University of Groningen

Computing Science

Task Scheduling Feasibility with Linear Programming

Author:
Wytze Hazenberg
student nr. 1810375

Supervisors:
A. Lazovik
A. Telea

August 14, 2014

Abstract

Assume you need to execute multiple tasks and you have some number of computer nodes available to execute a subset of these tasks. Each task requires a certain amount of capacity and each node has certain capacity to facilitate the execution of tasks. To determine which task has to be executed on which node, a scheduler has to take the various variables into account while producing an optimal scheduling topology.

Linear programming is a mathematical tool to optimize the solution to a problem. Given linear constraints, linear programming maximizes, or minimizes, a single linear objective function. Linear programming obtains the optimal solution, if any, without violating any constraints.

In this thesis, linear programming and methods to solve linear programming problems are discussed and a linear programming model is proposed to schedule tasks to nodes while respecting variable requirements. The proposed model is evaluated with several test configurations to determine whether linear programming is feasible for task scheduling.

Table of Contents

1 Introduction **4**

2 Problem Statement **5**
- 2.1 Node . . . 6
- 2.2 Task . . . 6
- 2.3 Topology . . . 7
- 2.4 Scheduler . . . 7
- 2.5 Variations . . . 8
  - 2.5.1 Task dependencies . . . 8
  - 2.5.2 Point in time . . . 8
  - 2.5.3 Duration . . . 8
  - 2.5.4 Task priority . . . 9
  - 2.5.5 Tags . . . 9
  - 2.5.6 Node's resource protection . . . 9
  - 2.5.7 Custom objective function . . . 9

3 Related work **10**
- 3.1 Dynamic programming / Knapsack . . . 10
- 3.2 Branch and Bound . . . 14
- 3.3 Constraint Programming . . . 17
- 3.4 Local Search . . . 21
- 3.5 Conclusion . . . 25

4 Linear Programming **26**
- 4.1 Diet Problem . . . 26
- 4.2 Standard Form . . . 29
- 4.3 Standard Form conversion . . . 30
- 4.4 Simplex . . . 32
  - 4.4.1 Slack variables . . . 32
  - 4.4.2 Dictionary . . . 33
  - 4.4.3 Algorithm . . . 34
  - 4.4.4 Initialization phase . . . 37
  - 4.4.5 Entering and leaving variable heuristics . . . 39
  - 4.4.6 Degeneracy, cycling and stalling . . . 40
  - 4.4.7 Complexity . . . 41
- 4.5 Duality . . . 42
- 4.6 Integer Linear Programming . . . 43
  - 4.6.1 Branch and Bound . . . 45
  - 4.6.2 Cutting Planes . . . 48
  - 4.6.3 Branch and Cut . . . 49

5 Task Scheduling with Linear Programming **50**
5.1 Overview . . . 50
5.2 Obtaining task requirements . . . 52
5.3 Obtaining Node Variables . . . 53
5.4 Linear Programming Model . . . 53
5.4.1 Example: 3 tasks, 2 nodes, 1 variable . . . 56
5.4.2 Example: 4 tasks, 2 nodes, 1 variable . . . 59
5.4.3 Example: 3 tasks, 2 nodes, 2 variables . . . 60
5.5 Dual Linear Programming Model . . . 62
5.5.1 Example: 3 tasks, 2 nodes, 1 variable . . . 65
5.5.2 Example: 4 tasks, 2 nodes, 1 variable . . . 66
5.5.3 Example: 3 tasks, 2 nodes, 2 variables . . . 67
5.6 Variations . . . 68
5.6.1 Task dependencies . . . 68
5.6.2 Point in time . . . 69
5.6.3 Duration . . . 72
5.6.4 Task priority . . . 72
5.6.5 Tags . . . 73
5.6.6 Node's resource protection . . . 75
5.6.7 Custom objective function . . . 75

6 Evaluation **77**
6.1 Test environment . . . 77
6.2 Test parameters . . . 77
6.3 Results LP relaxation . . . 78
6.4 Results Binary LP . . . 82

7 Discussion **87**

A Raw results **89**
A.1 LP relaxation . . . 89
A.2 ILP . . . 94

Glossary **102**

Acronyms **103**

Bibliography **104**

1 Introduction

A lot of data is generated each day by users and applications. Data needs to be stored and processed on some server, or node. The trend is that the amounts of generated data will continue to grow in the future.

Big data sets do not often fit on one physical node. Multiple commodity servers or cloud virtual machines are often used to scale data horizontally. NoSQL solutions offer horizontal scaling and partitioning of data sets to spread read and write operations.

To extract meaningful information from a data set, a data set often needs processing. Tasks are defined to solve a specific problem. One problem can be split up in multiple tasks which can be processed in parallel on one or more nodes. Hadoop [12] is an example of processing tasks in parallel. One or more tasks are executed on one or more specified nodes. Individual task results are merged together to yield the problem's solution.

In a distributed cloud environment, where multiple nodes are available to execute tasks, it is not that obvious how to schedule tasks for processing. How the tasks are scheduled impact the time it takes to obtain the solution. Some nodes have high workloads, many read and write requests, while other nodes have not. It is not always clear on which nodes to execute a specific task. Arbitrarily selecting a node to execute a task might result in an overscheduled node. In the end, the node consumes more time to process the task which leads to a slower yield of a task's solution. Given that each node has a certain available capacity, a discrete optimization technique can be deployed as part of a task scheduler to schedule tasks onto nodes for execution.

