturally not possible. (In the dual simplex method, $y_{re}=0$ indicates that x_e cannot be considered as an entering variable.)
- Dividing by the values $\pm\frac{1}{2}$ corresponds to multiplying by ± 2.
- Dividing by ± 1 only affects the sign.
- Dividing by ± 2 has to be considered more carefully. This could only happen when the edge r is outside the cycle of the 1-tree (see figure 3.3, p. 35), and all non-zero values of β are, therefore, outside the cycle of the 1-tree (see p. 45). The importance of this fact will soon be illustrated.

To consider the division involving the reduced costs ($\mathbf{v}$) we observe that there is a similarity to the values of y_{re} that was just discussed. The similarity arises because of the dual values (p. 29) that are calculated in a similar way to the β values. Similarly to the β value equations, the dual value equations will cause all dual values in a single 1-tree to either be all fractional (an odd integer divided by 2) or all integer (even integers in the implementation). Equation 3.2 defined the reduced costs as:

$$v_e = c_e - u_i - u_j \quad \forall e = (i, j) \in E$$

As mentioned before, the edge costs ($\mathbf{c}$) are also doubled in the implementation. The reduced cost of a non-basic edge connecting nodes within a single 1-tree is therefore an integer (c_e) minus the sum of two dual values (u_i and u_j), which are either both fractional (an odd integer divided by 2) or both integer. In this case the reduced cost will therefore be integer (an even integer in the implementation). As this is the only case in which $|y_{re}|$ can become as large as 2, no remainder will be lost as part of integer division in calculating the ratio v_e/y_{re}

Calculation of the ratio d_r/y_{rk} can be analysed in a similar way. We know that we only need to consider the case where $|y_{rk}| = 2$. It was mentioned to be a property of the basis (see section 3.2.1, p. 26) that basic variables only assume values of $\frac{1}{2}$ on the cycle of a 1-tree. In section 3.2.1 primal feasibility (which was not yet defined at that point) was assumed.

Even though the above mentioned calculations are during the dual simplex method, the same property in terms of fractional variable values exists. Fractional variable values are only possible on the cycle of a 1-tree. Calculating the ratio d_r/y_{rk} will therefore never cause the loss of a remainder.

7.3 Calculating node numbers from an edge number

In section 6.1.4 a constant time technique was presented to calculate which nodes are connected by a given edge. The improvement in asymptotic running time compared to the $O(log(n))$ method is of course good for solving larger problems, but the second reason for its importance is the fact that the calculation is performed many times during the execution of the program. Careful consideration of the implications of integer calculations allows for a more optimised implementation. The exact technique used is therefore presented here.

The formulas for calculating x and y, given k, is as follows (see section 6.1.4):

$$\begin{aligned} x &= n - \left\lceil \frac{1 + \sqrt{(2n-1)^2 - 8k}}{2} \right\rceil \\ y &= x + k + 1 - x(2n - x - 1)/2 \end{aligned}$$

The calculation of the square root and ceiling involves floating point arithmetic, which should be limited as far as possible. The calculation for x given above can be simplified to make use of fewer floating point operations. The following manipulations show the intermediate steps that are necessary to understand the simplified algorithm given later. For x:

$$\begin{aligned} x &= n - \left\lceil \frac{1 + \sqrt{(2n-1)^2 - 8k}}{2} \right\rceil \\ x &= n - \left\lceil \frac{2}{2} + \frac{\sqrt{(2n-1)^2 - 8k}}{2} \right\rceil \\ x &= n - 1 - \left\lceil \frac{\sqrt{(2n-1)^2 - 8k}}{2} \right\rceil \end{aligned}$$

The ceiling operation can be restricted to include only the square root function in the numerator, as long as the fraction is still integer. Taking the attributes of integer division into account, this requirement is unnecessary to enforce because the possible loss of information during the integer division still produces the correct result. In this way floating point arithmetic has been limited to fewer instructions.

$$x = n - 1 - \frac{\left\lceil \sqrt{(2n-1)^2 - 8k} \right\rceil}{2}$$

Although calculations for y can also be slightly simplified, it cannot be done without loss of information during the integer division. The only remaining improvement is to rewrite the formulas using two quantities c_1 and c_2, defined as follows to avoid unnecessary recalculation:

- $c_1 = 2n - 1$
- $c_2 = (2n - 1)^2$

The following algorithm for the function give_nodes() summarises all the improvements. The extra brackets in the calculation of y indicate the necessary order of calculation to avoid loss of information with integer division. Because c_1 is always odd, $(x(c_1 - x))$ will always be even, and no information will ever be lost during the division by 2.

```
**function** $give\_nodes(k, x, y)$
    $x = n - 1 - \left\lceil \sqrt{c_2 - 8k} \right\rceil / 2;$
    $y = x + k + 1 - (x(c_1 - x))/2;$
**end**
```

7.4 Parallelisation

In Chapter 2 the parallelisation of the CSM for the SPP was discussed. Good results were reported and superlinear speedup was achieved in some cases. Because of the computational complexity of the TSP, it is of course also worthwhile to consider parallelisation in order to solve bigger problems in reasonable time.

A similar technique is presented here for distributing the work on different processors. Each subtree rooted at a child of the root node of the CSM search tree is distributed from the master process to the next available process. When a solution is found that is better than the current best-known, the objective function value is sent to all the other processes, each of which updates its local copy of zIP, and shortens its list(s) of non-basic variables.

This has shown to be an ineffective technique for the smaller problems reported on here: the subtrees that are processed by the first few processors consume much more processor time than the other trees, with the effect that proper load balancing cannot be achieved. A more fine-grained approach should rectify this problem. One possibility is handing out grandchildren of the root node of the CSM search tree while those are big enough to justify the extra overhead.

7.5 Running times

The testing environment was a cluster of identical single processor workstations, each with the following configuration:

Hardware	
Processor	Pentium III 730MHz
Physical memory	128 MBytes
Interconnection	100 Mbits/s switched Ethernet
Software	
Operating System	Linux 2.4.20
File system	Network File System
	(file server not part of the cluster)
MPI Implementation	LAM/MPI 6.5.9
C compiler	GNU C Compiler 3.3

Running times for the five smallest problems that were solved are given here. The reported time is the average of three runs and is measured in seconds. Utilisation is the ratio of actual processor time used, to available processor time. Superlinear speedup was achieved with some problems (gr24 and fri26). The bad processor utilisation when running on sixteen or more processors could possibly explain the lack of good speedup in those cases. For more information about acceleration when solving Integer Programming problems with a cluster of workstations, see [6] and [12].

		Processors					
Problem		1	2	4	8	16	32
gr17	avg time	2.059	1.067	0.521	0.331	0.404	0.573
	speedup		1.930	3.955	6.228	5.097	3.594
	utilisation		1.000	1.000	0.869	0.451	0.242
gr24	avg time	2.596	0.337	0.262	0.223	0.200	0.237
	speedup		7.701	9.919	11.653	12.993	10.955
	utilisation		1.000	0.999	0.799	0.519	0.338
fri26	avg time	10.922	4.705	3.282	1.348	1.589	2.059
	speedup		2.321	3.328	8.103	6.874	5.305
	utilisation		1.000	1.000	0.815	0.491	0.253
bayg29	avg time	4.663	2.392	1.334	1.157	1.229	1.453
	speedup		1.949	3.497	4.029	3.794	3.208
	utilisation		1.000	1.000	0.827	0.510	0.275
bays29	avg time	4.203	2.918	2.425	2.241	2.408	3.126
	speedup		1.440	1.733	1.876	1.745	1.345
	utilisation		1.000	0.937	0.848	0.499	0.272
average	utilisation		1.000	0.987	0.832	0.494	0.276

7.6 Conclusion

The CSM has been implemented for the TSP and has also been parallelised. From the results reported here, it can be seen that the CSM implementation was not very efficient at solving larger TSP instances.

The difference between the SPP and the TSP affected the success of the CSM. Some of the fathoming rules that fathomed many nodes when applied to the SPP were inefficient when applied to the TSP - their cost outweighed their use.

Although good upper bounds (zIP) improved the running time, substancial amount of time was still spent on larger problems - even after the optimal solution has been found. More fathoming rules could possibly help to alleviate this problem.

Hoffman [15] suggested that removing integrality constraints from a combinatorial optimisation problem can possibly alter the structure of the problem too much. For the TSP, not only the integrality constraints, but also the 1-tree constraints were dropped. The approach to the solution was started from this substantially relaxed form. Such a relaxation alters the problem substantially. Hoffman suggests the consideration of extra constraints, which, for the TSP, could possibly correspond to the techniques presented by Padberg and Rinaldi [23].

Appendix A

Source code

The complete source code for the implementation of the CSM for the TSP is presented here. It was implemented in the C programming language. The compiler for the TSPLIB problem files were implemented using flex and bison (lex and yacc should work as well). GNU make was used for the build process.

The following files are provided:

- Makefile (p. 93) - specifies how build process must work
- global.h (p. 95) - header file for global.c
- simplex.h (p. 98) - header file for simplex.c
- primal.h (p. 99) - header file for primal.c
- dual.h (p. 100) - header file for dual.c
- parallel.h (p. 101) - header file for parallel.c
- csmtree.h (p. 102) - header file for csmtree.c
- istour.h (p. 104) - header file for istour.c
- set.h (p. 105)- header file for set.c
- csm.c (p. 106) - the main program and the implementation of the CSM

- global.c (p. 119) - global variables, defines, functions
- tsplib.l (p. 128) - lexical analyser (scanner) for the TSPLIB format
- tsplib.y (p. 131) - syntactical analyser for the TSPLIB format
- simplex.c (p. 146) - variables and functions common to both the primal and dual simplex methods
- primal.c (p. 152) - variables and functions for the primal simplex method
- dual.c (p. 162) - variables and functions for the dual simplex method
- parallel.c (p. 170) - variables and functions for parallel programming
- csmtree.c (p. 174) - variables and functions for dealing with the CSM search tree
- istour.c (p. 181) - variables and functions implementing the Union-Find algorithm to determine if a solution defines a tour
- set.c (p. 184) - variables and functions for dealing with the set U

A.1 Makefile

```
# Makefile for csm
# make
#   will make csm
# make compiler
#   will make a small program to parse tsplib files
# make clean
#   will remove coredumps, object files, backup files,
#   executables and generated files
# make linecount
#   will make clean and count the number of lines
# make testall
#   will run the testall script to do each of the problems
#   in the tsplib. Useful when combined with MAXNODES
#   defined in global.h
#

#flags for gcc
#DEFINES = -D__debug
#DEFINES += -D__debug_global
#DEFINES += -D__debug_simplex
#DEFINES += -D__debug_primal
#DEFINES += -D__debug_dual
#DEFINES += -D__debug_csm=2
#DEFINES += -D__debug_MPI
#DEFINES += -D__debug_mark
#DEFINES += -D__debug_csmtree
#DEFINES += -D__debug_istour
#DEFINES += -D__debug_tsplib
#DEFINES += -D__debug_U

PROFILE = -fprofile-arcs -fbranch-probabilities
NOG = -ftracer -fnew-ra -ffast-math
CFLAGS = -march=pentium4 -g -pg -Wall -Wshadow -Wunreachable-code $(DEFINES)
#CFLAGS = -O3 -march=pentium4 $(DEFINES)
LDFLAGS = -lm -pg
LDFLAGS = -lm
#CC = mpicc
YACC = bison

#flags for lex (flex)
LFLAGS =
#flags for yacc (bison)
YFLAGS = -d
#YFLAGS = -vd


all: csm
```

```
csm.o : csm.c *.h Makefile
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

%.o : %.c %.h Makefile global.h parallel.h
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
#or better:
#include *.d  - generated with gcc -M

### Start of things for tsplib compiler ###
lex.yy.c: tsplib.l global.h Makefile
    $(LEX) $(LFLAGS) tsplib.l

tsplib.tab.c: tsplib.y global.h Makefile
    $(YACC) $(YFLAGS) tsplib.y

compiler : lex.yy.c tsplib.tab.c global.h Makefile
    $(CC) $(CFLAGS) $(LDFLAGS) -DSTANDALONE=1 -o compiler tsplib.tab.c
### End of things for the tsplib compiler


tsplib.tab.o: tsplib.tab.c lex.yy.c global.h Makefile
    $(CC) $(CFLAGS) -DSTANDALONE=0 -Wno-unreachable-code -Wno-shadow \
            -Wno-unused-function -Wno-unused-label -c tsplib.tab.c
#To disable some warnings in the generated code.
#But there is handwritten code in there as well!

csm: tsplib.tab.o global.o simplex.o primal.o dual.o istour.o set.o csm.o \
            csmtree.o parallel.o

linecount: clean
    wc -l *.[chly]

clean:
    rm -f *~
    rm -f lex.yy.c
    rm -f *.tab.[ch] *.output
    rm -f *.o
    rm -f csm
```

A.2 global.h

```
/**
 * @file global.h
 * Global defines, variables and functions
 */

#ifndef __global_h
#define __global_h 1

#define MAXNODES      1000
/**< Used to limit the size of problems that are attempted */
#define ONE      2
#define HALF          1
#define ZERO          0

#if MPI
#define DEBUG(msg) printf("myId=%d, %s(%d)::%s() - %s", myId, __FILE__, __LINE__, \
        __FUNCTION__, msg);
#else
#define DEBUG(msg) printf("%s(%d)::%s() - %s", __FILE__, __LINE__, \
        __FUNCTION__, msg);
#endif
/**< Used for easily traceable debug output */

/** Used as bitmask to enable verbose debugging output */
#define VERBOSE 2

#define INDEX(i,j)  ((i<j)?Base[i]+j-i:Base[j]+i-j)
/**< Used to lookup the index of the edge touching nodes i and j */


// typedefs
typedef struct {
    int Parent, Child, Sibling;
    int Dist, Dual, Flow;
} NodeType;

typedef struct node * nodeptr;
struct node {        /* contains data for a node in the search tree */
    nodeptr parent, sibling, child;
    int next;    /* rc[next].column = number of next column to be
                fixed to 1 in a new child */
    int fixed;  /* number of column fixed to 1 to create this node */
    int sum;     /* sum of costs of columns fixed to 1 from root down
                to this node. Can be int for TSP */
    char * flows;    // the Node[].Flow values for this node
            // makes backtracking easier
};
```

```
/* global variables */
extern int pivots;
extern NodeType *Node;

extern int NumNodes, NumEdges;
extern int zLP;
extern int zIP;

extern int *Base;
extern int *cost;

extern nodeptr root;            // root node of the csm search tree
extern NodeType * NodeCopy;     // copy of Node[] at root
extern unsigned char * UCopy;       // copy of U[] at root
extern int **UPairCopy;         // copy of UPair[][] at root
extern int USizeCopy;           // copy of USize at root


/** solution stores the indices of the columns in the solution */
extern int *solution;           // array of length NumNodes
extern int solutions;
/** f is a scratch array used for different purposes in the code */
extern char *f;             // array of length NumNodes
/** cover stores a 1 for each covered row */
extern char *cover;         // array of length NumNodes
extern int covered;
extern char *mark;
/* Values in mark:
 *----------------
 * -1 denotes a column fixed at an altered value
 *      (1 for columns in L, 0 for columns in U)
 * 0 denotes a free (normal) column
 * 1 denotes a column fixed at it's initial value
 *      (0 for columns in L, 1 for columns in U)
 * 2 has the same meaning as 1, but when indicated with 2, the column's
 * status can't change anymore. This will indicate that the column's
 * reduced cost exceeds the difference between the best known goal function
 * value and the goal value of the LP relaxation. The column should never
 * be considered for inclusion in the basis again.
 */
void printParents();
void printDuals();
void printFlows();
void printU();
void printAll();
int CheckDuals();
void CheckDFeasibility();
int CheckVars();    //TODO:check:Just for primal feasiblity?
```

```
void give_nodes(int edge, int *x, int *y);
void give_nodes_init(); //just initialises to constants for optimising code

const char * double_value(int a);

void checkEquations();
void solutionprint();
void solution_found(nodeptr node, int subtree);

void time_init();
double seconds();

#endif //__global_h
```

A.3 simplex.h

```
/**
 * @file simplex.h
 * Defines, datastructures and interfaces for network simplex algorithm
 * These are used in both the primal and dual simplex algorithms
 */

#ifndef __simplex_h
#define __simplex_h 1

#if __debug
#define __debug_simplex 1
//#else
//#define __debug_simplex 0
#endif

extern int pivots;
extern long long dual_pivots;
extern int *Queue;
extern char *beta;

void simplex_init();
void simplex_free();

void WandYw(int U, int V, int *pW, int *pYw);
void ChangeFlows(int U, int V, int Sign, int W, int Yw, int Ratio);
void UpdateOneTrees(int U, int V, int OutNode, int NewFlow);
void UpdateDualsDistsSubTree(int U, int DualChange);
void UpdateDualsDistsOneTree(int U, int V, int CycleLength, int DualChange);

#endif
```

A.4 primal.h

```
/**
 * @file primal.h
 * Header file for primal.c that contains code for the primal simplex
 * algorithm in the column subtraction method
 */

#ifndef __primal_h
#define __primal_h 1


void SolveRMP2(int *ZRatioP, int *HRatioP, int *ORatioP);

#endif //__primal_h
```

A.5 dual.h

```
/**
 * @file dual.h
 * Header file for dual.c that contains code for the dual simplex
 * algorithm in the column subtraction method
 */

#ifndef __dual_h
#define __dual_h 1

#include "csmtree.h"

extern int dual_simplex;

int DualSimplex(nodeptr node, int *pZ);

#endif //__dual_h
```

A.6 parallel.h

```
/**
 * @file parallel.h
 * Header file for parallel.c that contains some of the code for parallisation
 */

#ifndef __parallel_h
#define __parallel_h 1

#define MPI      0         // set to 1 for message passing code

#define JOB_TAG       101
#define ZIP_TAG       102

extern int myId;
extern int numProcs;
extern int rootNext;

#if MPI
#include "mpi.h"

extern int zIPowner;              // which process found current zIP?

//Now the stuff for creating an MPI struct:
extern int blockcount;            // 6 ints in NodeType (see global.h)
extern MPI_Datatype type;
extern MPI_Aint displacement;
extern MPI_Datatype tsp_node;

extern MPI_Status status;
extern int finalize_count;        // count how many processes are finished

void parallel_arrays_init();          // register custom type, broadcast arrays
void update_zIP();            // see if another process found better zIP
void send_jobs();             // node 0 hand out jobs
void parallel_setRootNext();          // setRootNext for parallel environment
void parallel_finish();           // wait for all processes to finish
void fetch_solution();            // fetch solution from zIPowner

#endif //MPI

#endif //__parallel_h
```

A.7 csmtree.h

```
/**
 * @file csmtree.h
 * structures, prototypes and #defines for dealing with the csm search tree
 */

#ifndef __csmtree_h
#define __csmtree_h 1

#include "global.h"

#define RC_COLUMN(j) ((j < original_L_list_size) ? \
        L_list[j].column : \
        U_list[j-original_L_list_size].column)

/*#define RC_REDCOST(j) (j < L_list_size) ? \
        L_list[j].redcost : \
        U_list[j-original_L_list_size].redcost

//TODO:?? #define RC(j) (j < L_list_size) ? L_list[j] : U_list[j]
*/
struct pair {
    int column;
    int redcost;
};

extern struct pair * L_list, * U_list;  //Lists of non-basic variables at their lower &
                    //upper bounds respectively
extern int L_list_size, U_list_size;    //the sizes of the lists
extern int original_L_list_size;  //the size of the original L_list

extern long long total_created;
extern long long created;
extern int depth;
extern int maxdepth;
extern int treesize;
extern int maxtreesize;

//extern int rcsize;
extern int maxbranches;

extern long long fathom_solution;
extern long long fathom_early_solution;
extern long long fathom_dual;
extern long long fathom_marked;
extern long long fathom_inconsistent;
extern long long fathom_nocolumns;
extern long long fathom_onezerorows;
extern long long fathom_bounded;
extern long long fathom_depth;
extern long long fathom_branches;

nodeptr newnode(nodeptr parent, int next, int column);
nodeptr getnode();
void putnode(nodeptr ptr);
void nodes_free(nodeptr tmp_root);

int rc_init(int gap);
int rc_reduce(int gap);
void rc_free();

void fathom_statistics();

#endif
```

A.8 istour.h

```
/**
 * @file istour.h
 * Header for istour.c - Used to determine if a solution is a valid tour
 */

#ifndef __istour_h
#define __istour_h 1

#include "csmtree.h"

void istour_init();
void istour_free();

int istour(nodeptr node);

#endif
```

A.9 set.h

```
/**
 * @file set.h
 * datastructures and interfaces for handling the sets in RMP2
 */

#ifndef __set_h
#define __set_h 1

#include "simplex.h"


//#define AddToSet(i,Set)  Set[i/8]^=bits[i%8]
//#define DeleteFromSet(i,Set) Set[i/8]^=bits[i%8]
#define InSet(i,Set) (Set[i/8]&bits[i%8])
#define InU(id) (UArray[id/8] & bits[id%8])


extern int USize;
extern unsigned char *UArray;
extern int **UPair;     //UPair[0][i] -> UPair[1][i] is an edge in U
                //UPair[0][i] < UPair[1][i]
extern unsigned char bits[8];

void InitSet(int NumElts, unsigned char Set[]);
void U_init();
void U_free();
void AddToU(int i, int j, int id);
void DeleteFromU(int i, int j, int id);

#endif
```

A.10 csm.c

```
/**
 * @file csm.c
 * This is an implementation of the column subtraction algorithm for solving
 * the traveling salesman problem. It uses the continuous 2-matching problem
 * (RMP2) as relaxation which is solved using the generalised network simplex
 * algorithm.
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <limits.h>
#include <string.h>
#include <unistd.h>

#include <signal.h>
#include <errno.h>
#include <sysexits.h>

#include "global.h"
#include "simplex.h"
#include "primal.h"
#include "dual.h"
#include "set.h"
#include "csmtree.h"
#include "istour.h"
#include "parallel.h"


#if MPI
#include "mpi.h"
#endif

/** The following defines control which fathoming rules are used.
 *  Set to 1 to use. They are configurable because they are available,
 *  but have been found to not imporove execution times.
 */
#define CONFIG_RULE4          0
#define CONFIG_RULE5          0

int zero_ratio, half_ratio, one_ratio;
/* accounting variables for zero-, half- and one-ratio */

double elapsed, utilized, mintime, maxtime;
int *remember;
```

```
/* XXX non-elegant line needed to take compiler warning away. Problematic
 * since the tsplib compiler files are generated
 */
int readfile(char * filename);


static void csm_init()
{
    root = getnode();
    root->parent = root->sibling = root->child = NULL;
    root->next = 1; root->fixed = -1;
    root->sum = 0;
    mark = (char *) calloc(NumEdges, sizeof(char));
    cover = (char *) calloc(NumNodes, sizeof(char));
    solution = (int *) calloc(NumNodes, sizeof(int));
    NodeCopy = (NodeType *) malloc(NumNodes * sizeof(NodeType));
    UCopy = (unsigned char *) malloc(1+NumEdges/8 * sizeof(unsigned char));
    UPairCopy = (int **) malloc(2* sizeof(int *));
    UPairCopy[0]  = (int *) malloc(NumNodes * sizeof(int));
    UPairCopy[1]  = (int *) malloc(NumNodes * sizeof(int));

    istour_init();
    U_init();
    simplex_init();

    remember = (int *)malloc(NumEdges * sizeof(int));
    f = (char *) calloc(NumNodes, sizeof(char));
    if (f == NULL) {
        printf("out of memory **************** \n");
        exit(0);
    }
    give_nodes_init();
}

static void csm_free()
{
    free(f);
    free(remember);
    free(NodeCopy);
    free(UCopy);
    free(UPairCopy[0]);
    free(UPairCopy[1]);
    free(UPairCopy);

    free(solution);
    free(cover);
    free(mark);
    free(cost);

    nodes_free(root);
    rc_free();
    istour_free();
    U_free();
    simplex_free();
}

/** Determines which subtree should be traversed next */
static void setRootNext()
{
    int next;
    long long old_created;
#if MPI
    parallel_setRootNext();
#endif

    total_created += created;
    old_created = created;
    created = 0;

    next = root->next;
    root->next = rootNext;
    if (rootNext >= maxbranches) return;
    while (next < rootNext) {    /* TODO: only #if MPI ? */
#if __debug_mark
        DEBUG("marking column ");
        printf("%d", RC_COLUMN(next));
        if (InU(RC_COLUMN(next))) printf("(In U)");
        printf("\n");
#endif
        mark[RC_COLUMN(next)] = 2;
        next++;
    }
#if __debug_MPI
    /* We print
     *  - The number of nodes in the previous tree
     *  - The next root
     *  - maxbranches
     */
    printf("myId=%d, nodes=%lld, rootNext=%d, maxbranches=%d\n",
            myId, old_created, rootNext, maxbranches);
#endif
    if (myId == 0) {
        rootNext++;
        /* Do we need to skip over the gap between L and U? */
        if ((rootNext >= L_list_size) &&
                (rootNext < original_L_list_size)) {
            rootNext = original_L_list_size;
        }
    }
}
```

```
/** Fix a column into the optimal tour */
static void fixcolumn(int k, int node_a, int node_b)
{
    if (!InU(k)) {
        cover[node_a]++;
        cover[node_b]++;
        covered += 2;
    }
}

/** Remove a column from the optimal tour */
static void freecolumn(int k)
{
    int node_a, node_b;
    if (!InU(k)) {
        give_nodes(k, &node_a, &node_b);
        cover[node_a]--;
        cover[node_b]--;
        covered -= 2;
    }
}

/**
 * Does the quick checks to see if a node can be easily fathomed
 * Returns 1 iff node can not be easily fathomed
 * sets mark for column if not set
 * Also calls fixcolumn() if the node could not be easily fathomed
 */
static int checked(nodeptr node)
{
    int fixed;
    int node_a, node_b; /* temporary nodes */
    fixed = node->fixed;
#if MPI
    if (mark[fixed]) {
        fathom_marked++;
        return 0;
    }
#endif //MPI
    mark[fixed] = 1;

    /* check for consistency with fixed columns */
    give_nodes(fixed, &node_a, &node_b);
    if ((cover[node_a] >= 2) || (cover[node_b] >= 2)) {
        fathom_inconsistent++;  /* Rule 1 */
        return 0;
    }
#if MPI
    /* Updating zIP too often could actually hurt performance. As
```

```
     * relatively few solutions are found, the overhead of this
     * operation is quite costly.
     */
    //update_zIP();
#endif

#if CONFIG_RULE4
    /* check for worse solution */
    if (node->sum >= zIP) {
        fathom_bounded++;   /* Rule 4 */
        return 0;
    }
#endif

#if CONFIG_RULE5
    /* check for full cover - meaningless if we limit depth*/
    if (covered + 2 == 2 * NumNodes) {
        if (istour(node)) {
            zIP = (int) node->sum;
            maxbranches = rc_reduce(zIP - zLP);
            solution_found(node, root->next);
            fathom_solution++; /* TODO: correct variable? Rule 5 */
        }
        return 0;
    }
#endif

    mark[fixed] = -1;
    fixcolumn(fixed, node_a, node_b);
    return 1;
}

/** Restores the flow in the basis as it is stored in the parameter */
static void restore_flows(nodeptr node)
/* TODO: this function can perhaps be sped up if .Flow values are not kept
 * in the node struct but in an array so that a simple memcpy can do
 * what this function does.
 */
{
    int i;
    char *flows = node->flows;
    for (i = 0; i < NumNodes; i++) {
        Node[i].Flow = flows[i];
    }

}

/** Restores the basis to what it is at the root of the csm search tree */
static void restore_basis()
{
    //Restore Node[], U
    memcpy(UArray, UCopy, 1+NumEdges/8 * sizeof(unsigned char));
    memcpy(UPair[0], UPairCopy[0], NumNodes * sizeof(int));
    memcpy(UPair[1], UPairCopy[1], NumNodes * sizeof(int));
    memcpy(Node, NodeCopy, NumNodes * sizeof(NodeType));
    USize = USizeCopy;
}


/** Does the expensive checks to see if a node can be fathomed */
static int fathomed(nodeptr * pnode)
{
    int i, edge;
    int node_a, node_b;
    nodeptr node = *pnode;
    int tmp_zIP = node->sum;
    int return_value;
    int W, Yw;

//  for (i = 0; i < NumNodes; i++) f[i] = cover[i];
//  can rather use memcpy as long as f and cover are same type!
    memcpy(f, cover, NumNodes * sizeof(char));

    give_nodes(node->fixed, &node_a, &node_b);
    /* subtract right column node->fixed from right hand side
     * (column SUBTRACTION method)
     */
    WandYw(node_a, node_b, &W, &Yw);        // : 1, -1, 0, ONE);
    ChangeFlows(node_a, node_b, (InU(node->fixed))? -1 : 1, W, Yw, ONE);

    //save altered flows:
    for (i = 0; i < NumNodes; i++) node->flows[i] = Node[i].Flow;

    /* Calculate the objective function value that corresponds to the
     * basis as it is at the moment. We have to divide the result by
     * 2 otherwise we will be increasing tmp_zIP by 4 times the actual
     * change instead of the 2 times we want
     */
    for (i = 0; i < NumNodes; i++){
        tmp_zIP += (Node[i].Flow * cost[INDEX(i, Node[i].Parent)]) / 2;
    }
    for (i = 0; i < USize; i++){
        edge = INDEX(UPair[0][i], UPair[1][i]);
        if (-1 == mark[edge]) continue;
        tmp_zIP += cost[edge];
    }
#if MPI
    update_zIP();
#endif
    if (tmp_zIP >= zIP) {
        fathom_bounded++;   /* Rule 2 */
        return 1;
    }

    /* Lets check for a tour before Dualsimplex.
     * (noticed in bays29.tsp for example)
     */
    if (CheckVars() && istour(node)) {
        zIP = tmp_zIP;
        solution_found(node, root->next);
        maxbranches = rc_reduce(zIP - zLP);
        fathom_early_solution++;    /* Rule 3 */
        return 1;
    }


    /* now we know we have to do dual simplex */
    return_value =  DualSimplex(node, &tmp_zIP);
    fathom_dual += return_value;    /* Rules 7 and 8 */

    if (! return_value && istour(node)) {
        zIP = tmp_zIP;
        solution_found(node, root->next);
        maxbranches = rc_reduce(zIP - zLP);
        fathom_solution++;  /* Rule 9 */
        restore_basis();
        return 1;
    }
    restore_basis();
    return return_value;
}//fathomed

/** used to sort the solution variables */
static void sort(int * x, int first, int last)
{
    int left, right, key, temp;
    if (last - first >= 0) {
        key = x[first];
        left = first - 1;
        right = last + 1;
        while (1) {
            do right--; while (x[right] > key);
            do left++; while (x[left] < key);
            if (left >= right) break;
            temp = x[left];
            x[left] = x[right];
            x[right] = temp;
        }
        sort(x, first, left - 1);
        sort(x, right + 1, last);
    }
}

/** Calculate and print some information about the problem */
static void problemstats(char * filename)
{
    int i;
    double avecost = 0.0, stdcost = 0.0;
    int mincost, maxcost;
    time_t now = time(NULL);
    mincost = maxcost = cost[0];
    for (i = 1; i < NumEdges; i++ ) {
        avecost += cost[i];
        if (cost[i] < mincost)
            mincost = cost[i];
        else if (cost[i] > maxcost)
            maxcost = cost[i];
    }
    avecost /= NumEdges;
    stdcost = 0.0;
    for (i = 0; i < NumEdges; i++)
        stdcost += (cost[i] - avecost) * (cost[i] - avecost);
    stdcost /= NumEdges;
    stdcost = sqrt(stdcost);

    if (myId == 0) {
        printf("%sCSM solving %s on %d processors\n",
                ctime(&now), filename, numProcs);
        printf("NumNodes=%d NumEdges=%d cost range=%d "
            "cost stdev=%5.1f\n", NumNodes, NumEdges,
            maxcost-mincost, stdcost);
    }
}

/**
 * Reduce and calculate some statistics about solving of the problem
 */
static void solutionstats()
{
#if MPI
    int tmp;
    MPI_Reduce(&elapsed, &mintime, 1, MPI_DOUBLE, MPI_MIN, 0,
                    MPI_COMM_WORLD);
    MPI_Reduce(&elapsed, &maxtime, 1, MPI_DOUBLE, MPI_MAX, 0,
                    MPI_COMM_WORLD);
    MPI_Reduce(&elapsed, &utilized, 1, MPI_DOUBLE, MPI_SUM, 0,
                    MPI_COMM_WORLD);
    MPI_Reduce(&solutions, &tmp, 1, MPI_INT, MPI_SUM, 0,
                    MPI_COMM_WORLD);
    solutions = tmp;
    MPI_Reduce(&total_created, &tmp, 1, MPI_LONG_LONG_INT, MPI_SUM, 0,
                        MPI_COMM_WORLD);
    total_created = tmp;
    MPI_Reduce(&dual_simplex, &tmp, 1, MPI_INT, MPI_SUM, 0,
                        MPI_COMM_WORLD);
    dual_simplex = tmp;
    MPI_Reduce(&dual_pivots, &tmp, 1, MPI_INT, MPI_SUM, 0,
                        MPI_COMM_WORLD);
    dual_pivots = tmp;
    MPI_Reduce(&maxdepth, &tmp, 1, MPI_INT, MPI_MAX, 0,
                        MPI_COMM_WORLD);
    maxdepth = tmp;
    MPI_Reduce(&maxtreesize, &tmp, 1, MPI_INT, MPI_MAX, 0,
                        MPI_COMM_WORLD);
    maxtreesize = tmp;

    if (myId == 0) {
        utilized /= (numProcs * maxtime);
    }
#else
    mintime = maxtime = elapsed;
    utilized = 1;
#endif  //MPI
    if (myId != 0) return;
    printf("--------------------------\n");
    printf("Solution statistics\n");
    printf("--------------------------\n");
    printf("zIP=%d min=%7.4fs max=%7.4fs utilization=%7.4f\n",
        zIP/2, mintime, maxtime, utilized);
    printf("DualSimplex executed %d times\n", dual_simplex);
    printf("%lld Dual pivots in total\n", dual_pivots);
    printf("%lld subproblems in search tree\n", total_created);
    printf("%f pivots per DualSimplex\n",
            dual_pivots / (double)dual_simplex);
    printf("%f pivots per subproblem\n",
            dual_pivots / (double)total_created);
    printf("Maximum depth is %d\n", maxdepth);
    printf("Maximum tree size in memory is %d\n", maxtreesize);
}


/**
 * The column subtraction method.
 * The relaxation has been solved to optimality. Now we perform a
 * depth-first traversal of the search tree.
 */
static void csm()
{
    int i, k, next;
    nodeptr child, tmp_node, current;
```

```
/* prepare for column subtraction */
for (i = 0; i < NumNodes; i++) root->flows[i] = Node[i].Flow;
for (i = 0; i < myId; i++){
    mark[RC_COLUMN(i)] = 2;
}
root->next = myId;
rootNext = numProcs;    /* rootNext = next for first idle processor */
current = root;
total_created = 1;  /* lets count root explicitly */
created = 0;
do {
    next = current->next;
    /* create children of *current */
    while (next < maxbranches) {
        //If we reach the gap between L and U, we can skip:
        if (next == L_list_size) next = original_L_list_size;

        k = RC_COLUMN(next);
        child = newnode(current, next + 1, k);
        current->next = next + 1;
        if (checked(child)) {
            /* checked() fixes the column whether
             * it is fathomed or not
             */
            current = child;
            depth++;
            if (NumNodes/depth <= 4) {
                /*TODO: benchmark <=4 vs <4 etc, also
                 * consider placing after fathomed()
                 * or perhaps in checked()
                 */
                fathom_depth++;
                break;
            }

            if (depth > maxdepth) maxdepth = depth;
            if (fathomed(&current)) break;
            restore_flows(current); /* TODO: check!!! */
        }
        if (depth == 0) setRootNext();
        next = current->next;
    }
    child = current->child; /* rightmost child */
    /* now remove marks for variables fixed in children
     * of *current
     */
//  if (child != NULL && child->sibling != NULL) fathom_branches++;
    if (child != NULL) fathom_branches++;
    while (child != NULL) {
        if (mark[child->fixed] != 2) {
                mark[child->fixed] = 0;
            }
            tmp_node = child;
            child = child->sibling;
            putnode(tmp_node);
        }
        /* move up */
        if (current == root) break;
        freecolumn(current->fixed);
        if (mark[current->fixed] != 2) {
            mark[current->fixed] = 1;
        }
        tmp_node = current;
        current = current->parent;
        depth--;
        if (depth == 0) {
            setRootNext();
        }
        restore_flows(current);
#if MPI
        if (myId == 0) send_jobs();
#endif
    } while (1);
}//csm()

static void arrays_init()
{
#if MPI
    parallel_arrays_init(); /* also register NodeType struct with MPI */
#endif
    USizeCopy = USize;
    memcpy(NodeCopy, Node, NumNodes * sizeof(NodeType));
    memcpy(UCopy, UArray, 1+NumEdges/8 * sizeof(unsigned char));
    memcpy(UPairCopy[0], UPair[0], NumNodes * sizeof(int));
    memcpy(UPairCopy[1], UPair[1], NumNodes * sizeof(int));
}

int main(int argc, char ** argv)
{
#if MPI
    int tmp;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myId);
#endif  //MPI
    if (argc != 2) {
        if (myId == 0)
            fprintf(stderr, "Usage:\n%s <filename>\n", argv[0]);
        exit(1);
    }
    if (!readfile(argv[1])) {
        fprintf(stderr, "File not successfully read\n");
        exit(EX_UNAVAILABLE);
    }

    problemstats(argv[1]);
    csm_init();
    time_init();

    SolveRMP2(&zero_ratio, &half_ratio, &one_ratio);
    if (myId == 0) {
        printf("SolveRMP2 returns zLP=%.1f time=%5.3f pivots=%d\n",
            (double)zLP/2, seconds(), pivots);
    }

    if (istour(root)) {
        zIP = -zLP;
    }
#if MPI
    MPI_Allreduce(&zIP, &tmp, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    /* This call can be used to see if the RMP2 solution is perhaps
     * a tour. A negative zIP will work, as we reduce the minimum
     */
    zIP = tmp;
#endif
    if (zIP < 0) goto finish;
    if ((NumNodes > 23) && (zIP > 1.1 * zLP)) zIP = (int) (1.1 * zLP);
    /* 23 is just a hack to make gr17.tsp work again
     * it is the only problem with optimal zIP > 1.1 * zLP
     */
    arrays_init();

    /* Build list of non-basic variables with
     * reduced costs < gap. maxbranches is the index of the
     * first unconsidered variable in U
     */
    maxbranches = rc_init(zIP - zLP);

    csm();
    elapsed = seconds();
#if MPI
    parallel_finish();
#endif

finish:
    if (myId == 0) {
        if (zIP < 0) {
            printf("csm not necessary\n");
            zIP = -zIP;
        }
        sort(solution, 0, NumNodes - 1);
        solutionprint();
    }
    solutionstats();
    fathom_statistics();
#if MPI
    MPI_Finalize();
    if (myId == 0) sleep(1);
    /* hopefully we can get node 0's profile overwriting the others */
#endif
    csm_free();
    return 0;
}//main
```