Linear programming [2, 11] is a mathematical tool to optimize the solution to a problem. Given linear problem constraints, linear programming maximizes, or minimizes, a single objective function. Linear programming obtains the optimal solution, if any, without violating any constraints. Linear programming is discussed in chapter 4.

The research question is whether linear programming is feasible to build into a distributed task scheduler. Linear programming would be feasible if the resulting task scheduler is efficient, fair, dynamic and transparent [7]. The problem statement is described in chapter 2. Sub questions of the main research question are:

- how to obtain task variable values (section 5.2);
- how to obtain node variable values (section 5.3);
- does the LP model accept variations, such as dependencies and tags? (section 5.6)

The main contribution consists of evaluating a linear programming model with various test configurations to assess feasibility and is discussed in chapter 6. As part of the main contribution, the proposed linear programming model for task scheduling is discussed in chapter 5 along with various variations.

2 Problem Statement

Nowadays, processing power becomes widely available due to the uprising of cloud service providers. Problems that need to be solved increase in complexity over time and solutions need to be found in an efficient manner. The cloud's processing power can be leveraged to solve problems whether the problems are solved as a whole or split up in chunks. Ideally, these processors, virtual machines or nodes, are fully utilized to concurrently solve as many problems as possible.

As stated in the introduction, the research question is whether linear programming is feasible to build into a distributed task scheduler. Linear programming is a feasible component of a task scheduler when the resulting task scheduler is efficient, fair, dynamic and transparent [7]. The efficiency of a task scheduler with linear programming is that is should improve the performance of scheduled jobs as much as possible and the process of scheduling should incur low overhead such that it does not counterattack the benefits. The fairness of the task scheduler is that each task receives a fair share of the shared resources available when demand is high. A dynamic task scheduler adapts to load changes and exploits the full extent of the resources available. A task scheduler's transparency is that it should make no difference on which node a task is executed, there should be no difference, and no user effort should be required in deciding where to execute a task or initiating remote execution; a user should not even be aware of remote processing.

The problem that linear programming needs to solve is how to allocate, or schedule, tasks to nodes, preferably optimal. An algorithm is required to act as part of a task scheduler where characteristics of tasks and nodes are inputs, and a topology is the output. The inputs are to be collected by measurement or otherwise. The linear programming problem should maximize the amount of tasks scheduled. The next sections define each term in more detail.

A scheduling, or topology, should be feasible and optimal. A feasible scheduling is a scheduling where one or more tasks are scheduled to nodes without violating any constraints, however, there exists another feasible scheduling in where more tasks are scheduled. The optimal scheduling is a scheduling where no more tasks can be scheduled without violating any capacity constraints. As discussed in chapter 4, linear programming obtains a solution or determines the problem is not feasible. When linear programming determines that a problem is feasible; each yielded solution is feasible, after a pivoting step, and the last yielded solution is guaranteed to be optimal.

2.1 Node

A node is the reference to a physical server or a cloud instance that executes an operating system and user processes, denoted as $Node_j$ where j is the unique reference, or identifier, to that node. A node has a certain capacity of processing power, or resources. Among others, a node's resources are internal memory capacity and hard disk space. A node's resource is generalized and referred to as a variable. Since a node is under constant load, the node's available resources fluctuate over time. The sampled state of the node's resources at some point in time t is defined as:

$$Node_j^t = \begin{pmatrix} V^t \\ C^t \end{pmatrix} \tag{2.1}$$

Where V^t is a vector of values which represent the current measured state of the node for each variable v at point in time t. Vector C represents the node's total capacity for each variable v at point in time t. The vectors V and C can be used to determine a node's current utilization.

Assume there is a node j that has 4096MB of internal memory from which 2048MB is currently in use. The corresponding value in the V vector for variable v at point in time t, i.e. V_v^t, is 2048. Similarly, $C_v^t = 4096$. The available capacity for variable v can be calculated by subtraction:

$$\begin{aligned} & C_v^t - V_v^t \\ = \\ & 4096 - 2048 \\ = \\ & 2048 \end{aligned} \tag{2.2}$$

2.2 Task

A task is a user-provided solution to a given problem. A task consists of one or more commands that can be executed on a node. Once a task is scheduled to a node, the node executes the task's commands to solve the user's problem. The i-th task is identified as $task_i$. For task i to be scheduled to node j, node j has to have a minimal number of resources available to be able to execute task i's commands.

For example, a task has to load a data set of 100MB into memory for processing. The task requires a node with at least 100MB of available memory to be able to execute the task. The task cannot be scheduled when no node is found that satisfies this constraint.

A task is defined as:

$$task_i = \begin{bmatrix} r_1 \\ \vdots \\ r_v \\ \vdots \\ r_k \end{bmatrix} \tag{2.3}$$

Where r_v is the node's minimal required capacity to execute task i for variable v. Task i can only be executed on node j if and only if node j can satisfy the capacity for all k variables, i.e. $\forall v \in \{1, \ldots, k\} : (C_v^t - V_v^t) - r_v \geq 0$ has to be satisfied. The variable values have to be comparable and the measurement unit has to be uniform per variable between nodes and tasks.

2.3 Topology

A topology is a logical ordering, or schedule, of which task is assigned to which node for execution. After a topology has been yielded, tasks can be distributed among the nodes for execution. Every node has to respect the scheduling for execution.