A.11 global.c

```
/**
 * @file global.c
 * Global defines, variables and functions
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <limits.h>
#include <time.h>
#include <errno.h>


#include "global.h"
#include "csmtree.h"         // for depth
#include "set.h"
#include "parallel.h"

#if MPI
#include "mpi.h"
#endif

int NumNodes, NumEdges;
NodeType *Node;
int zLP = 0;
int zIP = INT_MAX;

int *Base;
int *cost;

nodeptr root;            // the root of the csm search tree
NodeType * NodeCopy;         // copy of Node[] at root
unsigned char * UCopy;       // copy of U[] at root
int **UPairCopy;         // copy of UPair[][] at root
int USizeCopy;           // copy of USize at root

int *solution;           // array of length NumNodes
int solutions;
char *f;             // array of length NumNodes
char *cover;             // array of length NumNodes
int covered = 0;
char *mark;          // array of length NumEdges

/* Two constants for optimising of give_nodes(); */
static int magic_constant1;      // 2*NumNodes - 1
static int magic_constant2;      // (2*NumNodes - 1) * (2*NumNodes - 1)
```

```
#if MPI
static double start;
#else
static clock_t start;        // for timing
#endif


void printParents()
{
    int i, counter = 1;
    char * printed;
    printed = (char *) calloc(NumNodes, sizeof(char));

    printf("0->");
    printed[0]++;
    i = Node[0].Parent;
    while (counter < NumNodes) {
        printf("%d", i);
        if (!printed[i]) {
            printed[i]++;
            counter++;
            i = Node[i].Parent;
            printf("->%d", i);
        }
        if (! printed[i]) {
            i = Node[i].Parent;
            printf("->");
        } else {
            printf("   ");
            while (printed[i] && counter < NumNodes) {
                i++;
                if (i == NumNodes) {
                    i = 1;  /* 0 is already printed */
                }
            }
        }
    }
    printf("\n");
    free(printed);
}

void printDuals()
{
    int i, right = 0;
    printf("Duals at double their value:\n");
    for (i = 0; i < NumNodes; i++, right = !right) {
        printf("Node[%3d].Dual=%4d\t", i, Node[i].Dual);
        if (right) printf("\n");
    }
    if (right) printf("\n");
}

void printFlows()
{
    int i, right = 0;
    for (i = 0; i < NumNodes; i++, right = !right) {
        printf("Node[%3d].Flow=%10s  ", i, double_value(Node[i].Flow));
        if (right) printf("\n");
    }
    if (right) printf("\n");
}

void printU()
{
    int i, right = 0;
    int edge;
    for (i = 0; i < USize; i++, right = !right) {
        edge = INDEX(UPair[0][i], UPair[1][i]);
        switch (mark[edge]) {
            case 2  : printf("(fixed2)"); break;
            case 1  : printf("(fixed1)"); break;
            case -1 : printf("(now  0)"); break;
            case 0  : printf("        "); break;
            default : fprintf(stderr, "Invalid entry in "
                      "mark[%d] = %d!\n", edge, mark[edge]);
        }
        printf("U:(%3d,%3d)  ", UPair[0][i], UPair[1][i]);
        if (right) printf("\n");
    }
    if (right) printf("\n");
}

void printCover()
{
    int i, right = 0;
    for (i = 0; i < NumNodes; i++, right = !right) {
        printf("cover[%d]=%4d\t", i, cover[i]);
        if (right) printf("\n");
    }
    if (right) printf("\n");
}

void printAll()
{
    printParents();
    printDuals();
    printFlows();
    printU();
    printCover();
```

```
}

int CheckDuals()
{
    int i, j;
    int correct = 1;
    for (i = 0; i < NumNodes; i++) {
        j = Node[i].Parent;
        if (Node[i].Dual + Node[j].Dual != cost[INDEX(i, j)]) {
            printf("Error in duals of (%d,%d) (%d, should be %d)\n",
                i,j, Node[i].Dual - Node[j].Dual,
                cost[INDEX(i,j)]);
            correct = 0;
        }
    }
    return correct;
}

/** Checks to see if current basis is dual feasible. */
void CheckDFeasibility()
{
    int i, dual_feasible;
    int rc;
    int a, b;
#if __debug_dual & VERBOSE
    printf("Checking dual feasibility...");
#endif
    dual_feasible = CheckDuals();
    for (i = 0; i < NumEdges; i++) {
        if (mark[i]) continue;
        give_nodes(i, &a, &b);
        rc = cost[i] - Node[a].Dual - Node[b].Dual;
        if ((InU(i) && rc > 0) || (!InU(i) && rc < 0)) {
            printf("\nColumn %d (%d,%d) not dual feasible - rc=%d",
                    i, a, b, rc);
            if (InU(i)) printf(" (in U)");
            else printf(" (in L)");
            printf("\ncost[%d]=%d, Node[%d].Dual=%d, Node[%d].Dual=%d",
                    i, cost[i], a, Node[a].Dual,
                    b, Node[b].Dual);
            dual_feasible = 0;
        }
    }
#if __debug_dual & VERBOSE
    if (dual_feasible) printf("Ok");
    printf("\n");
#endif
}

/** Check for primal feasibility. This function is called in normal operation,
 * not just for debugging
 */
int CheckVars() {
    int i;
    for (i = 0; i < NumNodes; i++) {
        if (Node[i].Flow < ZERO || Node[i].Flow > ONE) {
            return 0;
        }
    }
    return 1;
}


inline int my_sqrt(int x)
{
    int y = ceil(sqrt(x) - 1e-12);
    /* Lets avoid the branching by rewriting as an addition: */
    //if (!(y%2)) y++;
    //y += 1 - (y%2);
    //y += 1;
    /* rather just alter the return value from the caller, since its
     * modified anyway
     */
    return y;
}

/** Calculates which two nodes are connected by edge.
 * This function is called A LOT. There is usually an order of magnitude
 * more function calls to this code than anything else. Therefore the
 * hand-optimisation. See also above in my_sqrt(). See thesis for why this
 * works.
 */
void give_nodes(int edge, int *x, int *y)
{
/* Note: assigning *x to a temporary variable didn't increase performance -
 * it actually decreased it slightly
 */

    *x = NumNodes + (-2 - my_sqrt(magic_constant2 - 8*edge))/2;
    *y = *x + edge + 1 - (*x) * (magic_constant1 - *x)/2;

}

/** Just initialises two constants for optimising give_nodes() */
void give_nodes_init()
{
    magic_constant1 = 2*NumNodes - 1;
    magic_constant2 = magic_constant1 * magic_constant1;
}
```

```
const char * double_value(int a)
{
    static char output_string[] = " -0/2 (!!!)";
    switch (a) {
        /* XXX we don't need breaks because we return */
        case -ONE    : return("(-ONE)");
        case -HALF   : return("(-HALF)");
        case ZERO    : return("ZERO");
        case HALF    : return("HALF");
        case ONE     : return("ONE");
        default      : sprintf(output_string,
                       "%2d/2 (!!!)", a);
                     return(output_string);
    }
}

/** Check to see if all the constraints are satisfied */
void checkEquations()
{
    int i;
    int success = 1;
    /* Rows covered by fixed columns */
    memcpy(f, cover, NumNodes * sizeof(char));
    for (i = 0; i < NumNodes; i++) {
        f[i] <<= 1;
    }

    /* Basic Variables */
    for (i = 0; i < NumNodes; i++) {
        f[i] += Node[i].Flow;
        f[Node[i].Parent] += Node[i].Flow;
    }
    /* Non-basic variables in U */
    for (i = 0; i < USize; i++) {
        if (-1 == mark[INDEX(UPair[0][i], UPair[1][i])]) continue;
        f[UPair[0][i]] += ONE;
        f[UPair[1][i]] += ONE;
    }

    /* check for uncovered and overcovered rows */
    for (i = 0; i < NumNodes; i++) {
        if (f[i] != 2 * ONE) {
            fprintf(stdout, "**  equation %d != 2  (%s)**\n",
                    i, double_value(f[i]));
            success = 0;
        }
    }
    if (! success) {
        fprintf(stdout, " * * Equations not satisfied!!\n");
        printAll();
        exit(2);
    }
}

/** Displays the currently stored solution */
void solutionprint()
{
    int i, a, b, u_vars = 0;
    printf("solution : zIP = %d\n", zIP/2);
    printf("------------(0-based)(1-based)\n");
    if (numProcs > 1) printf("Solutions found by other processes \
                    are not retrieved!!!\n");
    for (i = 0; i < NumNodes; i++) {
        give_nodes(solution[i], &a, &b);
        printf("Edge %5d, (%3d,%3d)(%3d,%3d)", solution[i],
               a, b, a+1, b+1);
        if (InSet(solution[i], UCopy)) {
            printf(" (in U)");
            u_vars++;
        }
        printf("\n");
    }
    printf("%d solution variables in U\n", u_vars);
    printf("---------------\n");
}

/**
 * Called when a solution has been found.
 * Some sanity checking is done, and the current solution is stored
 */
void solution_found(nodeptr node, int subtree)
{
    int i;
    int column, nextvar = 0;
    int node_a, node_b; /* two temporary nodes */
    int sum = 0;
    nodeptr next = node;
#if MPI
    for (i = 0; i < numProcs; i++)
        if (i != myId)
            MPI_Send(&zIP, 1, MPI_INT, i, ZIP_TAG, MPI_COMM_WORLD);
#endif

    solutions++;
    memset(f, 0, NumNodes * sizeof(char));
    /* pick up variables fixed at 1 */
    while (next != root && next != NULL) {
        column = next->fixed;
        if (InSet(column, UCopy)) {
            /* we do all variables in U later */
            next = next->parent;
            continue;
        }
        sum += cost[column];
        solution[nextvar++] = column;
        give_nodes(column, &node_a, &node_b);
        f[node_a]++;
        f[node_b]++;
        next = next->parent;
    }
    /* Basic Variables at ONE */
    for (i = 0; i < NumNodes; i++) {
        if (Node[i].Flow) {
            column = INDEX(i, Node[i].Parent);
            solution[nextvar++] = column;
            sum += cost[column];
            f[i]++;
            f[Node[i].Parent]++;
        }
    }
    /* Non-basic variables in U */
    for (i = 0; i < USize; i++) {
        node_a = UPair[0][i];
        node_b = UPair[1][i];
        column = INDEX(node_a, node_b);
        if (-1 == mark[column]) continue;
        solution[nextvar++] = column;
        sum += cost[column];
        f[node_a]++;
        f[node_b]++;
    }
    /* Can't use the following if we use a heuristic to improve zIP,
     * or when we might update zIP because of other processes'
     * activities.
     */
/*  if (sum != zIP) {
        fprintf(stderr, "sum = %d, zIP = %d ****************\n",
                sum, zIP/2);
        fprintf(stderr, "edges in solution:%d\n", nextvar);
        solutionprint();
        printAll();
        exit(0);
    }
*/
    /* check for uncovered and overcovered rows */
    for (i = 0; i < NumNodes; i++) {
        if (f[i] != 2)
            fprintf(stderr, "**  row %d != 2  (%d)**\n", i, f[i]);
    }
    if (node != NULL) {
#if MPI
        printf("myId=%d, ", myId);
#endif
        printf("Tour found in subtree %d at depth %d, zIP=%d\n",
                        subtree, depth, zIP/2);
    } else {
        if (myId == 0)
            printf("Tour found during RMP2, zIP=%d\n", zIP/2);
    }
    if (fflush(stdout)) perror(strerror(errno));
}

void time_init()
{
#if MPI
    start = MPI_Wtime();
#else
    start = clock();
#endif
}

/** returns the seconds that passed since t0 */
double seconds()
{
#if MPI
    /* Wall clock time - we need to account idle time to calculate
     * utilisation
     */
    return MPI_Wtime() - start;
#else
    /* clock() returns user time (not system/idle etc). Usefull while
     * machine is used for other things, but it overflows after something
     * like 120 minutes.
     */
    return (clock() - start)/(double)CLOCKS_PER_SEC;
#endif
}
```

A.12 tsplib.l

```
%{
/* tsplib.l
 * A lexical analyser for problems of the tsplib
 * Use flex to generate the scanner. The scanner is used in conjunction with
 * tsplib.y, the parser
 *
 * Some problems were encountered with the files of the TSPLIB that doesn't
 * conform exactly to the format specified. Every effort was made to make all
 * the files work
 *
 * F Wolff (2003)
 */
#include <math.h>
#include "tsplib.tab.h"
%}

%option noyywrap
%option yylineno

DIGIT    [0-9]

%x string
%x integer
%x type
%x capacity
%x edge_weight_type
%x edge_weight_format
%x edge_data_format
%x node_coord_type
%x display_data_type

%%

<*>\n                {return NEWLINE;}

<INITIAL>[+-]?{DIGIT}+      {yylval = atoi(yytext); return INTEGER;}

<INITIAL>[+-]?{DIGIT}+"."{DIGIT}*([Ee][+-]?{DIGIT}+)? {yylval = atoi(yytext); return REAL;}

<INITIAL>NAME            {BEGIN(string); return NAME;}

<INITIAL>TYPE            {BEGIN(type); return TYPE;}
<type>TSP           {BEGIN(INITIAL); return TSP;}
<type>ATSP          {BEGIN(INITIAL); return ATSP;}
<type>SOP           {BEGIN(INITIAL); return SOP;}
<type>HCP           {BEGIN(INITIAL); return HCP;}
<type>CVRP          {BEGIN(INITIAL); return CVRP;}
```

```
<type>TOUR           {BEGIN(INITIAL); return TOUR;}


<INITIAL>COMMENT        {BEGIN(string); return COMMENT;}

<INITIAL>DIMENSION      {return DIMENSION;}

<INITIAL>CAPACITY       {return CAPACITY;}

<INITIAL>EDGE_WEIGHT_TYPE   {BEGIN(edge_weight_type); return EDGE_WEIGHT_TYPE;}
<edge_weight_type>EXPLICIT  {BEGIN(INITIAL); return EXPLICIT;}
<edge_weight_type>EUC_2D    {BEGIN(INITIAL); return EUC_2D;}
<edge_weight_type>EUC_3D    {BEGIN(INITIAL); return EUC_3D;}
<edge_weight_type>MAX_2D    {BEGIN(INITIAL); return MAX_2D;}
<edge_weight_type>MAX_3D    {BEGIN(INITIAL); return MAX_3D;}
<edge_weight_type>MAN_2D    {BEGIN(INITIAL); return MAN_2D;}
<edge_weight_type>MAN_3D    {BEGIN(INITIAL); return MAN_3D;}
<edge_weight_type>CEIL_2D   {BEGIN(INITIAL); return CEIL_2D;}
<edge_weight_type>GEO       {BEGIN(INITIAL); return GEO;}
<edge_weight_type>ATT       {BEGIN(INITIAL); return GEO;}
<edge_weight_type>XRAY1     {BEGIN(INITIAL); return GEO;}
<edge_weight_type>XRAY2     {BEGIN(INITIAL); return GEO;}
<edge_weight_type>SPECIAL   {BEGIN(INITIAL); return GEO;}

<INITIAL>EDGE_WEIGHT_FORMAT {BEGIN(edge_weight_format); return EDGE_WEIGHT_FORMAT;}
<edge_weight_format>FUNCTION        {BEGIN(INITIAL); return FUNCTION;}
<edge_weight_format>FULL_MATRIX     {BEGIN(INITIAL); return FULL_MATRIX;}
<edge_weight_format>UPPER_ROW       {BEGIN(INITIAL); return UPPER_ROW;}
<edge_weight_format>LOWER_ROW       {BEGIN(INITIAL); return LOWER_ROW;}
<edge_weight_format>UPPER_DIAG_ROW  {BEGIN(INITIAL); return UPPER_DIAG_ROW;}
<edge_weight_format>LOWER_DIAG_ROW  {BEGIN(INITIAL); return LOWER_DIAG_ROW;}
<edge_weight_format>UPPER_COL       {BEGIN(INITIAL); return UPPER_COL;}
<edge_weight_format>LOWER_COL       {BEGIN(INITIAL); return LOWER_COL;}
<edge_weight_format>UPPER_DIAG_COL  {BEGIN(INITIAL); return UPPER_DIAG_COL;}
<edge_weight_format>LOWER_DIAG_COL  {BEGIN(INITIAL); return LOWER_DIAG_COL;}

<INITIAL>EDGE_DATA_FORMAT   {BEGIN(edge_data_format); return EDGE_DATA_FORMAT;}
<edge_data_format>EDGE_LIST {BEGIN(INITIAL); return EDGE_LIST;}
<edge_data_format>ADJ_LIST  {BEGIN(INITIAL); return ADJ_LIST;}

<INITIAL>NODE_COORD_TYPE    {BEGIN(node_coord_type); return NODE_COORD_TYPE;}
<node_coord_type>TWOD_COORDS    {BEGIN(INITIAL); return TWOD_COORDS;}
<node_coord_type>THREED_COORDS  {BEGIN(INITIAL); return THREED_COORDS;}
<node_coord_type>NO_COORDS  {BEGIN(INITIAL); return NO_COORDS;/*DEFAULT*/}

<INITIAL>DISPLAY_DATA_TYPE  {BEGIN(display_data_type); return DISPLAY_DATA_TYPE;}
<display_data_type>COORD_DISPLAY {BEGIN(INITIAL); return COORD_DISPLAY;}
<display_data_type>TWOD_DISPLAY {BEGIN(INITIAL); return TWOD_DISPLAY;}
<display_data_type>NO_DISPLAY   {BEGIN(INITIAL); return NO_DISPLAY;}
```

```
<INITIAL>NODE_COORD_SECTION {return NODE_COORD_SECTION;}

<INITIAL>DEPOT_SECTION      {return DEPOT_SECTION;}

<INITIAL>DEMAND_SECTION     {return DEMAND_SECTION;}

<INITIAL>EDGE_DATA_SECTION  {return EDGE_DATA_SECTION;}

<INITIAL>FIXED_EDGES_SECTION    {return FIXED_EDGES_SECTION;}

<INITIAL>DISPLAY_DATA_SECTION   {return DISPLAY_DATA_SECTION;}

<INITIAL>TOUR_SECTION       {return TOUR_SECTION;}

<INITIAL>EDGE_WEIGHT_SECTION    {return EDGE_WEIGHT_SECTION;}

<string>[ a-zA-z0-9\t(){}\[\]`~!@#$%^&*\-_+=\\/\.,<>:;\"\']+ {
                    BEGIN(INITIAL); return STRING;
                }

 <*>[ ]+                ;
<INITIAL,type,edge_weight_type,edge_weight_format,node_coord_type,display_data_type>:    ;
<<EOF>>                 {return END;}
<INITIAL>EOF                {return END;}

.                   ;


%%
```

A.13 tsplib.y

```
/* tsplib.y
 * A bison grammar file for files of the tsplib
 *
 * TSPLIB website are available at
 * http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/
 * or
 * http://www.densis.fee.unicamp.br/~moscato/TSPBIB_home.html
 *
 * F Wolff (2003)
 */

%token STRING
%token INTEGER
%token REAL

%token NEWLINE
%token COLON

%token NAME

%token TYPE
%token TSP ATSP SOP HCP CVRP TOUR        //all the problem types

%token COMMENT
%token DIMENSION
%token CAPACITY
%token EDGE_WEIGHT_TYPE
%token EXPLICIT EUC_2D EUC_3D MAX_2D MAX_3D MAN_2D MAN_3D CEIL_2D GEO ATT XRAY1 XRAY2 SPECIAL
//all the edge_weight_types

%token EDGE_WEIGHT_FORMAT
%token FUNCTION FULL_MATRIX UPPER_ROW LOWER_ROW UPPER_DIAG_ROW LOWER_DIAG_ROW
%token UPPER_COL LOWER_COL UPPER_DIAG_COL LOWER_DIAG_COL
//all the edge_weight_formats

%token EDGE_DATA_FORMAT
%token EDGE_LIST ADJ_LIST           //all the edge_data_formats

%token NODE_COORD_TYPE
%token TWOD_COORDS THREED_COORDS NO_COORDS  //all the node_coord_types DEFAULT:NO_COORDS

%token DISPLAY_DATA_TYPE
%token COORD_DISPLAY TWOD_DISPLAY NO_DISPLAY    //all the display_data_types

%token END  //EOF

%token NODE_COORD_SECTION
```

```
%token DEPOT_SECTION
%token DEMAND_SECTION
%token EDGE_DATA_SECTION
%token FIXED_EDGES_SECTION
%token DISPLAY_DATA_SECTION
%token TOUR_SECTION
%token EDGE_WEIGHT_SECTION

%{

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <errno.h>
#include <sysexits.h>                //for aditional proper exit codes
#include <assert.h>

#include "global.h"
#include "lex.yy.c"
#include "simplex.h"

#if __debug
#define __debug_tsplib 1
//#else
//#define __debug_tsplib 0
#endif

#define ROUND(x) (int)(x + 0.5)
/*< nint() in the TSPLIB specification. This code is from the FAQ */
//#define STANDALONE 0               //used to test as a standalone app
/*** Defined in Makefile ***/

#if STANDALONE
int * cost = NULL;                //stores costs in upper-triangular form
int NumNodes, NumEdges;
#endif
//int counter = 0;

int **matrix    = NULL;              //used for explicit weight specifications
double **coords = NULL;              //used to store node coordinates
//int cost_index = 1;

int current_edge_weight_type    = -1;
int current_edge_weight_format  = -1;
int current_edge_data_format    = -1;
int current_node_coord_type = -1;
int current_display_data_type   = -1;

int edge_weight_type_warning    = 1;          //1 means warning should be given
int edge_weight_format_warning  = 1;
int edge_data_format_warning    = 1;
int node_coord_type_warning = 1;
int display_data_type_warning   = 1;

int success = 1;

void yyerror(char *);                //yyerror prints the message, with the line
                        //number where it was found. Sets success = 0
void yyassert(int condition, char * msg);   //checks if condition is true, if not calls
                        //yyerror(msg)
void custom_error(char *);           //just prints the message. Sets success = 0

void problem_init(int);              //set NumEdges, malloc cost
void all_free();
void coords_init();
void coords_free();
void matrix_init();
void matrix_free();

void calc_costs();
void explicit_costs();

void next_edge_weight(int weight);
void next_node_coord(int index, double x, double y, double z);

%}

%start tspfile

%%

tspfile : specification data END    {
#if __debug_tsplib
    printf("success\n");
#endif
    YYACCEPT;
    }
    | specification error       {yyerror("Error in data section"); YYABORT;}
//  | error             {yyerror("Error in specification section"); YYABORT;}
    ;

/*****************************************************************************/
specification : specification_line
    | specification specification_line
    ;
specification_line :
      name
    | type
    | comment
    | dimension
    | capacity
    | edge_weight_type
    | edge_weight_format
    | edge_data_format
    | node_coord_type
    | display_data_type
//  | specification error
//      {fprintf(stderr, "Error in specification section\n"); YYABORT;}
;

name     : NAME STRING NEWLINE
    ;
type     : TYPE type_param NEWLINE
    |TYPE error             {yyerror("Error in TYPE:"); YYABORT;}
        ;
type_param :
      TSP            {}
    | ATSP           {custom_error("TYPE: ATSP is unsupported"); YYABORT;}
    | SOP            {custom_error("TYPE: SOP is unsupported"); YYABORT;}
    | HCP            {custom_error("TYPE: HCP is unsupported"); YYABORT;}
    | CVRP           {custom_error("TYPE: CVRP is unsupported"); YYABORT;}
    | TOUR           {custom_error("TYPE: TOUR is unsupported"); YYABORT;}
       ;

comment : COMMENT STRING NEWLINE
    ;
dimension : DIMENSION INTEGER NEWLINE       {problem_init($2);}
    ;
capacity : CAPACITY INTEGER NEWLINE
        {yyerror("CAPACITY is used for CVRP, which is unsupported");
        YYABORT;
        }
    ;
edge_weight_type :
    EDGE_WEIGHT_TYPE edge_weight_type_param NEWLINE
    ;
edge_weight_type_param  :
      EXPLICIT  {current_edge_weight_type = EXPLICIT;}
    | EUC_2D    {current_edge_weight_type = EUC_2D; coords_init();}
    | EUC_3D    {current_edge_weight_type = EUC_3D; coords_init();}
    | MAX_2D    {current_edge_weight_type = MAX_2D; coords_init();}
    | MAX_3D    {current_edge_weight_type = MAX_3D; coords_init();}
    | MAN_2D    {current_edge_weight_type = MAN_2D; coords_init();}
    | MAN_3D    {current_edge_weight_type = MAN_3D; coords_init();}
    | CEIL_2D   {current_edge_weight_type = CEIL_2D; coords_init();}
    | GEO       {current_edge_weight_type = GEO; coords_init();}
    | ATT       {current_edge_weight_type = ATT;}
    | XRAY1     {current_edge_weight_type = XRAY1;}
    | XRAY2     {current_edge_weight_type = XRAY2;}
    | SPECIAL   {current_edge_weight_type = SPECIAL;}
    | STRING    {yyerror("Invalid edge_weight_type\n");}
    ;

edge_weight_format :
      EDGE_WEIGHT_FORMAT edge_weight_format_param NEWLINE
    ;
edge_weight_format_param :
      FUNCTION  {yyassert(current_edge_weight_type == EXPLICIT,
                  "EDGE_WEIGHT_TYPE should be EXPLICIT");
             current_edge_weight_format = EXPLICIT;
            }
    | FULL_MATRIX   {yyassert(current_edge_weight_type == EXPLICIT,
              "EDGE_WEIGHT_TYPE should be EXPLICIT");
            current_edge_weight_format = FULL_MATRIX;
            matrix_init();
            }
    | UPPER_ROW {yyassert(current_edge_weight_type == EXPLICIT,
              "EDGE_WEIGHT_TYPE should be EXPLICIT");
            current_edge_weight_format = UPPER_ROW;
            matrix_init();
            }
    | LOWER_ROW {yyassert(current_edge_weight_type == EXPLICIT,
              "EDGE_WEIGHT_TYPE should be EXPLICIT");
            current_edge_weight_format = LOWER_ROW;
            matrix_init();
            }
    | UPPER_DIAG_ROW{yyassert(current_edge_weight_type == EXPLICIT,
              "EDGE_WEIGHT_TYPE should be EXPLICIT");
            current_edge_weight_format = UPPER_DIAG_ROW;
            matrix_init();
            }
    | LOWER_DIAG_ROW{yyassert(current_edge_weight_type == EXPLICIT,
              "EDGE_WEIGHT_TYPE should be EXPLICIT");
            current_edge_weight_format = LOWER_DIAG_ROW;
            matrix_init();
            }
    | UPPER_COL {yyassert(current_edge_weight_type == EXPLICIT,
              "EDGE_WEIGHT_TYPE should be EXPLICIT");
            current_edge_weight_format = UPPER_COL;
            matrix_init();
            }
    | LOWER_COL {yyassert(current_edge_weight_type == EXPLICIT,
              "EDGE_WEIGHT_TYPE should be EXPLICIT");
            current_edge_weight_format = LOWER_COL;
            matrix_init();
            }
    | UPPER_DIAG_COL{yyassert(current_edge_weight_type == EXPLICIT,
              "EDGE_WEIGHT_TYPE should be EXPLICIT");
            current_edge_weight_format = UPPER_DIAG_COL;
            matrix_init();
            }
    | LOWER_DIAG_COL{yyassert(current_edge_weight_type == EXPLICIT,
                "EDGE_WEIGHT_TYPE should be EXPLICIT");
            current_edge_weight_format = LOWER_DIAG_COL;
            matrix_init();
            }
    | STRING    {yyerror("Invalid edge_weight_format\n");}
    ;

edge_data_format : EDGE_DATA_FORMAT edge_data_format_param NEWLINE
    ;
edge_data_format_param :
      EDGE_LIST {current_edge_data_format = EDGE_LIST;}
    | ADJ_LIST  {current_edge_data_format = ADJ_LIST;
                 yyerror("ADJ_LIST is not supported");
            }
    | STRING    {yyerror("Invalid edge_data_format");}
    ;
node_coord_type : NODE_COORD_TYPE node_coord_type_param NEWLINE
    ;
node_coord_type_param   :
      TWOD_COORDS   {current_node_coord_type = TWOD_COORDS;}
    | THREED_COORDS {current_node_coord_type = THREED_COORDS;}
    | NO_COORDS //default
    | STRING        {yyerror("Invalid node_coord_type");}
    ;
display_data_type : DISPLAY_DATA_TYPE display_data_type_param NEWLINE
    ;
display_data_type_param :
      COORD_DISPLAY {current_display_data_type = COORD_DISPLAY;}
    | TWOD_DISPLAY  {current_display_data_type = TWOD_DISPLAY;}
    | NO_DISPLAY    {}
    | STRING    {yyerror("Invalid display_data_type");}
    ;

/************************************************************************/
data    : data_section
    | data data_section     /* perhaps this is not always allowed */
    ;
data_section :
      node_coord_section
    | depot_section
    | demand_section
    | edge_data_section
    | fixed_edges_section
    | display_data_section
    | tour_section
    | edge_weight_section
//  | error         {yyerror("Error in data part of file\n");}
    ;
```

```
node_coord_section :
      NODE_COORD_SECTION NEWLINE node_coord_section_data    {calc_costs();}
    ;
node_coord_section_data :
      node_coord_section_data_line
    | node_coord_section_data node_coord_section_data_line
            ;
node_coord_section_data_line :
      INTEGER REAL REAL NEWLINE {next_node_coord($1, $2, $3, 0);}
    | INTEGER REAL INTEGER NEWLINE{next_node_coord($1, $2, $3, 0);}
    | INTEGER INTEGER REAL NEWLINE{next_node_coord($1, $2, $3, 0);}
    | INTEGER INTEGER INTEGER NEWLINE{next_node_coord($1, $2, $3, 0);}
    | INTEGER REAL REAL REAL NEWLINE{next_node_coord($1, $2, $3, $4);}
    ;

depot_section   : DEPOT_SECTION NEWLINE depot_section_data
    ;
depot_section_data  : /* empty */
    | depot_section_data INTEGER NEWLINE    {yyerror("depot_section is unsupported\n");}
    ;

demand_section  : DEMAND_SECTION NEWLINE demand_section_data
    ;
demand_section_data : /* empty */
    | demand_section_data INTEGER INTEGER NEWLINE
    ;

edge_data_section   :
      EDGE_DATA_SECTION NEWLINE edge_data_section_data
    ;
edge_data_section_data  : /* empty */
    | edge_data_section_data INTEGER INTEGER NEWLINE    //for EDGE_LIST
                                //ADJ_LIST unsupported
    ;

fixed_edges_section : FIXED_EDGES_SECTION NEWLINE fixed_edges_section_data
    ;
fixed_edges_section_data    :
      INTEGER INTEGER NEWLINE
    | fixed_edges_section_data INTEGER NEWLINE {assert ($2 == -1);}
    | fixed_edges_section_data INTEGER INTEGER NEWLINE
            //terminated by -1
    ;

display_data_section    :
      DISPLAY_DATA_SECTION NEWLINE display_data_section_data
    | DISPLAY_DATA_SECTION error    {yyerror("error in DISPLAY_DATA_SECTION");}
    ;
display_data_section_data :
      display_data_section_data_line NEWLINE
    | display_data_section_data display_data_section_data_line NEWLINE
    ;
display_data_section_data_line  :
      INTEGER REAL REAL
    | INTEGER REAL INTEGER
    | INTEGER INTEGER REAL
    | INTEGER INTEGER INTEGER
      //All only valid if DISPLAY_DATA_TYPE is TWOD_DISPLAY
    ;

tour_section    : TOUR_SECTION NEWLINE tour_section_data
    ;
tour_section_data   :
      INTEGER NEWLINE
    | tour_section_data INTEGER NEWLINE
    ;

edge_weight_section :
      EDGE_WEIGHT_SECTION NEWLINE edge_weight_section_data {
        edge_weight_format_warning = 1;
        explicit_costs();
    }
    ;
edge_weight_section_data :
      edge_weight_section_data_line NEWLINE {}
    | edge_weight_section_data edge_weight_section_data_line NEWLINE
    ;
edge_weight_section_data_line :
      INTEGER               {next_edge_weight($1);}
    | edge_weight_section_data_line INTEGER {next_edge_weight($2);}
    ;

%%
/***************************************************************************/
void next_upper_row_edge_weight(int weight)
{
    static int cost_index = 0;
    cost[cost_index++] = weight;
}

void next_full_matrix_edge_weight(int weight)
{
    static int matrix_x = 0, matrix_y = 0;
    yyassert(matrix_y < NumNodes, "Too many rows in matrix");
    matrix[matrix_x][matrix_y] = weight;
    matrix[matrix_y][matrix_x] = weight;
    matrix_x++;
    if (matrix_x >= NumNodes) {
        matrix_x = 0;
        matrix_y++;
    }
}

void next_lower_row_edge_weight(int weight)
{
    static int matrix_x = 0, matrix_y = 1;
    yyassert(matrix_y < NumNodes, "Too many rows in matrix");
    matrix[matrix_x][matrix_y] = weight;
    matrix[matrix_y][matrix_x] = weight;
    matrix_x++;
    if (matrix_x == matrix_y+1) {
        matrix_x = 0;
        matrix_y++;
    }
}

void next_lower_diag_row_edge_weight(int weight)
{
    static int matrix_x = 0, matrix_y = 0;
    if (matrix_x ==  matrix_y) {
        yyassert(weight == 0, "Entry on diagonal not 0");
        matrix_x = 0;
        matrix_y++;
        return;
    }
    matrix[matrix_y][matrix_x] = weight;
    matrix[matrix_x][matrix_y] = weight;
    matrix_x++;
}

void next_edge_weight(int weight)
{
    switch (current_edge_weight_format) {
    case UPPER_ROW      : next_upper_row_edge_weight(weight); break;
    case FULL_MATRIX    : next_full_matrix_edge_weight(weight); break;
    case LOWER_ROW      : next_lower_row_edge_weight(weight); break;
    case LOWER_DIAG_ROW : next_lower_diag_row_edge_weight(weight); break;
    /*
    case FUNCTION       :
    case UPPER_DIAG_ROW : //next_upper_diag_row_edge_weight(weight); break;
    case UPPER_COL      :
    case LOWER_COL      :
    case UPPER_DIAG_COL :
    case LOWER_DIAG_COL : break;
    */
    default          :
        if (edge_weight_format_warning)
            yyerror("Current EDGE_WEIGHT_FORMAT not specified/supported");
        edge_weight_format_warning = 0;
    }//switch
}//next_edge_weight()

void next_node_coord(int index, double x, double y, double z)
{
#if __debug_tsplib
    printf("next_node_coord %d\n", index);
#endif
    coords[index][0] = x;
    coords[index][1] = y;
    coords[index][2] = z;
}

void calc_costs()
{
    int i,j,k=0;
    double dx, dy, dz;
#if __debug_tsplib
    printf("calc_costs()\n");
#endif
    for (i = 1; i < NumNodes; i++){
        for (j = i+1; j <= NumNodes; j++){
            errno = 0;
            dx = coords[i][0] - coords[j][0];
            dy = coords[i][1] - coords[j][1];
            dz = coords[i][2] - coords[j][2];

            switch (current_edge_weight_type) {
            case EXPLICIT   :
                fprintf(stderr, "calc_cost() should not be \
                    called for EXPLICIT edge_weight_type\n");
                exit(EX_SOFTWARE);  //internal software error
                break;
            case EUC_2D :
            case EUC_3D :
                cost[k++] = ROUND(sqrt(dx*dx + dy*dy + dz*dz));
                if (errno != 0) perror(NULL);
                break;
            case CEIL_2D    :
                cost[k++] = ceil(sqrt(dx*dx + dy*dy + dz*dz));
                if (errno != 0) perror(NULL);
                break;
            case MAX_2D :
            case MAX_3D :

            case MAN_2D :
            case MAN_3D :

            case GEO    :
            case ATT    :
```

```
            case XRAY1  :
            case XRAY2  :
            case SPECIAL    :
            default : if (edge_weight_type_warning)
                custom_error("Current EDGE_WEIGHT_TYPE not implemented");
                edge_weight_type_warning = 0;
            }//switch

        }
    }
    coords_free();
#if __debug_tsplib
    printf("finished calc_costs()\n");
#endif
}//calc_costs()

/** explicit_costs() is called to copy the correct format of values into the cost
 * matrix.
 */
void explicit_costs()
{
    int i, j, k = 0;
#if __debug_tsplib
    printf("explicit_costs()\n");
#endif
    switch (current_edge_weight_format) {
    case UPPER_ROW      : break;    //already done;
    case FUNCTION       :
    case FULL_MATRIX    :
    case LOWER_ROW      :
    case UPPER_DIAG_ROW :
    case LOWER_DIAG_ROW :
    case UPPER_COL      :
    case LOWER_COL      :
    case UPPER_DIAG_COL :
    case LOWER_DIAG_COL :
        for (i=0; i < NumNodes - 1; i++){
            for (j = i + 1; j < NumNodes; j++){
                cost[k++] = matrix[i][j];
            }
        }
        break;
    default : if (edge_weight_format_warning)
        custom_error("Invalid/unsupported EDGE_WEIGHT_FORMAT");
        edge_weight_format_warning = 0;
    }
#if __debug_tsplib
    printf("finished explicit_costs()\n");
#endif
}
```

```
void coords_init()
{
    int j;
#if __debug_tsplib
    printf("Allocating memory for coordinates of %d nodes\n", NumNodes);
#endif
    coords = (double **) malloc((NumNodes+1) * sizeof(double));
    for (j = 1; j <= NumNodes; j++) {
        coords[j] = (double *) malloc(3 * sizeof(double));
    }
    if (coords == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(ENOMEM);
    }
}//coords_init()

void matrix_init()
{
    int j;
#if __debug_tsplib
    printf("Allocating memory for %dx%d matrix\n", NumNodes, NumNodes);
#endif
    matrix = (int **) malloc(NumNodes * sizeof(int));
    for (j = 0; j < NumNodes; j++) {
        matrix[j] = (int *) malloc(NumNodes * sizeof(int));
    }
    if (matrix == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(ENOMEM);
    }
}

/** This is called as soon as we know how what the dimension of the problem is
 */
void problem_init(int dimension)
{
    if (dimension > MAXNODES) {
        fprintf(stderr, "Problem too big\n");
        exit(ENOMEM);
    }
    NumNodes = dimension;
    if (NumNodes % 2)   //odd
        NumEdges = (NumNodes - 1) / 2 * NumNodes;
    else                //even
        NumEdges = NumNodes / 2 * (NumNodes - 1);
    cost = (int *) malloc (NumEdges * sizeof(int));
    if (cost == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(ENOMEM);
    }
}

void coords_free()
{
    int j;
    if (coords == NULL) return;
    for (j = 1; j <= NumNodes; j++)
        free(coords[j]);
    free(coords);
    coords = NULL;
}

void matrix_free()
{
    int j;
    if (matrix == NULL) return;
    for (j = 0; j < NumNodes; j++)
        free(matrix[j]);
    free(matrix);
    matrix = NULL;
}

void all_free()
{
#if STANDALONE
    free(cost);
    cost = NULL;
#endif
    coords_free();
    matrix_free();
}


void yyerror(char *msg)
{
    success = 0;
    fprintf(stderr, "%s - (Discovered in line %d)\n", msg, yylineno);
}

void yyassert(int condition, char * msg)
{
    if (!condition) {
        yyerror(msg);
    }
}


void custom_error(char *msg)
{
    success = 0;
    fprintf(stderr, "%s\n", msg);
}

/***************************************************************************/

#if ! STANDALONE
int readfile(char * filename)
{
    int i;
    FILE * f;
    f = fopen(filename, "r");
    if (f == NULL)
    {
        fprintf(stderr, "could not open %s\n", filename);
        exit(EIO);
    }
    yyin = f;
    success &= !yyparse();  //yyparse() returns 0 on success
    for (i = 0; i < NumEdges; i++) cost[i] *= 2;
        //the costs are doubled for our own implementation
#if __debug_tsplib
    //print out the first 100 costs (upper-triangular form)
    for (i = 0; i < NumEdges && i < 300; i++) printf("cost[%d]:%d\t", i, cost[i]);
#endif
    all_free();
    fclose(f);
    return success;
}


#else
int main(int argc, char ** argv)
{
#if __debug_tsplib
    int i;
#endif
    FILE * f;
    if (argc > 1) {
        f = fopen(argv[1], "r");
        if (f == NULL) {
            fprintf(stderr, "could not open %s\n", argv[1]);
            exit(EIO);
        }
        yyin = f;
    }
    success &= !yyparse();  //This is the function for to call bison's parser
                //yyparse() returns 0 on success
    printf("NumNodes : %d\tNumEdges : %d\n", NumNodes, NumEdges);
```

```
#if __debug_tsplib
    //print out the first 100 costs (upper-triangular form)
    for (i = 0; i < NumEdges && i < 100; i++) printf("cost[%d]:%d\t", i, cost[i]);
#endif

    all_free();
    fclose(f);
    if (success)
        return EX_OK;
    else
        return EX_UNAVAILABLE;  //we don't know exactly why we didn't succeed
}
#endif
```