The topology is defined as:

$$T^t = \begin{pmatrix} task_1 & node_1 & \delta(1,1) \\ task_1 & node_2 & \delta(1,2) \\ & \vdots & \\ task_1 & node_m & \delta(1,m) \\ & \vdots & \\ task_i & node_j & \delta(i,j) \\ & \vdots & \\ task_n & node_m & \delta(n,m) \end{pmatrix} \qquad \delta(i,j) = \begin{cases} 0 & \text{if task } i \text{ is not scheduled to node } j \\ 1 & \text{if task } i \text{ is scheduled to node } j \end{cases} \tag{2.4}$$

Where the value of the δ-function determines whether task i is scheduled to node j. Topology T is produced by the scheduler at some point in time t due to fluctuating node variables. It is not guaranteed that node j has to execute at least a single task at point in time t and, also, it is not guaranteed that task i is scheduled at point in time t.

2.4 Scheduler

The scheduler function takes nodes and tasks as input and has a topology T as output. Based on the variables of each task and each node, an optimal schedule is obtained by examining different schedules and come up with the optimal schedule while respecting task and node constraints. An optimal topology is considered optimal if each node's capacity is fully utilized while maximizing the number of available tasks scheduled. In other words, tasks are scheduled to available nodes, which are capacity limited, in a way that the number of tasks scheduled is maximized. In other words, finding an ordering of tasks scheduled to nodes such that no better ordering can be found based on its inputs and the number of tasks scheduled is

maximized. A valid, or feasible, schedule is a suboptimal schedule where one or more tasks are scheduled on the available nodes but there exists another possible schedule that schedules more tasks.

The scheduler function is defined as:

$$T^t = scheduler(Tasks, Nodes) \tag{2.5}$$

Where

$$Tasks = \begin{pmatrix} Task_1 \\ \vdots \\ Task_n \end{pmatrix} Nodes = \begin{pmatrix} Node_1 \\ \vdots \\ Node_m \end{pmatrix} \tag{2.6}$$

Not all available tasks have to be scheduled at the same time because there is not enough node capacity available. This is why the problem is an optimization problem; there is a greater request for resources than there is capacity available.

2.5 Variations

In practice, a basic scheduling of tasks to nodes is not always the case. The following subsections describe possible variations to the scheduling process.

2.5.1 Task dependencies

It is not uncommon to have dependencies between tasks. Some tasks can, or have to, be executed in parallel while other tasks have to be executed in serial order.

For example, aggregations are to be executed in parallel to produce intermediate results. In turn, intermediate results are processed by another set of parallel tasks to obtain the final result. Tasks that produce intermediate results and task that produce the end result have to be executed in serial order.

2.5.2 Point in time

There are tasks that cannot be executed before a certain point in time. Time constrained tasks are prepared and made available to the scheduler in advance by the task administrator. Execution has to be postponed until the specific point in time is reached. For instance, backup tasks usually are executed during the night.

Generally, it is possible to think that every task has its specific point in time of execution; Tasks that do not have specified a specific point in time are scheduled as soon as possible, depending on available node resources.

2.5.3 Duration

It is possible for tasks to be services. Services run over a period of time and require capacity in order to function properly and continuously. Normally, tasks are executed and capacity frees up over time. However, services require capacity depending on activity, i.e. more capacity is required when there is a lot of activity in comparison to an idle service. When it is uncertain when capacity is required, a choice might be to allocate sufficient capacity for all service scenarios over a certain period of time.

2.5.4 Task priority

Execution of some tasks is preferred over execution of other tasks, i.e. some tasks have priority over others. Higher priority tasks have to be considered before considering lower priority tasks. For example, backup tasks generally have less priority than tasks that perform business critical tasks.

2.5.5 Tags

Tags are useful to group tasks and/or nodes together. Nodes receive a user-defined tag. If a task requires a certain tag, that task gets scheduled on one or more nodes matching the same tag. It is possible for nodes and tasks to have multiple assigned tags.

For example, tags are useful for datacenter awareness, to partition nodes geographically. For instance, a company has data centers in the US and in Europe. Some tasks have to be executed in the US while other tasks are to be executed in Europe. Assigning tags to tasks and nodes separates them from each other and prevents execution of a task in the wrong data center.

Defining a unique tag to each node enables to schedule a task to a single specific node, or multiple specific nodes. For example, executing a node specific backup task or to test a newly created task on a specific test node.

2.5.6 Node's resource protection

Setting a limit to a node's resources prevents a node from reaching a critical workload. It is good practice to not utilize a full 100% of the free resources available. The node's operating system needs resources as well. The remaining resources can form a buffer for the operating system and tasks that have poorly obtained requirements. Another reason is to prevent under-performing nodes when a node's cpu or memory is fully used. Fully utilizing memory might cause memory to swap to hard disk and fully utilizing cpu causes applications to wait on available cpu time and slow down significantly.

2.5.7 Custom objective function

The current objective of de linear programming problem is to maximize the number of scheduled tasks while respecting the capacity constraints. However, as variation, it is not always straightforward to maximize a single value, i.e. maximizing the number of tasks scheduled. A second value may be important in some use cases.

Assume the use case of having a certain cost to schedule a task to a particular node. An optimal solution is the solution where the most tasks get scheduled. However, the cost of scheduling should be minimized. So, the objective function of the linear programming problem should be to maximize the number of scheduled tasks while, at the same time, minimize the cost of scheduling.

3 Related work

Task scheduling is closely related to the scheduling, or assignment, problem in the linear programming's field of research [16]. Typical assignment problems include employee - job assignment, department resource allocation [5], the traveling sales man problem [15] and many other problems have similarities.

Linear programming originates from around the 1940s. The simplex method, the first method to solve linear programming problems, was published in 1947. Other methods for solving linear programming problems were developed in 1955, the Hungarian method by H. Kuhn [6], and the interior-point method in 1984, by N. Karmarkar [11, 15, 16]. The research in the linear programming field is still ongoing to solve linear programming problems more efficiently. However, it is difficult to create a solver that performs equally well on a large class of problems [11].

Scheduling is a frequently studied problem in general. Scheduling solutions have been implemented all around the computing science field, whether it is used in CPU scheduling, I/O scheduling or distributed scheduling in systems like Hadoop [4]. Schedulers are concerned with efficiency and fairness [7], improvements are constantly researched based on new findings and technology. For example; with the uprising of SSD technology, as of Linux's 3.13 kernel, Ubuntu 14.04's default I/O scheduler changed from CFQ to Deadline to gain performance benefits [13].

Discrete optimization algorithms dynamic programming, branch and bound, constraints programming and local search are discussed in the following sections to provide insight in the field of research.

3.1 Dynamic programming / Knapsack

The knapsack problem [15] is a problem where a set of items have to be fitted into a knapsack. Each item has a certain value and a certain weight. The knapsack has a certain capacity and items are placed inside the knapsack. Once an item is placed inside the knapsack, the item's weight is subtracted from the remaining capacity. The capacity of the knapsack cannot be exceeded. An allocation of items has to be found such that the combined value of the placed items is maximized.

The definition of the knapsack problem is:

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in I} v_i x_i \\ \text{subject to (s.t.)} \quad & \sum_{i \in I} w_i x_i \leq K \\ & x_i \in \{0, 1\} \quad (i \in I) \end{aligned} \tag{3.1}$$

Where v_i is the value of item i, w_i is the weight of item i and K is the knapsack's capacity. The set I contains all possible items to fit into the knapsack. The objective is to maximize the value in the knapsack while respecting the capacity constraint of all added weights in the knapsack.

It is possible to solve the knapsack problem with different optimization techniques. Dynamic programming and branch and bound are two possible techniques to solve

the knapsack problem. The following problem [15] illustrates solving the knapsack problem using dynamic programming:

$$\begin{aligned} \text{maximize } & 5x_1 + 6x_2 + 3x_3 \\ \text{s.t. } & 4x_1 + 5x_2 + 2x_3 \leq 9 \\ & x_1, x_2, x_3 \in \{0, 1\} \end{aligned} \tag{3.2}$$

Dynamic programming sequentially adds items to the knapsack at each step and finds the best possible combination out of the subset of items formed by maximizing the sum of values. The last step evaluates the last possible item. Each step uses values from the previous step to determine whether or not an item should be included in the knapsack. The process is best illustrated by constructing a table. Algorithm 1 demonstrates how to construct a dynamic programming table in pseudocode.

Algorithm 1: construction of a dynamic programming table

input : set of items I and total capacity K
output: table T

```
1  let table T be a matrix of zeros with dimensions capacity x items.
2  foreach item i ∈ I do
3      value ← value of item i;
4      weight ← weight of item i;
5      prev_item ← predecessor of item i;
6      for every capacity in K do
7          if capacity < weight then // the minimum capacity is not reached, yet
8              T[capacity][item] ← T[capacity][prev_item];
9          else
10             T[capacity][item] ← Max( T[capacity][prev_item],
                                        T[capacity - weight][prev_item] + value);
                   /* deciding to pick the item or not, to optimize value */
11         end
12     end
13 end
```

The resulting table of example (3.2) is shown in table 3.1. The first item has a weight of 4 and a value of 5. The value 5 is added to the table on the 4-th capacity row and onwards. Item 2 has weight 5 and value 6. The best value at a capacity of 4 is value 5, from the first item, because the second item's weight requirement is not met. Item 2 is considered on capacity row 5. There is a choice to be made on capacity row 5; selecting item 1 would yield an objective value of 5 while selecting item 2 yields an objective value of 6. On capacity row 9, both item 1 and item 2 fit. The value of item 1 and item 2 sums up to an objective value of 11. This process continues until all items are processed. The optimal objective value is the value at the bottom right, values summing up to 11 for the chosen items.

capacity	item			
	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	3
3	0	0	0	3
4	0	5	5	5
5	0	5	6	6
6	0	5	6	8
7	0	5	6	8
8	0	5	6	8
9	0	5	11	11
		$v_1 = 5$	$v_2 = 6$	$v_3 = 3$
		$w_1 = 4$	$w_2 = 5$	$w_2 = 2$

Table 3.1: Table of dynamic programming example (3.2)