A.14 simplex.c

```
/**
 * @file simplex.c
 * This file contains the code that is common to both the primal and
 * dual simplex methods. Also contains some useful functions for
 * debugging.
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <math.h>

#include "global.h"
#include "simplex.h"
#include "set.h"

int pivots = 0;
long long dual_pivots = 0;
int *Queue;
char *beta;

void simplex_init()
{
    int i, j;
    beta = (char *) malloc(NumNodes * sizeof(char));
    Node = (NodeType *) malloc(NumNodes * sizeof(NodeType));
    Base = (int *) malloc(NumNodes * sizeof(int));
    Queue = (int *) malloc(NumNodes * sizeof(int));

    Base[0] = -1; /* upper triangle of cost matrix is stored rowwise */
    for (j = 0, i = NumNodes - 1; j < NumNodes - 1; j++, i--) {
        Base[j + 1] = Base[j] + i;
    }
}

void simplex_free()
{
    free(beta); beta = NULL;
    free(Node); Node = NULL;
    free(Base); Base = NULL;
    free(Queue);    Queue = NULL;
}


/**
 * Temporary function to calculate where (if) two nodes' path meet
 * on their way to the cycle. Should do better later.
 */
void WandYw(int U, int V, int *pW, int *pYw)
{
    int YofPath[2], NodeOnPath[2];
    int Path = -1;
    int Y, Yw, W = -1;
    int dist, SumDist, k, i, j, r;

    NodeOnPath[0] = U;
    NodeOnPath[1] = V;
    YofPath[0] = YofPath[1] = 1;
    if (Node[U].Dist < 0) {          /* U is on cycle */
        if (Node[V].Dist < 0) dist = 0;
        else {               /* V is not on cycle */
            dist = Node[V].Dist;
            Path = 1;
        }
        SumDist = dist;
    } else {                 /* U not on cycle */
        if (Node[V].Dist < 0) {      /* V on cycle */
            dist = Node[U].Dist;
            SumDist = dist;
            Path = 0;
        } else {             /* V is also not on cycle */
            dist = Node[U].Dist - Node[V].Dist;
            if (dist > 0) Path = 0;
            else {
                Path = 1;
                dist = -dist;
            }
            SumDist = Node[U].Dist + Node[V].Dist;
        }
    }
    if (dist > 0) {
        /* move towards cycle on longest path */
        i = NodeOnPath[Path];
        Y = YofPath[Path];
        k = 0;
        while (k < dist) { /* Trace longest of Pu and Pv */
            Y = -Y;
            i = Node[i].Parent;
            k++;
        }
        NodeOnPath[Path] = i;
        YofPath[Path] = Y;
    }
    /* now same distance away from cycle(s) on both paths */
    dist = Node[NodeOnPath[0]].Dist;
    if (dist < 0) dist = 0;
    while (NodeOnPath[0] != NodeOnPath[1] && dist > 0) {
        /* move towards cycle(s) on both paths */
        k = 0;
        while (k < 2) { /* Trace Pu and Pv */
            i = NodeOnPath[k];
            NodeOnPath[k] = Node[i].Parent;
            YofPath[k] = -YofPath[k];
            k++;
        }
        dist--;
    }
    if (NodeOnPath[0] == NodeOnPath[1]) {
        W = NodeOnPath[0];
        if (YofPath[0] == -YofPath[1]) Yw = 0;
        else {  /* trace common path to cycle */
            Yw = YofPath[0];
        }
    } else {/* (NodeOnPath[0] != NodeOnPath[1] && (dist == 0) */
        /* at different nodes on cycle(s) on both paths */
        i = NodeOnPath[0];
        j = NodeOnPath[1];
        r = i;
        if (Node[i].Dist == Node[j].Dist) { /* if cycles of same size */
            dist = 0;
            do {
                r = Node[r].Parent;
                dist++;
            } while (r != j && r != i);
        }

        if (r == j) { /* i and j on same cycle */
            if ((SumDist + dist) % 2 == 1) {
                Path = 0;
            } else {
                Path = 1;
            }
            i = NodeOnPath[Path];
        ////     Y = YofPath[Path];
            W = NodeOnPath[1 - Path];
            Yw = 0;
        }/* otherwise i and j in different cycles */
    }

    *pW = W;
    *pYw = Yw;
}

void ChangeFlows(int U, int V, int Sign, int W, int Yw, int Ratio)
{
    int I, K, R, Y;
    if (W >= 0) { /*PathNode[1] and PathNode[2] in same one-tree */
        I = U;
        for (K = 0; K < 2; K++) { /* trace to W on Pu and Pv */
            Y = Sign * Ratio; /* Sign = +- 1 */
            while (I != W) {
                Node[I].Flow -= Y;
                I = Node[I].Parent;
                Y = -Y;
            }
            I = V;
        }
        if (Yw != 0) {
            Y *= 2; I = W;
            while (Node[I].Dist > 0) { /* trace from W to cycle */
                Node[I].Flow -= Y;
                I = Node[I].Parent;
                Y = -Y;
            }
            Y /= 2;
            R = I;
            do {
                Node[I].Flow -= Y;
                I = Node[I].Parent;
                Y = -Y;
            } while (I != R);
        }
    } else { /* U and V on different cycles */
        I = U;
        for (K = 0; K < 2; K++) {
            Y = Sign * Ratio;       /* Sign = +- I */
            while (Node[I].Dist > 0) { /* trace to cycle */
                Node[I].Flow -= Y;
                I = Node[I].Parent ;
                Y = -Y;
            }
            R = I;
            Y /= 2;
            do {
                Node[I].Flow -= Y;
                I = Node[I].Parent;
                Y = -Y;
            } while (I != R);
            I = V;
        }
    }
} /* ChangeFlows */

/**
 * Makes U a child of V.
 * Also makes all the parents of U part of the subtree rooted at V
 * precondition : node OutNode must be on backpath of node U
 */
void UpdateOneTrees(int U, int V, int OutNode, int NewFlow)
{
//taken from Geldenhuys - uses fewer variables
    int K, W;
    int OldFlow;

    do {
        W = Node[U].Parent;
        /* remove child U of W */
        if (Node[W].Child == U)
            Node[W].Child = Node[U].Sibling;
        else {
            K = Node[W].Child;
            while (Node[K].Sibling != U) K = Node[K].Sibling;
            Node[K].Sibling = Node[U].Sibling;
        }
        /* insert U as child of V */
        Node[U].Parent = V;
        Node[U].Sibling = Node[V].Child;
        Node[V].Child = U;

        OldFlow = Node[U].Flow;
        Node[U].Flow = NewFlow;
        NewFlow = OldFlow;

        V = U; U = W; /* move up on backpath */
    } while (V != OutNode);
} /* UpdateOneTrees */

/**
 * Change dual variables and distances for arborescence with root U.
 */
void UpdateDualsDistsSubTree(int U, int DualChange)
{
    int Head, Tail, Parent, Child, OldTail, NewDist;

    Parent = Node[U].Parent;
    NewDist = (Node[Parent].Dist < 0) ? 1 : Node[Parent].Dist + 1;
    Node[U].Dual += DualChange;
    Node[U].Dist = NewDist;
    Head = 0;
    Tail = 0;
    Queue[Tail] = U;
    do {
        DualChange = -DualChange;
        NewDist++;
        OldTail = Tail;
        do {
            Parent = Queue[Head];
            Child = Node[Parent].Child;
            while (Child > -1) {
                Node[Child].Dual += DualChange;
                Node[Child].Dist = NewDist;
                Queue[++Tail] = Child;
                Child = Node[Child].Sibling;
            }
        } while (++Head <= OldTail);
    } while (Tail != OldTail); /* no new nodes added to queue */
#if __debug_simplex
    printDuals();
#endif
} /* UpdateDualsDistsSubTree */

/**
 * Updates the .Dual and .Dist values in the basis tree.
 * Sets distances for all nodes on new cycle = -CycleLength. Updates
 * the duals and distances for every arborescence with a node of the
 * new cycle as root.
 */
void UpdateDualsDistsOneTree(int U, int V, int CycleLength, int DualChange)
{
    int Child, Start;
    Start = V;
    do {
        Node[V].Dist = -CycleLength;
        Node[V].Dual += DualChange;
        Child = Node[V].Child;
        DualChange = -DualChange;
        while (Child > -1) {
            if (Child != U)
                UpdateDualsDistsSubTree(Child, DualChange);
            Child = Node[Child].Sibling;
        }
        U = V;
        V = Node[V].Parent;
    } while (V != Start);
} /* UpdateDualsDistsOneTree */
```