At this point the optimal objective value is known. However, to obtain information on which item is chosen to put into the knapsack, the table needs some reverse traversal. The starting point is at the right-most bottom value, the optimal objective value, on capacity row K. If the value in the column to the left, on the same row, has the same value then the current item has not been chosen to go into the knapsack. If the value differs from the value to the left, the current item has been chosen. Subtracting the weight of the chosen item from the residual capacity, initially K on the K-th row, to yield the capacity row on which to continue the process from. In example 3.2, item 3 has not been chosen because values of item 2 and item 3 on row $K = 9$ are equal. Item 2 has been chosen because values of item 1 and item 2 differ on row K. Item 2's weight is 5, the residual capacity is $K - 5 = 4$, inspection continues from capacity row 4. Item 1 has been chosen because the value of item 0 and item 1 differ on row 4. The resulting decision variable values are:

$$\begin{aligned} x_1 &= 1 \\ x_2 &= 1 \\ x_3 &= 0 \end{aligned} \tag{3.3}$$

In contrast, dynamic programming can be used to schedule tasks. Assume that a task has only one variable. The weight of a task is set to the required variable's value. The value could either be one, a user defined value, or set equally to the weight. The knapsack problem represents a single node with a single variable. Capacity K is the amount of free capacity that the node has for the variable. The knapsack problem formulation of the example illustrated in subsection 5.4.1 is as follows:

$$\begin{aligned} \text{maximize } & 200x_1 + 500x_2 + 300x_3 \\ \text{s.t. } & 200x_1 + 500x_2 + 300x_3 \leq 500 \\ & x_1, x_2, x_3 \in \{0, 1\} \end{aligned} \tag{3.4}$$

The knapsack represents the first node from the example with the capacity $K = 500$. Solving this problem, the table will be a 500 x 3 matrix of integer values. There is

room for optimization because the matrix is a sparse matrix; many integer values will be zero, e.g. capacity rows 0-199. When solving the problem, the decision variable values are:

$$\begin{aligned} x_1 &= 0 \\ x_2 &= 1 \\ x_3 &= 0 \end{aligned} \tag{3.5}$$

Only one task is scheduled because of the objective function. In this case, a task's value was set equally to the task's weight. Redefining the objective function to $x_1 + x_2 + x_3$ yields the decision variable values:

$$\begin{aligned} x_1 &= 1 \\ x_2 &= 0 \\ x_3 &= 1 \end{aligned} \tag{3.6}$$

Both objective functions yield an optimal objective value of 500, which is node 1's maximum capacity.

However, the knapsack problem only solved the scheduling of three tasks on a single node with only one variable. More dimensions, or multiple knapsacks, are required to facilitate the scheduling of tasks on multiple nodes, let alone multiple variables.

The memory footprint is $\mathcal{O}(K \cdot n)$ integers per knapsack, where n is the number of items. Assume an integer consumes 1 byte of memory. For a problem where a single node, or knapsack, has $K = 100,000$ capacity, there are $n = 10,000$ tasks to be scheduled and only a single variable, the storage requirement is:

$$\begin{aligned} & \mathcal{O}(K \cdot n) \\ = \\ & 100,000 \cdot 10,000 = 1,000,000,000 \text{ required bytes} \\ = \\ & \frac{1,000,000,000}{1024 \cdot 1024} \approx 954 \text{ MB (bytes to kilobytes to megabytes)} \end{aligned} \tag{3.7}$$

Almost one gigabyte of memory is required to construct the table. Even with many optimizations, dynamic programming seems inefficient as an alternative to linear programming for task scheduling due to its memory footprint required to obtain the scheduling and its flexibility of solving multiple problem dimensions.

3.2 Branch and Bound

The branch and bound algorithm [15] is discussed in subsection 4.6.1 to solve integer linear programming problems. Branch and bound can be used as a technique to solve the knapsack problem. Since, in a sense, the knapsack problem is similar to the scheduling problem, branch and bound is evaluated as an alternative. The example described in subsection 5.4.1 is used for illustration purposes.

Assume the case of scheduling multiple tasks on one node with only one variable. After depth-first branch and bound has been completed, the resulting tree looks like the tree shown in figure 3.1.

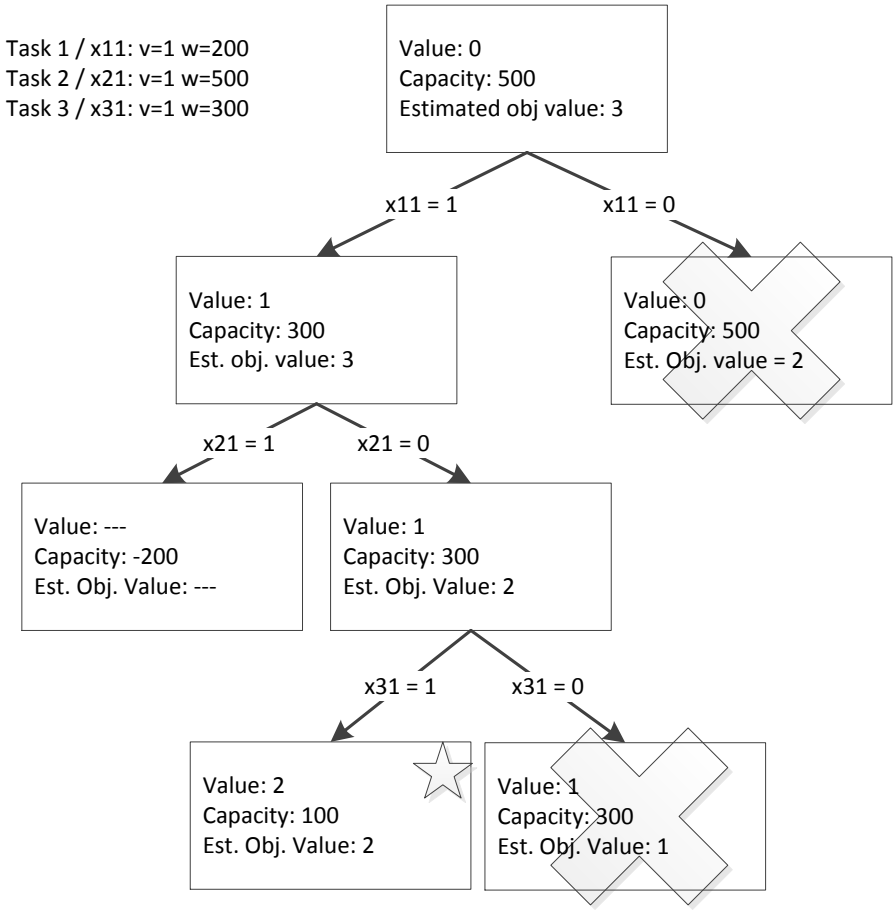


Figure 3.1: branch and bound tree for scheduling three tasks on one node with one variable

At the root node, the residual capacity of the node is 500 and the estimated objective value is 3. The estimated value of 3 is the total number of nodes possible in the scheduling. Every level in the binary search tree represents a task, either chosen or not chosen. The node where task 1 is chosen; the capacity is decreased by the weight of the task, the value is updated to 1 and the estimated objective value is unchanged because it is possible for all tasks to be scheduled regardless of capacity

constraints. After tasks 1 has been chosen, task 2 is considered as chosen. The node had a residual capacity of 300 whereas task 2 requires 500. Choosing task 1 and task 2 results in an infeasible solution because the capacity constraint has been violated.

The node where task 1 and task 3 are chosen, $x_{11} = 1, x_{21} = 0, x_{31} = 1$, is a leaf node that has a feasible solution. At this point all tasks were considered with an optimal value of 2 and a residual capacity of 100. The neighboring leaf node has been pruned away since the objective value is lower than the highest value already yielded.

The $x_{11} = 0$ branch can be pruned away since the best objective value the branch can yield is 2. Since the best objective value is already 2, the branch cannot yield a better solution. Any other greedy technique would possibly have explored all possible combinations, depth-first branch and bound explored only 6 nodes in this particular case. Other heuristics may produce different results.

Assume the same example with two variables. The resulting depth-first branch and bound tree would look like the tree shown in figure 3.2. The idea is the same for more variables.

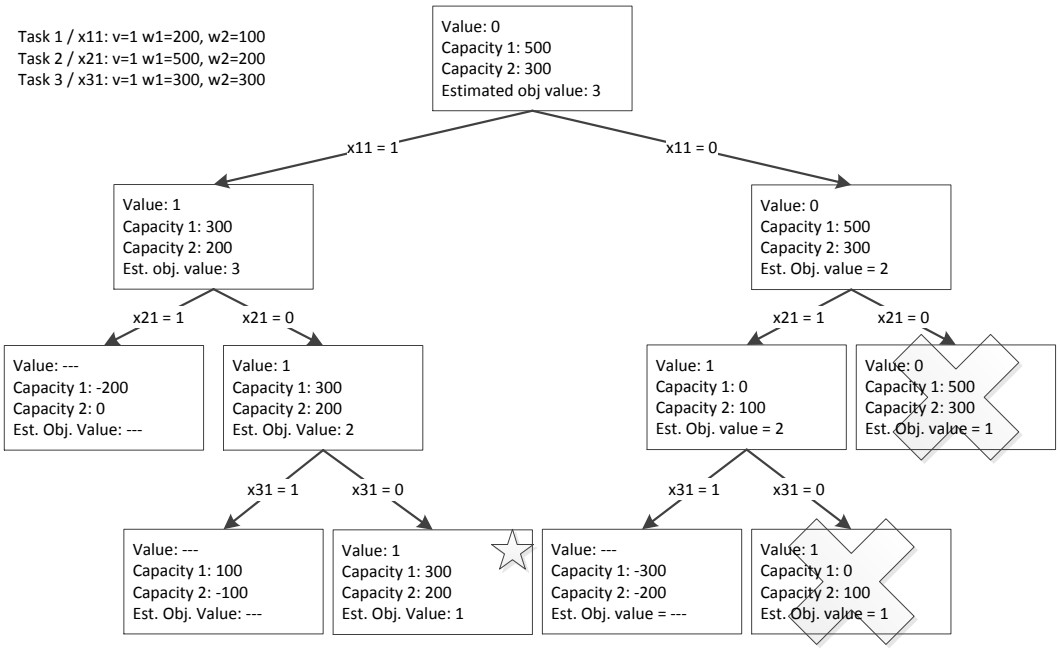


Figure 3.2: branch and bound tree for scheduling three tasks on a single node with two variables

For some v number of variables, v number of capacity variables are added to each node to respect the capacity constraints. At each level, the task's corresponding weights w_1 and w_2 are subtracted from the residual capacity variables $capacity_1$ and $capacity_2$ of the parent node when the task is selected, i.e. where $x_{ij} = 1$.

In this particular case, a scheduling of only task 1 is optimal, $x_{11} = 1, x_{21} = 0, x_{31} = 0$, due to the capacity constraints. Another possible scheduling is $x_{11} = 0, x_{21} = 1, x31 = 0$ but is pruned away because there was already a schedule found with the same objective value. In this case, the traversal method used influences which tasks are

scheduled, task 1 versus task 2.

Assume the case of the full example; two nodes, three tasks and a single variable. The example's depth-first branch and bound tree is shown in figure 3.3.