A.15 primal.c

```
/**
 * @file primal.c
 * This file contains code for the simplex method that only applies
 * to the primal simplex part of column subtraction method.
 * Almost entirely the original of Leenen
 */

#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

#include "global.h"
#include "primal.h"
#include "simplex.h"
#include "set.h"
#include "parallel.h"
#include "istour.h"

/**
 * Builds the initial basic solution.
 * Finds a tour by arbitrary insertion. If NumNodes is odd, this tour can
 * be used as starting basis for the primal simplex. If even, one basic
 * is transferred to U to make the initial 1-tree have a cycle with an odd
 * number of nodes.
 */
static void FindTour()
{
    int *Cst;
    int Inc, Min, I, J, K, R, S, Dist, Sign, Start;
    int Sum;

    Cst = (int *) malloc(NumNodes * sizeof(int));
    Node[0].Parent = 1;
    Node[1].Parent = 0;
    Cst[0] = cost[0];
    Cst[1] = cost[0];
    for (J = 2; J < NumNodes; J++) {
        /* find the node S such that the cost of inserting node J
         * between Node[S].Parent and S is a minimum over all such
         * insertions
         */
        Min = INT_MAX;
        K = 0;
        do {
            I = Node[K].Parent;
            Inc = cost[INDEX(I,J)] + cost[INDEX(J, K)] - Cst[K];
            if (Inc < Min) {Min = Inc; S = K;}
            K = I;
        } while (K != 0);
        /* replace edge (Node[S].Parent,S) with the edges
         * (Node[S].Parent,J) and (J,S)
         */
        R = Node[S].Parent;
        Node[J].Parent = R;
        Cst[J] = cost[INDEX(R, J)];
        Node[S].Parent = J;
        Cst[S] += Min - Cst[J];
    }

    /* compute tour value and set child, sibling and flow values */
    for (J = 0; J < NumNodes; J++) {
        zLP += Cst[J];
        I = Node[J].Parent;
        Node[J].Sibling = -1;
        Node[J].Flow = ONE;
        Node[I].Child = J;
    }
    zIP = zLP;

    solution_found(NULL, 0);
    if ((NumNodes % 2) == 0) {
        /* makes edge between node 1 and Node[1].Child non-basic
         * with flow of 1 and edge between node Node[1].Parent
         * and Node[1].Child basic with flow of 0
         */
        K = Node[0].Child;
        Node[0].Child = -1;
        Node[0].Sibling = K;
        I = Node[0].Parent;
        AddToU(K, 0, K - 1);
        Node[K].Parent = I;
        Node[K].Flow = 0;
        Cst[K] = cost[INDEX(K, I)];
    }
    J = Node[0].Parent;
    /* compute dual value of node J on cycle */
    Sum = 0;
    Sign = 1;
    Start = J;
    do {
        I = Node[J].Parent;
        Sum += Sign * Cst[J];
        J = I;
        Sign = -Sign;
    } while (J != Start);
    Node[J].Dual = Sum / 2;
    /* compute dual distance values */
    if ((NumNodes % 2) == 1) Node[J].Dist = -NumNodes;
    else {
        Node[J].Dist = -(NumNodes - 1);
        Node[0].Dual = Cst[0] - Node[J].Dual;
        Node[0].Dist = 1;
    }
    Dist = Node[J].Dist;
    for (K = 1; K < -Dist; K++){
        I = Node[J].Parent;
        Node[I].Dual = Cst[J] - Node[J].Dual;
        Node[I].Dist = Dist;
        J = I;
    }
    free(Cst);
} /* FindTour */

/**
 * Performs the ratiotest to select leaving edge.
 * W is set to a node in both backpaths only if U and V are in the
 * same  onetree while CycleLength is set to the length of the new
 * cycle only if such a cycle can be formed when adding the edge
 * between U and V to the basis graph
 */
static void RatioTest(int U, int V, int Sign, int *PathP, int *OutNodeP,
        int *WP, int *CycleLengthP, int *YwP, int *RatioP)
{
    int YofPath[2], NodeOnPath[2];
    int I, Dist, SumDist, K, MinPath, R, J;
    int Path = -1, OutNode, W = -1, CycleLength = 0;
    int Y, MinRatio, Yw, Ratio;


    Ratio = ONE;
    NodeOnPath[0] = U;
    NodeOnPath[1] = V;
    YofPath[0] = YofPath[1] = 1;
    if (Node[U].Dist < 0) {         /* U is on cycle */
        if (Node[V].Dist < 0) Dist = 0;
        else {                /* V is not on cycle */
            Dist = Node[V].Dist;
            Path = 1;
        }
        SumDist = Dist;
    } else {                  /* U not on cycle */
        if (Node[V].Dist < 0) {     /* V on cycle */
            Dist = Node[U].Dist;
            SumDist = Dist;
            Path = 0;
        } else {              /* V is also not on cycle */
            Dist = Node[U].Dist - Node[V].Dist;
            if (Dist > 0) Path = 0;
            else {
                Path = 1;
                Dist = -Dist;
            }
            SumDist = Node[U].Dist + Node[V].Dist;
        }
    }
    if (Dist > 0) {
        /* move towards cycle on longest path */
        I = NodeOnPath[Path];
        Y = YofPath[Path];
        K = 0;
        while (K < Dist && Ratio == ONE) { /* Trace longest of Pu and Pv */
            if (Y == Sign) {
                if (Node[I].Flow == ZERO) {
                    Ratio = ZERO;
                    OutNode = I;
                }
            } else {
                if (Node[I].Flow == ONE) {
                    Ratio = ZERO;
                    OutNode = I;
                }
            }
            Y = -Y;
            I = Node[I].Parent;
            K++;
        }
        NodeOnPath[Path] = I;
        YofPath[Path] = Y;
    }
    /* now same distance away from cycle(s) on both paths */
    Dist = Node[NodeOnPath[0]].Dist;
    if (Dist < 0) Dist = 0;
    while (NodeOnPath[0] != NodeOnPath[1] && Dist > 0 && Ratio == ONE) {
        /* move towards cycle(s) on both paths */
        K = 0;
        while (K < 2 && Ratio == ONE) { /* Trace Pu and Pv */
            I = NodeOnPath[K];
            if (YofPath[K] == Sign) {
                if (Node[I].Flow == ZERO) {
                    Ratio = ZERO;
                    OutNode = I;
                    Path = K;
                }
            } else {
                if (Node[I].Flow == ONE) {
                    Ratio = ZERO;
                    OutNode = I;
                    Path = K;
                }
            }
            NodeOnPath[K] = Node[I].Parent;
            YofPath[K] = -YofPath[K];
            K++;
        }
        Dist--;
    }
    if (Ratio == ONE) {
        if (NodeOnPath[0] == NodeOnPath[1]) {
            W = NodeOnPath[0];
            CycleLength = (SumDist - 2 * Dist) + 1;
            if (YofPath[0] == -YofPath[1]) Yw = 0;
            else {  /* trace common path to cycle */
                Y = Yw = YofPath[0];
                MinPath = (Node[U].Dist < Node[V].Dist) ? 0 : 1;
                I = W;
                while (Dist > 0 && Ratio > ZERO) { /* Trace Pw - Cw */
                    if (Y == Sign) {
                        if (Ratio > Node[I].Flow / 2) {
                            Ratio = Node[I].Flow / 2;
                            OutNode = I;
                            Path = MinPath;
                        }
                    } else {
                        if (Ratio > (ONE - Node[I].Flow) / 2) {
                            Ratio = (ONE - Node[I].Flow) / 2;
                            OutNode = I;
                            Path = MinPath;
                        }
                    }
                    Y = -Y;
                    I = Node[I].Parent;
                    Dist--;
                }
                if (Ratio > ZERO) {
                    R = I;
                    MinRatio = (Node[R].Flow == HALF) ? HALF : ZERO;
                    do { /* trace common cycle */
                        if (Y == Sign) {
                            if (Ratio > Node[I].Flow) {
                                Ratio = Node[I].Flow;
                                OutNode = I;
                                Path = MinPath;
                            }
                        } else {
                            if (Ratio > (ONE - Node[I].Flow)) {
                                Ratio = (ONE - Node[I].Flow);
                                OutNode = I;
                                Path = MinPath;
                            }
                        }
                        Y = -Y;
                        I = Node[I].Parent;
                    } while (I != R && Ratio > MinRatio);
                }
            }
        } else {/* (NodeOnPath[0] != NodeOnPath[1] && (Dist == 0) */
            /* at different nodes on cycle(s) on both paths */
            I = NodeOnPath[0];
            J = NodeOnPath[1];
            R = I;
            if (Node[I].Dist == Node[J].Dist) { /* if cycles of same size */
                Dist = 0;
                do {
                    R = Node[R].Parent;
                    Dist++;
                } while (R != J && R != I);
            }

            if (R == J) { /* I and J on same cycle */
                if ((SumDist + Dist) % 2) { // == 1
                    Path = 0;
                    CycleLength = SumDist - Dist - Node[R].Dist + 1;
                } else {
                    Path = 1;
                    CycleLength = SumDist + Dist + 1;
                }
                I = NodeOnPath[Path];
                Y = YofPath[Path];
                W = NodeOnPath[1 - Path];
                Yw = 0;
                MinRatio = (Node[I].Flow == HALF) ? HALF: ZERO;
                do {
                    if (Y == Sign) {
                        if (Ratio > Node[I].Flow) {
                            Ratio = Node[I].Flow;
                            OutNode = I;
                        }
                    } else {
                        if (Ratio > (ONE - Node[I].Flow)) {
                            Ratio = (ONE - Node[I].Flow);
                            OutNode = I;
                        }
                    }
                    Y = -Y;
                    I = Node[I].Parent;
                } while (I != W && Ratio > MinRatio);
            } else { /* I and J in different cycles */
```

```
                K = 0;
                while (K < 2 && Ratio > ZERO) {
                    R = NodeOnPath[K];
                    if (Node[R].Flow != HALF) {
                        Y = YofPath[K];
                        I = R;
                        do { /* trace cycle */
                            if (Y == Sign) {
                                if (Node[I].Flow == ZERO) {
                                    Ratio = ZERO;
                                    OutNode = I;
                                    Path = K;
                                }
                            } else {
                                if (Node[I].Flow == ONE) {
                                    Ratio = ZERO;
                                    OutNode = I;
                                    Path = K;
                                }
                            }
                            Y = -Y;
                            I = Node[I].Parent;
                        } while (I != R && Ratio > ZERO);
                    }
                    K++;
                }
            } /* I and J on different cycles */
        }
    }
#if __debug_primal
    printf("Path=%d, OutNode=%d, Ratio=%d, CycleLength=%d, W=%d\n",
            Path, OutNode, Ratio, CycleLength, W);
#endif
    *PathP = Path;
    *OutNodeP = OutNode;
    *WP = W;
    *CycleLengthP = CycleLength;
    *YwP = Yw;
    *RatioP = Ratio;
} /* RatioTest */

/**
 * Searches for nonbasic edges with *UP as the "endnode" onwards.
 * Returns the endnodes *UP and *VP of the incoming edge. If the
 * solution is optimal, ReducedCost is 0. Otherwise ReducedCost is the
 * reduced cost of the entering edge.
 */
static void SelectEnteringEdge(int *UP, int *VP, int *ReducedCostP)
{
    int BaseU, StartU, U, J, K;
    int DualU, MaxD = 0, D;

    U = *UP;
    StartU = U;
    do {
        DualU = Node[U].Dual;
        BaseU = Base[U] - U;

        for (J = 0; J < U; J++) {
            K = Base[J] + U - J;     //don't need to test if U < J
            D = Node[J].Dual + DualU - cost[K]; //-reduced cost
            if (InU(K)) D = -D;
            if (D > MaxD) {
                MaxD = D;
                *VP = J;
            }
        }
        for (J = U + 1; J < NumNodes; J++) {
            K = BaseU + J;
            D = Node[J].Dual + DualU - cost[K];
            if (InU(K)) D = -D;
            if (D > MaxD) {
                MaxD = D;
                *VP = J;
            }
        }
        if (U == NumNodes - 1) U = 0; else U++;
    } while (MaxD == 0 && U != StartU);
    if (MaxD > 0) {
        *UP = (U == 0) ? NumNodes - 1 : U - 1;
        *ReducedCostP = InU(INDEX(*UP, *VP)) ? MaxD : -MaxD;
    }
    else {
        *ReducedCostP = 0;
    }
} /* SelectEnteringEdge */

/**
 * Primal network simplex algorithm solving the RMP2 relaxation.
 */
void SolveRMP2(int *ZRatioP, int *HRatioP, int *ORatioP)
{
    int Sign, Path, W, V, U, OldU, OutNode, CycleLength, K;
    int ReducedCost;
    int NewFlow;
    int ZRatio = 0, HRatio = 0, ORatio = 0, i = 0, Ratio, Yw;
    int Zchange = 0;

    FindTour();
    U = 0;
    /*U is a node of the current incoming edge */
    SelectEnteringEdge(&U, &V, &ReducedCost);
    while (abs(ReducedCost) > 0) {
        i++;

        K = INDEX(U , V);
        if (InU(K)) {
            Sign = -1;
            DeleteFromU(U, V, K);
        } else {
            Sign = 1;
        }
        RatioTest(U, V,Sign, &Path, &OutNode, &W, &CycleLength,
                       &Yw, &Ratio);
        if (Path) { /* == 1 - we want U furthest from cycle*/
            OldU = U;
            U = V;
            V = OldU;
        }        /*change flows*/
        if (Ratio > ZERO) {
            Zchange += Sign * (ReducedCost * Ratio) / 2;
            /* We have to divide by 2 because Zchange is increased
             * by 4 times its real value instead of the 2 times its
             * value that we want
             */
            ChangeFlows(U, V, Sign, W, Yw, Ratio);
        }
        /*change basis structure*/
        if (Ratio == ONE) {
            if (Sign == 1) AddToU(U, V, K);
            ORatio++;
        } else {
            (Ratio)? HRatio++ : ZRatio++;
            NewFlow = (Sign == 1) ? Ratio : ONE - Ratio;
            if (Node[OutNode].Flow == ONE)
                AddToU(OutNode, Node[OutNode].Parent,
                    INDEX(OutNode, Node[OutNode].Parent));
            UpdateOneTrees(U, V, OutNode, NewFlow);
            if (CycleLength > 0)
                UpdateDualsDistsOneTree(U, V, CycleLength,
                        ReducedCost / 2);
            else
                UpdateDualsDistsSubTree(U, ReducedCost);
        }
        if (Path) U = OldU; //if Path == 1
        if (U == NumNodes - 1) U = 0; else U++;

#if __debug_primal
        CheckDuals(NumNodes);
        CheckVars(NumNodes);
#endif
        SelectEnteringEdge(&U, &V, &ReducedCost);
        pivots++;
        if ((pivots % numProcs == myId) &&
            (zLP + Zchange < zIP) && istour(NULL)) {
            zIP = zLP + Zchange;
            solution_found(NULL, -1);
        }
    }
    zLP += Zchange;     //Zchange already devided by 2 each time
    *ZRatioP = ZRatio;
    *HRatioP = HRatio;
    *ORatioP = ORatio;
#if __debug_primal
    printf("Optimal solution:\n");
    printAll();
    CheckDFeasibility();
#endif
} /* SolveRMP2 */
```