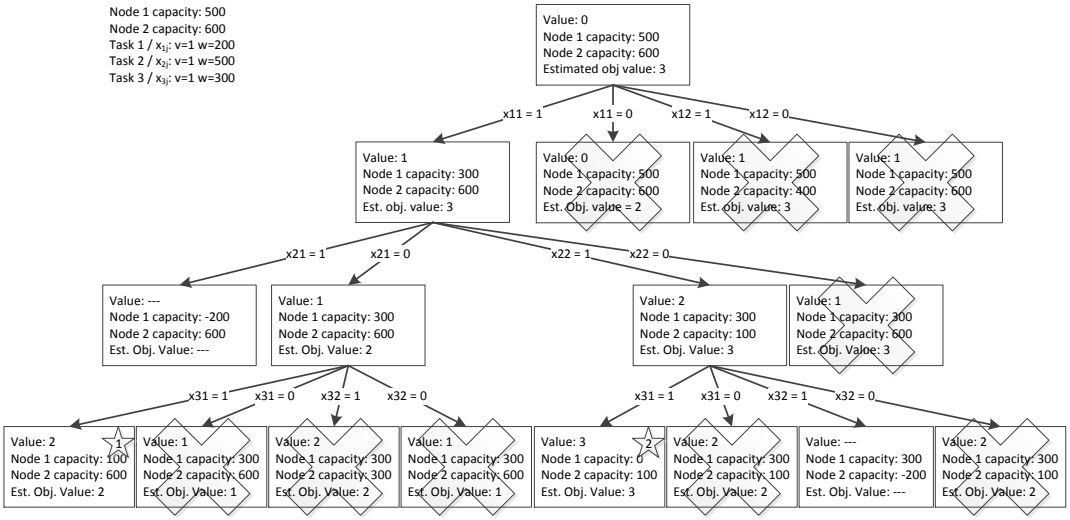


Figure 3.3: branch and bound tree for scheduling three tasks on two nodes with a single variable

At each level, all task's corresponding decision variables are explored to include all nodes in the scheduling. With depth-first branch and bound, the first objective value is found with configuration $x_{11} = 1, x_{21} = 0, x_{31} = 1$ with an objective value of 2. The node with the first feasible solution is depicted by a star with a one written in it. The second feasible solution is found in the configuration $x_{11} = 1, x_{22} = 1, x_{31} = 1$ with objective value 3, depicted by a star with a two written in it. The best possible estimate, see root node, is the value 3. After the second objective value has been found, all other nodes can be pruned away.

Branch and bound seems like a good alternative to linear programming in terms of task scheduling. In the example given, the technique is rather efficient in terms of explored nodes. The minimal number of nodes explored is largely dependent on the depth-first traversal method. Another method of traversal might not have pruned away nodes in an early stage.

3.3 Constraint Programming

Where linear programming and branch and bound focus aggressively on increasing to objective value, constraint programming [15] focusses on the problem's constraints. A constraint problem, or constraint satisfaction problem, uses constraints to prune the search space by eliminating values that cannot belong to any solution.

A propagation engine is at the base of constraint programming. A decision variable is chosen and a value is assigned to the variable by a search engine. The constraints are checked for feasibility. The search space is pruned when the given values are feasible. This process is repeated until no values can be pruned and the variable domain is feasible. The variable domain is the search space for each variable. In other words, each variable has lower and upper bounds to the value it accepts.

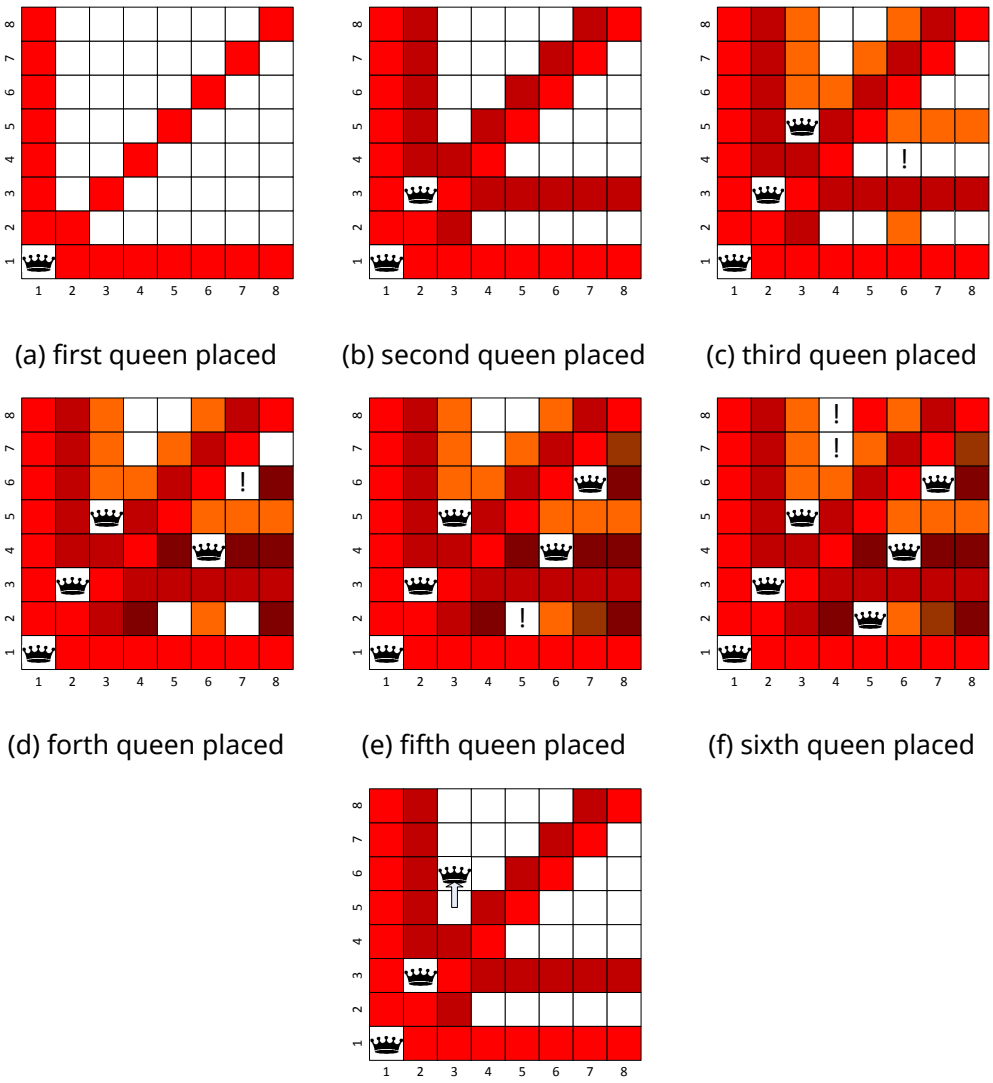
The search engine is basically branching on decision variable values in a search tree. A very basic search is to evaluate every possible value. Once the propagation engine reports an infeasible solution, another value is chosen for the decision variable. When no values were accepted for that decision variable, the algorithm backtracks to a previous decision variable to choose a different value and the process continues. Because possible values are removed in the variable domain, by propagation, the search space is decreased. When assigning a value to a decision variable, forward checking validates the corresponding constraints and temporarily prunes the search space. If forward checking detects that the current value is infeasible, it proceeds to the next value to prune the search space faster.