A.16 dual.c

```
/**
 * @file dual.c
 * The implementation of the dual simplex method for use in
 * the column subtraction method.
 *
 * Based on the code by Geldenhuys
 * TODO: Reference parameters int *param_W and int local_W?
 *       *param_W = local_W;
 *       return;
 *       -Check SelectEnteringColumn and others
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>

#include "global.h"
#include "simplex.h"
#include "csmtree.h"
#include "set.h"

int dual_simplex;

/**
 * Return the row with the most infeasible value
 */
static int leavingrow(int *ChangeFlow, int *IncreaseIndic)
{
    int i, r;
    int maxDev = 0, flow;

    r = -1;
    for (i = 0; i < NumNodes; i++) {
        flow = Node[i].Flow;
        if (flow < ZERO) {
            if (-flow > maxDev) {
                *ChangeFlow = flow;
                maxDev = -flow;
                r = i;
                *IncreaseIndic = 1;
            }
        } else if (flow > ONE) {
            flow -= ONE;
            if (flow > maxDev) {
                *ChangeFlow = flow;
                maxDev = flow;
                r = i;
                *IncreaseIndic = 0;
            }
        }
        if (maxDev >= 2) return i;  //TODO: test effect more
    }
    return r;
} /* leavingrow */

static void betaInSubTree(int R)
{
    int head, tail, oldTail, child, parent;
    int nextBeta;

    head = 0;
    tail = 0;
    Queue[tail] = R;
    do {
        oldTail = tail;
        do {
            parent = Queue[head];
            nextBeta = -beta[parent];
            child = Node[parent].Child;
            while (child > -1) {
                beta[child] =  nextBeta;
                Queue[++tail] = child;
                child = Node[child].Sibling;
            }
        } while (++head <= oldTail);
    } while (tail != oldTail);
} /* betaInSubTree */

static void calcBeta(int R)
{
/* Calculate beta = 2 * (row R of B inverse) */
    int j, k, child, start, nextBeta;
    memset(beta, 0, NumNodes * sizeof(char));
    if (Node[R].Dist < 0) {
        j = Node[R].Parent;
        start = j;
        k = R;
        nextBeta = HALF;
        do {
            beta[j] = nextBeta;
            nextBeta = -nextBeta;
            child = Node[j].Child;
            while (child > -1) {
                if (child != k) {
                    beta[child] = nextBeta;
                    betaInSubTree(child);
                }
                child = Node[child].Sibling;
            }
            k = j;
            j = Node[k].Parent;
        } while (j != start);
    } else {
        beta[R] = ONE;
        betaInSubTree(R);
    }
} /* calcBeta */


/**
 * Selects which column should enter the basis.
 * returns a 0 if no entering column could be selected
 *  (Set corresponding to IncreaseIndic is empty)
 * returns a 1 if entering column found (Successful)
 */
static int selectEnteringColumn(int IncreaseIndic, int leaving,
        int * min_ratio_edge, int * min_y_re, int * min_rc,
        int * min_W, int * min_Yw)
        /* min_y_re is y_rk in dissertation */
{
    int j;       /* j is currently considered edge */
    int a, b;    /* nodes touched by edge j = (a,b) */
    int rc;      /* reduced cost of j */
    int y_re;
    int delta;
    int ratio, min_ratio = INT_MAX;
    delta = IncreaseIndic ? -1 : 1;
    *min_W = -1;

    a = 0;
    b = 0;
    calcBeta(leaving);
    /* TODO: rather use lists L and U because:
     * - they are shorter (fewer variables);
     * - we can mostly eliminate the iteration over permanently fixed vars;
     * - don't need to check for basic variables;
     * - we can possibly break out of loops early if we detect an increase
     *   in Z that will be too big.
     * Having to call give_nodes() makes it too slow, though
     */
    for (j = 0; j < NumEdges; j++) {
        b++;
        if (b == NumNodes) {
            a++;
            b = a+1;
        }
```

```
/* if j is a fixed column, continue (can't change it) */
if (mark[j]) continue;

/* We check for valid values of y_re first. y_re == 0
 * very often. Only NumNodes edges will be basic (see
 * later test)
 */
y_re = beta[a] + beta[b];
if (! y_re) continue;
y_re *= delta;

/* the check for equality in the following lines are
 * superfluous if we do it before, but it makes it
 * easier to test the effect of testing for equality
 * seperately
 */
if (InU(j)) {
    if (y_re >= ZERO) continue;
} else {
    if (y_re <= ZERO) continue;
}

/* Check if variable is basic: */
if (Node[a].Parent == b || Node[b].Parent == a) continue;
/* TODO: check:
 * XXX: This seems to cause dual infeasiblity sometimes:
 */
if ((cover[a] >= 2) || (cover[b] >= 2)) {
    continue;
}
/* calculate ratio, compare to best */
rc = cost[j] - Node[a].Dual - Node[b].Dual;
/* Because we are dividing rc by y_re, (both of which
 * are stored at 2 times their actual value), we might
 * loose information because of the integer division.
 * Therefore we have to multiply rc with 2. This gets
 * rid of integer division problems.
 *
 * There is no remainder possible with the division in
 * the case where |y_re| == 2 (4 in our representation).
 * See disertation for detail.
 */
ratio = 2*rc / y_re;
if (ratio < min_ratio) {
    min_ratio = ratio;
    *min_y_re = y_re * delta;
    *min_rc = rc;
    *min_ratio_edge = j;
    if (min_ratio == 0) break;
```

```
        }
    }
    if (min_ratio == INT_MAX) return 0;
    return 1;
}  /* SelectEnteringColumn */

/**
 * orientate (*N1,*N2) to have R on the backpath of *N1.
 * If no new cycle is formed by replacing (R,Node[R].Parent)
 * by (*N1,*N2) then *CycleLength is set to 0, otherwise
 * *CycleLength is set to length of new cycle
 */
static void orientate(int *N1, int *N2, int R, int *CycleLength)
{
    int NodeOnPath[2], Path;
    int Dist, SumDist, I, K, U, V, S, J;

    *CycleLength = 0; /* no new cycle is formed */
    U = *N1;
    V = *N2;
    NodeOnPath[0] = U;
    NodeOnPath[1] = V;
    if (Node[U].Dist < 0) {
        if (Node[V].Dist < 0) {
            Dist = 0;
            Path = 0;
        } else {
            Dist = Node[V].Dist;
            Path = 1;
        }
        SumDist = Dist;
    } else {
        if (Node[V].Dist < 0) {
            Dist = Node[U].Dist;
            SumDist = Dist;
            Path = 0;
        } else {
            Dist = Node[U].Dist - Node[V].Dist;
            if (Dist > 0)
                Path = 0;
            else {
                Path = 1;
                Dist = -Dist;
            }
            SumDist = Node[U].Dist + Node[V].Dist;
        }
    }
    if (Dist > 0) { /* trace longest backpath to a cycle */
        I = NodeOnPath[Path];
        for (K = 0; K < Dist && I != R; K++) I = Node[I].Parent;
        if (K < Dist) { /* encountered R before completing trace */
            if (Path == 1) {
                *N1 = V;
                *N2 = U;
            }
            return;
        }
        NodeOnPath[Path] = I;
    }
    /* now NodeOnPath[0].Dist = NodeOnPath[1].Dist */
    Dist = Node[NodeOnPath[0]].Dist;
    if (Dist < 0)  Dist = 0;
    while (NodeOnPath[0] != NodeOnPath[1]  && Dist > 0) {
        I = NodeOnPath[0];
        if (I == R) {
            return;    /* R on path 0 */
        } else {
            NodeOnPath[0] = Node[I].Parent;
            I = NodeOnPath[1];
            if (I == R) {
                *N1 = V;
                *N2 = U;
                return; /* R on path 1 */
            } else {
                NodeOnPath[1] = Node[I].Parent;
                Dist--;
            }
        }
    }
    if (NodeOnPath[0] == NodeOnPath[1]) { /* new cycle is formed */
        *CycleLength = (SumDist - Dist * 2) + 1;
        if (Node[U].Dist < Node[V].Dist) Path = 0; else Path = 1;
    } else { /* on cycle on both backpaths */
        /* NodeOnPath[0] != NodeOnPath[1] and Dist = 0 */
        I = NodeOnPath[0];
        J = NodeOnPath[1];
        S = I;
        Dist = 0;
        do {
            S = Node[S].Parent;
            Dist++;
        } while (S != J && S != I);
        if (S == J) { /* I and J on same cycle */
            if ((SumDist + Dist) % 2) {
                Path = 0;
                *CycleLength = SumDist - Dist - Node[R].Dist + 1;
            } else {
                Path = 1;
                *CycleLength = SumDist + Dist + 1;
            }
        } else { /* I and J on different cycles */
            S = J;
            /* search for R in backpath 2 */
            do S = Node[S].Parent; while (S != R && S != J);
            if (S == R) Path = 1; else Path = 0;
        }
    }
    if (Path == 1) {
        *N1 = V;
        *N2 = U;
    }
} /* orientate */

/**
 * Attempts to restore primal feasibility
 * Returns 1 if the node can be fathomed
 * Returns 0 otherwise
 * @param node the current node in the csm search tree
 * @param Z the current objective function value at node before invoking
 *       the dual simplex method
 */
int DualSimplex(nodeptr node, int *pZ)
{
    int leaving, ChangeFlow, IncreaseIndic, entering;
    int y_rk, rc, W, Yw;
    int node_a, node_b;
    int CycleLength;
    int Z = *pZ;
    /* ChangeFlow, y_rk and Yw store their value*2 */
    dual_simplex++;
    while ((leaving = leavingrow(&ChangeFlow, &IncreaseIndic)) >= 0) {
#if __debug_dual
        checkEquations();
#endif
        if (! selectEnteringColumn(IncreaseIndic, leaving, &entering,
                 &y_rk, &rc, &W, &Yw)) {
            return 1;
        }
        Z += (ChangeFlow * rc) / y_rk;
        if (Z >= zIP) {
            return 1;
        }
        /* Because we are dividing ChangeFlow by y_rk, (both of which
         * are stored at 2 times their actual value), the result will
         * not be two times the value it represents (the change in
         * flow). We want it to be 2 times the value it represents -
         * we will be updating flow values with it.  Therefore we have
         * to multiply ChangeFlow with 2.
         *
         * As y_rk might go up to +-4 (2*(+-2)), do we need to do
         * more?
         *
         * No, |y_rk| will only be 4 outside the cycle of the 1-tree,
         * where the flow values will be integer (even integers in our
         * representation). Therefore the division will work if we
         * mulitply with 2 only
         */
        ChangeFlow *= 2;
        /* Update the flow values for all nodes: */
        give_nodes(entering, &node_a, &node_b);
        /* TODO:can remove above call if selectEnteringColumn() can
         * also return the two nodes it had anyway
         */
        ChangeFlow /= y_rk;
        WandYw(node_a, node_b, &W, &Yw);
        ChangeFlows(node_a, node_b, 1, W, Yw, ChangeFlow);
        /* Update basis structure: */
        if (InU(entering)) {
            DeleteFromU(node_a, node_b, entering);
            ChangeFlow += ONE;
        }
        if (! IncreaseIndic) {
            AddToU(leaving, Node[leaving].Parent,
                INDEX(leaving, Node[leaving].Parent));
        }
        orientate(&node_a, &node_b, leaving, &CycleLength);
        UpdateOneTrees(node_a, node_b, leaving, ChangeFlow);
        if (CycleLength > 0) {
            UpdateDualsDistsOneTree(node_a, node_b, CycleLength,
                    rc / 2);
        } else {
            UpdateDualsDistsSubTree(node_a, rc);
        }

#if __debug_dual
        CheckDFeasibility();
#endif
        dual_pivots++;
    }
#if __debug_dual
    if (! CheckVars())
        printf("NOT PRIMAL FEASIBLE AFTER dualsimplex()!!!!\n");
    checkEquations();
#endif
    *pZ = Z;
    return 0;
} /* DualSimplex() */
```

A.17 parallel.c

```
/**
 * @file parallel.c
 * Variables and functions for multiprocessing
 */

#include <stdio.h>

#include "global.h"
#include "parallel.h"
#include "csmtree.h"
#include "set.h"

int myId = 0;
int numProcs = 1;
int rootNext;

#if MPI
#include "mpi.h"

int zIPowner;           // which process found current zIP?

//Now the stuff for creating an MPI struct:
int blockcount = 6;     // 6 ints in NodeType (see global.h)
MPI_Datatype type = MPI_INT;
MPI_Aint displacement = 0;
MPI_Datatype tsp_node;

MPI_Status status;      // used in all MPI code
int finalize_count = 0;     // count how many processes are finished

/** Register the NodeType struct with MPI and broadcast arrays
 * Since there is only one type in the struct, it is quite simple. We also
 * need to pass the arrays from the root node to the others so that all
 * processes have the same starting basis each time. Also necessary for
 * creating the array(s) of non-basic variables
 */
void parallel_arrays_init()
{
    MPI_Type_struct(1, &blockcount, &displacement, &type, &tsp_node);
    MPI_Type_commit(&tsp_node);

    MPI_Bcast(Node, NumNodes, tsp_node, 0, MPI_COMM_WORLD);
    MPI_Bcast(UArray, 1+NumEdges/8, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
    MPI_Bcast(UPair[0], NumNodes, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(UPair[1], NumNodes, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&USize, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

```
/** See if another process found better zIP
 * When a process finds a better zIP, it sends it to all other processes. So
 * we probe to see if there is a message, and update things as necessary.
 */
void update_zIP()
{
    int flag, min;
    //see if a message has been sent
    MPI_Iprobe(MPI_ANY_SOURCE, ZIP_TAG,
            MPI_COMM_WORLD, &flag, &status);
    while (flag) {
        MPI_Recv(&min, 1, MPI_INT, MPI_ANY_SOURCE,
            ZIP_TAG, MPI_COMM_WORLD, &status);
        if (min < zIP) {
            zIPowner = status.MPI_SOURCE;
            zIP = min;
            maxbranches = rc_reduce(zIP - zLP);
            if ((myId == 0) && (rootNext >= L_list_size)
                    && (rootNext < original_L_list_size))
                rootNext = original_L_list_size;
        }
        //test again
        MPI_Iprobe(MPI_ANY_SOURCE, ZIP_TAG,
                MPI_COMM_WORLD, &flag, &status);
    }
}

/** Hand out jobs from process 0. Also be carefull to jump over hole between
 * L and U and update finalize_count if applicable.
 */
void send_jobs()
{
    int flag, other_maxbranches;
    MPI_Iprobe(MPI_ANY_SOURCE, JOB_TAG,
            MPI_COMM_WORLD, &flag, &status);
    while (flag) {
        MPI_Recv(&other_maxbranches, 1, MPI_INT, MPI_ANY_SOURCE,
            JOB_TAG, MPI_COMM_WORLD, &status);
        MPI_Send(&rootNext, 1, MPI_INT, status.MPI_SOURCE,
                JOB_TAG, MPI_COMM_WORLD);
#if __debug_MPI
        printf("Just sent out %d, maxbranches=%d at node 0\n",
                rootNext, maxbranches);
        printf("got maxbranches=%d sending out rootNext=%d\n",
                other_maxbranches, rootNext);
#endif
        if (rootNext >= other_maxbranches) {
            finalize_count++;
#if __debug_MPI
```

```
            DEBUG("finalize_count=");
            printf("%d\n", finalize_count);
#endif
        }
        rootNext++;
        if ((rootNext >= L_list_size) &&
                (rootNext < original_L_list_size))
            rootNext = original_L_list_size;
        //rootNext will increment 1 by 1, but L_list_size might
        //become much less suddenly

        //test again
        MPI_Iprobe(MPI_ANY_SOURCE, JOB_TAG,
                MPI_COMM_WORLD, &flag, &status);
    }
}

/** setRootNext() for parallel environment */
void parallel_setRootNext()
{
#if __debug_MPI
    printf("id=%d, setRootNext()\n", myId);
#endif
    if (myId > 0) {
        /* send idle indicator to processor 0 */
        MPI_Send(&maxbranches, 1, MPI_INT, 0, JOB_TAG, MPI_COMM_WORLD);
        MPI_Recv(&rootNext, 1, MPI_INT, 0,
                JOB_TAG, MPI_COMM_WORLD, &status);
#if __debug_MPI
        printf("myId=%d, received tree %d, maxbranches=%d\n",
                myId, rootNext, maxbranches);
#endif
        if (rootNext == -1) rootNext = maxbranches;
    } else {
        send_jobs();
        //rootNext++;
    }
}

/** Wait for all processes to finish */
void parallel_finish()
{
    int tmp;
#if __debug_MPI & VERBOSE
    printf("myId=%d finished csm()\n", myId);
#endif
    rootNext = maxbranches;
    if (myId == 0) {
        finalize_count++;
        rootNext = -1;  //to have definite indication of termination
```

```
#if __debug_MPI & VERBOSE
        printf("root has finalize_count=%d\n", finalize_count);
#endif
        while (finalize_count < numProcs) {
            MPI_Recv(&tmp, 1, MPI_INT, MPI_ANY_SOURCE,
                    JOB_TAG, MPI_COMM_WORLD, &status);
            MPI_Send(&rootNext, 1, MPI_INT, status.MPI_SOURCE,
                    JOB_TAG, MPI_COMM_WORLD);
            finalize_count++;
        }
        update_zIP();   //other processes could find better zIP after
                //root node has finished csm().
                //TODO:need to get actual solution data from
                //node for solutionprint()XXX
                //do in solutionprint()
    }
    MPI_Barrier(MPI_COMM_WORLD);
}

#endif  //MPI
```

A.18 csmtree.c

```
/**
 * @file csmtree.c
 * This is the implementation of the methods that deal with the csm search tree
 */

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>      //For errno
#include <string.h>     //For strerror()

#include "global.h"
#include "csmtree.h"
#include "simplex.h"
#include "set.h"
#include "parallel.h"

#if MPI
#include <mpi.h>
#endif

struct pair *L_list, *U_list;   //Lists of non-basic variables at their lower &
                //upper bounds respectively
int L_list_size, U_list_size;   //the sizes of the lists (number of elements)
int original_L_list_size;   //the size of the original L_list
                //1/0 indicate fixed / not fixed  by csm

nodeptr nodelist = NULL;

long long total_created = 0;
long long created    = 0;
int depth       = 0;
int maxdepth        = 0;
int treesize        = 0;
int maxtreesize     = 0;

//int rcsize;
int maxbranches;

long long fathom_solution    = 0;
long long fathom_early_solution = 0;
long long fathom_dual        = 0;
long long fathom_marked = 0;
long long fathom_inconsistent    = 0;
long long fathom_nocolumns   = 0;
long long fathom_onezerorows     = 0;
long long fathom_bounded     = 0;
long long fathom_depth       = 0;
long long fathom_branches    = 0;

//#if __debug_csmtree
void rc_print()
{
    int i;
    for (i = 0; i < L_list_size; i++) {
        printf("L_list[%d] .column=%d .redcost=%d\n",
                i, L_list[i].column, L_list[i].redcost);
    }
    for (i = 0; i < U_list_size; i++) {
        printf("U_list[%d] .column=%d .redcost=(-)%d\n",
                i, U_list[i].column, U_list[i].redcost);
    }
}
//#endif

nodeptr newnode(nodeptr parent, int next, int column)
{
#if __debug_csmtree
    printf("newnode(parent:%d, next:%d, fixed:%d)\n",
            parent->fixed, next, column);
#endif
    /* add node as left child of *parent and fix column */
    nodeptr child = getnode();
    child->parent = parent;
    child->child = NULL;
    child->sibling = parent->child;
    child->next = next;
    child->sum = parent->sum;
    child->fixed = column;
    if (!InU(column)) child->sum += cost[column];
    parent->child = child;

    if ((total_created + created) % 1000000 == 0) {
        printf("added node %d on level %d below %d\n",
                column, depth + 1, parent->fixed);
        if (fflush(stdout)) perror(strerror(errno));
    }
    return child;
}

nodeptr getnode()
{
    nodeptr temp;
    created++;
    treesize++;
    if (treesize > maxtreesize) maxtreesize = treesize;
    if (nodelist == NULL) {
        temp = (nodeptr) calloc(1, sizeof(struct node));
        temp->flows = (char *) malloc(NumNodes * sizeof(char));
#if __debug_csmtree
        printf("allocated new node\n");
#endif
    } else {
        temp = nodelist;
        nodelist = nodelist->child;
#if __debug_csmtree
        printf("reused old node\n");
#endif
    }
    return temp;
}

void putnode(nodeptr ptr)
{
    if (ptr->parent) {
        ptr->parent->child = ptr->sibling;
    }
    treesize--;
    ptr->child = nodelist;
    nodelist = ptr;
}

void nodes_free(nodeptr tmp_root)
{
    free(tmp_root->flows);
    free(tmp_root);
    while (nodelist != NULL) {
        tmp_root = nodelist->child;
        free(nodelist->flows);
        free(nodelist);
        nodelist = tmp_root;
    }
}

static void selectsort(struct pair * rc, int first, int last)
{
    int i, j, min_pos;
    struct pair temp;
    int min;
    for (i = first; i < last; i++) {
        min = rc[i].redcost;
        min_pos = i;
        for (j = i + 1; j <= last; j++)
            if (rc[j].redcost < min) {
                min = rc[j].redcost;
                min_pos = j;
            }
        if (min_pos > i) {
            temp = rc[i];
            rc[i] = rc[min_pos];
            rc[min_pos] = temp;
        }
    }
}

static void quicksort(struct pair * rc, int first, int last)
{
    int left, right;
    struct pair temp;
    int key;
    if (last - first >= 4) {
        key = rc[first].redcost;
        left = first - 1;
        right = last + 1;
        while (1) {
            do right--; while (rc[right].redcost > key);
            do left++; while (rc[left].redcost < key);
            if (left >= right) break;
            temp = rc[left];
            rc[left] = rc[right];
            rc[right] = temp;
        }
        quicksort(rc, first, left - 1);
        quicksort(rc, right + 1, last);
    } else
        selectsort(rc, first,last);
}

/** Initialises L_list and U_list and calls rc_reduce() */
int rc_init(int gap)
{
    int i;
    int node_a, node_b;          /* two temporary nodes */
    int L_counter = 0, U_counter = 0;    /* current size of each list */
    U_list_size = USize;
    L_list_size = NumEdges - NumNodes - USize;
    original_L_list_size = L_list_size;
    L_list = (struct pair *) malloc(L_list_size * sizeof(struct pair));
    U_list = (struct pair *) malloc(U_list_size * sizeof(struct pair));

    node_a = 0;
    node_b = 0;
    for (i = 0; i < NumEdges; i++) {
        node_b++;
        if (node_b == NumNodes) {
            node_a++;
            node_b = node_a + 1;
        }
        if ((Node[node_a].Parent == node_b) ||
            (Node[node_b].Parent == node_a)) {
            continue;   /* basic */
        }
        if (InU(i)) {
            U_list[U_counter].column = i;
            U_list[U_counter].redcost = cost[i]
                    - Node[node_a].Dual - Node[node_b].Dual;
            /* We invert the sign. This enables us to use the
             * same sorting function for both arrays:
             */
                U_list[U_counter].redcost *= -1;
            U_counter++;
            continue;
        }
        /* i is in L */
        L_list[L_counter].column = i;
        L_list[L_counter].redcost = cost[i]
                    - Node[node_a].Dual - Node[node_b].Dual;
        L_counter++;
    }
    quicksort(L_list, 0, L_list_size - 1);
    quicksort(U_list, 0, U_list_size - 1);
#if __debug_csmtree
    if (myId == 0) {
        rc_print();
        printf("%d variables in L\n", L_list_size);
        printf("%d variables in U\n", U_list_size);
        printf("Now reducing rc with gap=%d\n", gap/2);
    }
#endif
    return (rc_reduce(gap));
}

/**
 * Reduces both L_list and U_list according to the gap specified
 * The gap should always be given as (zIP - zLP). NOT zIP - zLP - ONE. The
 * reduction done here only allows columns with reduced costs LESS than the
 * specified gap.
 */
int rc_reduce(int gap)
{
    while (L_list_size > 0) {   /* L_list_size : number of elements */
        L_list_size--;
        if (L_list[L_list_size].redcost < gap) {
            L_list_size++;
            break;
        } else {
            mark[L_list[L_list_size].column] = 2;
        }
    }
    //gap *= -1;
    while (U_list_size > 0) {   /* U_list_size : number of elements */
        U_list_size--;
        if (U_list[U_list_size].redcost < gap) {
            U_list_size++;
            break;
        } else {
            mark[U_list[U_list_size].column] = 2;
        }
    }
#if __debug_csmtree
    if (myId == 0) {
        rc_print();
        printf("Reduced rc with gap=%d\n", gap/2);
    }
#endif
    if (myId == 0) printf("rcsize now %d\n", U_list_size + L_list_size);
    return (original_L_list_size + U_list_size);
}

void rc_free()
{
    free(L_list);
    free(U_list);
}

void fathom_statistics()
{
#if MPI
    long long int tmp;
    MPI_Reduce(&fathom_marked, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_marked = tmp;
    MPI_Reduce(&fathom_inconsistent, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_inconsistent = tmp;
    MPI_Reduce(&fathom_depth, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_depth = tmp;
    MPI_Reduce(&fathom_branches, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_branches = tmp;
    MPI_Reduce(&fathom_bounded, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_bounded = tmp;
    MPI_Reduce(&fathom_onezerorows, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_onezerorows = tmp;
    MPI_Reduce(&fathom_nocolumns, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_nocolumns = tmp;
    MPI_Reduce(&fathom_solution, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_solution = tmp;
    MPI_Reduce(&fathom_early_solution, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_early_solution = tmp;
    MPI_Reduce(&fathom_dual, &tmp, 1, MPI_LONG_LONG_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    fathom_dual = tmp;
    if (myId != 0) return;
#endif
    printf("--------------------\n");
    printf("Fathoming statistics\n");
    printf("--------------------\n");
    printf("marked\t\t%lld\n", fathom_marked);
    printf("inconsistent\t%lld\n", fathom_inconsistent);
    printf("depth\t\t%lld\n", fathom_depth);
    printf("branches\t%lld\n", fathom_branches);
    printf("bounded\t\t%lld\n", fathom_bounded);
    printf("onerow/zerorow\t%lld\n", fathom_onezerorows);
    printf("no columns\t%lld\n", fathom_nocolumns);
    printf("solution\t%lld\n", fathom_solution);
    printf("early solution\t%lld\n", fathom_early_solution);
    printf("dual\t\t%lld\n", fathom_dual);

    printf("\ntotal fathoms\t%lld\n", fathom_marked +
        fathom_inconsistent +
        fathom_depth +
        fathom_branches +
        fathom_bounded +
        fathom_onezerorows +
        fathom_nocolumns +
        fathom_solution+
        fathom_dual);
    printf("not accounted\t%lld\n", total_created - fathom_marked -
        fathom_inconsistent -
        fathom_depth -
        fathom_branches -
        fathom_bounded -
        fathom_onezerorows -
        fathom_nocolumns -
        fathom_solution -
        fathom_dual);
}
```

A.19 istour.c

```
/**
 * @file istour.c
 * used to determine if a solution is a valid tour
 * We use the Union-Find algorithm with path compression and weight balancing
 */

#include <stdlib.h>
#include <stdio.h>

#include "global.h"
#include "istour.h"
#include "simplex.h"
#include "set.h"

static int * parent;
/**<
 * parent represents a forest of trees
 * if parent[i] is non-negative, it is the parent of i
 * if parent[i] is negative, i is a root of a tree with -parent[i] nodes
 * numbered from 0 to n-1
 */

#if __debug_istour
void istour_print()
{
    int i;
    for (i = 0; i < NumNodes; i++)
        printf("parent[%d] = %d\n", i, parent[i]);
}
#endif

void istour_init()
{
    parent = (int *) malloc(NumNodes * sizeof(int));
}

void istour_free()
{
    free(parent);
}

/**
 * findroot returns the root of the given node and compresses the whole
 * path from x to its root
 */
static int findroot(int x)
{
    int xroot, temp;
    xroot = x;
    while (parent[xroot] >= 0) xroot = parent[xroot];
    while (parent[x] >= 0) {
        temp = x;
        x = parent[x];
        parent[temp] = xroot;
    }
    return xroot;
}

/**
 * weighted_union puts i and j in the same tree
 */
static int weighted_union(int i, int j)
{
    i = findroot(i);
    j = findroot(j);
    if (i == j) {         /* already part of the same tree */
        return (-parent[i] == NumNodes);
    }
    if (parent[i] > parent[j]) {
        parent[j] += parent[i];
        parent[i] = j;
        return (parent[j] == -NumNodes);
    } else { /* parent[j] <= parent[i] */
        parent[i] += parent[j];
        parent[j] = i;
        return (parent[i] == -NumNodes);
    }

}

/** Determine if current basis structure with fixed columns define a tour.
 * When called istour() assumes primal and dual feasibility, but does check
 * integrality.
 */
int istour(nodeptr node)
{
    int i,j;                  /* for for loops */
    int a, b;                 /* temp nodes */
    int return_value = 0;
    /* Initialise the whole array to -1
     * in other words, each node is the root of a tree with 1 node
     */
    for (i = 0; i < NumNodes; i++) parent[i] = -1;

    /* first we iterate through all the basic edges */
    for (i = 0; i < NumNodes; i++) {
        /* Now we look for all nodes with flow values of 1 */
        if (Node[i].Flow == HALF) {
            return 0;        /* not an integer solution */
        }
        if (Node[i].Flow == ONE) {
            j = Node[i].Parent;
            return_value = weighted_union(i, j);
        }
        /* Nodes with Flow == ZERO is simply skipped */
    }
    if (return_value) return 1;

    /* Now we insert all the fixed columns */
    while (node != NULL) {
        if (node->fixed == -1) break;    /* test for root */
        if (!InSet(node->fixed, UCopy)) {
            /* we'll handle variables in U later */
            give_nodes(node->fixed, &a, &b);
            return_value = weighted_union(a, b);
        }
        node = node->parent;
    }
    if (return_value) return 1;

    /* Now we do the edges in U */
    for (i = 0; i < USize; i++) {
        a = UPair[0][i];
        b = UPair[1][i];
        if ((-1 < mark[INDEX(a, b)]) && weighted_union(a, b))
            return 1;
    }

#if __debug_istour
    printf("istour() returns 0\n");
    istour_print();
#endif
    return 0;
}/* istour */
```

A.20 set.c

```
/**
 * @file set.c
 * Implementation of the functionality for handling the set U (non-basic variables
 * at the upper-bound)
 */

#include <stdlib.h>
#include <stdio.h>

#include "global.h"
#include "set.h"

#if __debug_U
#include <stdio.h>
#endif

unsigned char bits[8] = {1,2,4,8,16,32,64,128};
unsigned char *UArray;
int **UPair;
int USize;

void U_init()
{
    UArray = (unsigned char *) calloc(1+NumEdges/8, sizeof(unsigned char));
    UPair = (int **) malloc(2* sizeof(int *));
    UPair[0]  = (int *) malloc(NumNodes * sizeof(int));
    UPair[1]  = (int *) malloc(NumNodes * sizeof(int));
    USize = 0;
} /*InitU*/

void U_free()
{
    free(UArray);
    UArray = NULL;
    free(UPair[0]);
    free(UPair[1]);
    free(UPair);
    USize = 0;
}

/**
 * adds edge id between i and j to U
 */
void AddToU(int i, int j, int id)
{
    if (mark[id]) {
        fprintf(stderr, "  ** putting fixed column into U\n");
    }
    UArray[id/8] ^= bits[id%8];
    if (i < j) {
        UPair[0][USize] = i;
        UPair[1][USize] = j;
    } else {
        UPair[0][USize] = j;
        UPair[1][USize] = i;
    }
    USize++;
} /*AddToU*/

/** deletes edge id between i and j from U */
void DeleteFromU(int i, int j, int id)
{
    int Help;
    if (mark[id]) {
        fprintf(stderr, "  ** putting fixed column into U\n");
    }
    UArray[id/8] ^= bits[id%8];
    if (i > j) {
        Help = i;
        i = j;
        j = Help;
    }
    Help = 0;
    USize--;
    do {
        if (UPair[0][Help] == i && UPair[1][Help] == j) {
            if (Help != USize) {
                UPair[0][Help] = UPair[0][USize];
                UPair[1][Help] = UPair[1][USize];
            }
            Help = -1;
        } else Help++;
    } while (Help != -1);
} /*DeleteFromU*/
```

Bibliography

[1] Amdahl, G. M. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings* (1967), vol. 30, pp. 483–485.

[2] Applegate, D., Bixby, R., Chvátal, V., and Cook, W. Concorde tsp solver. http://www.tsp.gatech.edu/concorde.html.

[3] Applegate, D., Bixby, R., Chvátal, V., and Cook, W. Milestones in the solution of tsp instances. http://www.tsp.gatech.edu/histmain.html.

[4] Applegate, D., Bixby, R., Chvátal, V., Cook, W., and Helsgaun, K. Sweden 24,978. http://www.tsp.gatech.edu/sweden/index.html.

[5] Baase, S. *Computer Algorithms - Introduction to Design and Analysis*, second ed. Addison-Wesley Publishing Company, Inc, 1988.

[6] Cannon, T. L., and Hoffman, K. L. Large-scale 0-1 linear programming on distributed workstations. *Annals of Operations Research 22* (1990), 181–217.

[7] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. MIT Press, 1990.

[8] Dantzig, G. B., Fulkerson, R., and Johnson, S. M. Solution of a large-scale traveling salesman problem. *Operations Research 2* (1954), 393–410.

[9] Geldenhuys, C. E. *An implementation of a Branch-and-Cut algorithm for the travelling salesman problem.* Master's dissertation, RAU, May 1998.

[10] Glover, F., Klingman, D., and Stutz, J. Extensions of the augmented predecessor index method to generalized network problems. *Transportation Science 7* (1974), 377–384.

[11] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI - Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. MIT Press, 1999.

[12] Hajian, M. T., Hai, I., and Mitra, G. A distributed processing algorithm for solving integer programs using a cluster of workstations. *Parallel Computing 23* (June 1997), 733 – 753. Issue 6.

[13] Harche, F., and Thompson, G. L. The column subtraction algorithm: an exact method for solving weighted set covering, packing and partitioning problems. *Computers and Operations Research 21*, 6 (1994), 689–705.

[14] Held, M., and Karp, R. M. The traveling-salesman problem and minimum spanning trees. *Operations Research 18* (1970), 1138–1162.

[15] Hoffman, K. L. Combinatorial optimization: Current successes and directions for the future. *Journal of Computational and Applied Mathematics 124* (December 2000), 341–360.

[16] Hoffman, K. L., and Padberg, M. Solving airline crew scheduling problems by branch-and-cut. *Management Science 39* (1993), 657–682.

[17] Hoffman, K. L., and Padberg, M. Set-covering, packing and partitioning problems. In *Encyclopedia of Optimization*, C. Floudas and P. Pardalos, Eds., vol. 5. Kluwer Academic Publishers, 2001, pp. 174–178.

[18] Kennington, J., and Helgason, R. *Algorithms for network programming.* John Wiley and Sons, New York, 1980.

[19] Lai, T.-H., and Sahni, S. Anomalies in parallel b&b algorithms. *Research Contributions 27*, 6 (1984), 594 – 602.

[20] Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., and Shmoys, D. B., Eds. *The Traveling Salesman Problem - A Guided Tour of Combinatorial Optimization.* John Wiley and Sons, New York, 1986.

[21] Leenen, L. *Contributions towards an implementation of a Branch-and-Cut algorithm for the travelling salesman problem.* Master's dissertation, RAU, September 1992.

[22] Meyer, T. W. S. *Die implementering en eksperimentele evaluering van enkele diskrete optimeringsalgoritmes.* Master's dissertation, RAU, May 1986.

[23] Padberg, M., and Rinaldi, G. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review 33*, 1 (March 1991), 60–100.

[24] Reinelt, G. Tsplib home page. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

[25] Reinelt, G. Tsplib: Questions and answers. http://www.informatik.uni-heidelberg.de/groups/comopt/software/TSPLIB95/TSPFAQ.html.

[26] Reinelt, G. Tsplib95. http://www.informatik.uni-heidelberg.de/groups/comopt/software/TSPLIB95/DOC.PS.

[27] Sedgewick, R. *Algorithms*, second ed. Addison-Wesley Publishing Company, Inc, 1988.

[28] Smith, T. H. C., Meyer, T. W. S., and Leenen, L. An efficient primal simplex implementation for the continuous 2-matching problem. *South African Computer Journal 5* (1991), 28–31.

[29] Smith, T. H. C., and Thompson, G. L. A parallel implementation of the column subtraction algorithm. *Parallel Computing 21* (1995), 63–74.

[30] Smith, T. H. C., and Thompson, G. L. An improved column subtraction method for the set partitioning problem. Working paper, August 2000.

[31] Winston, W. L. *Operations Research : Applications and algorithms*, third ed. International Thomson Publishing, 1994.

Index