Formally, a constraint satisfaction problem is defined as a triplet $<X, D, C>$, where

$$\begin{aligned} &X = \{X_1, \ldots, X_n\} \text{ set of variables,} \\ &D = \{D_1, \ldots, D_n\} \text{ set of domain values and} \\ &C = \{C_1, \ldots, C_m\} \text{ set of constraints.} \end{aligned} \tag{3.8}$$

Each domain value corresponds to a variable. The variable's corresponding domain value bounds the solution space. The possible domain values decrease by the propagation engine with respect to the constraints.

To illustrate constraint programming, consider the 8-queens problem. The problem states that eight queens have to be placed on an 8 by 8 square matrix in such a way that only one queen is located on each row, each column and each diagonal. Figure 3.4 visually shows constraint processing with forward checking of a single branch in the search tree. Figure 3.4a places the first queen on position x=1 and y=1. The second and third queens are placed at the next available positions, as seen in figures 3.4b and 3.4c. Pruning the variable domain forces the fourth, fifth and the sixth queen to be positioned at a fixed location, as seen in figures 3.4c, 3.4d and 3.4e. As depicted in figure 3.4f, placement of the seventh and eight queen results in an infeasible solution. The algorithm has to backtrack until a previously set variable can be changed. In this case, the third queen is able to change its position, as depicted in figure 3.4g. From this position, the process continues until a feasible solution is found. When the solution space did not yield a solution, the problem is infeasible. The objective of the n-queens problem is to obtain a single solution, i.e. a satisfaction problem.



(a) first queen placed

(b) second queen placed

(c) third queen placed

(d) forth queen placed

(e) fifth queen placed

(f) sixth queen placed

(g) alternative placement of third queen

Figure 3.4: Eight queens problem with forward checking

In case of task scheduling, constraint programming can be set up similarly to branch and bound in where each node in the search tree represents a different scheduling. The root node is the scheduling of zero tasks. Child nodes represent a scheduling in where possibly only one task is added to its parent's scheduling. The leaf nodes represent a feasible or an infeasible scheduling. Since the root node represents a feasible solution, i.e. no scheduling satisfies the constraints, an objective function has to be maximized, e.g. the sum of scheduled tasks.

The triplet <X, D, C> for the example described in subsection 5.4.1, with three tasks and two nodes and a single variable is:

$$\begin{aligned} X =& \{task_1, task_2, task_3\} \\ D =& \{(1,2),(1,2),(1,2)\} \\ C =& 200 \cdot task_1(1) + 500 \cdot task_2(1) + 300 \cdot task_3(1) \leq 500 \\ & 200 \cdot task_1(2) + 500 \cdot task_2(2) + 300 \cdot task_3(2) \leq 600 \end{aligned} \tag{3.9}$$

$task_i$ is a variable that represents the scheduling of task i on node j, where $j \in D_i$.

The constraints are similar to the linear programming model, where

$$task_i(j) = \begin{cases} 0 & \text{when } task_i \neq j, j \in D_i \\ 1 & \text{when } task_i = j, j \in D_i \end{cases} \tag{3.10}$$

The definition of a feasible solution has to be adjusted. It is possible for a scheduling to not include one or more tasks. It is implied that a task is not scheduled when a variable's domain is empty.

The solution for the example is depicted in figure 3.5. The residual capacity of each node is visualized, in green, at the bottom of each node. The task requirements are visualized at the top left of the figure, in variations of red. The variable capacity and requirements are drawn to scale.

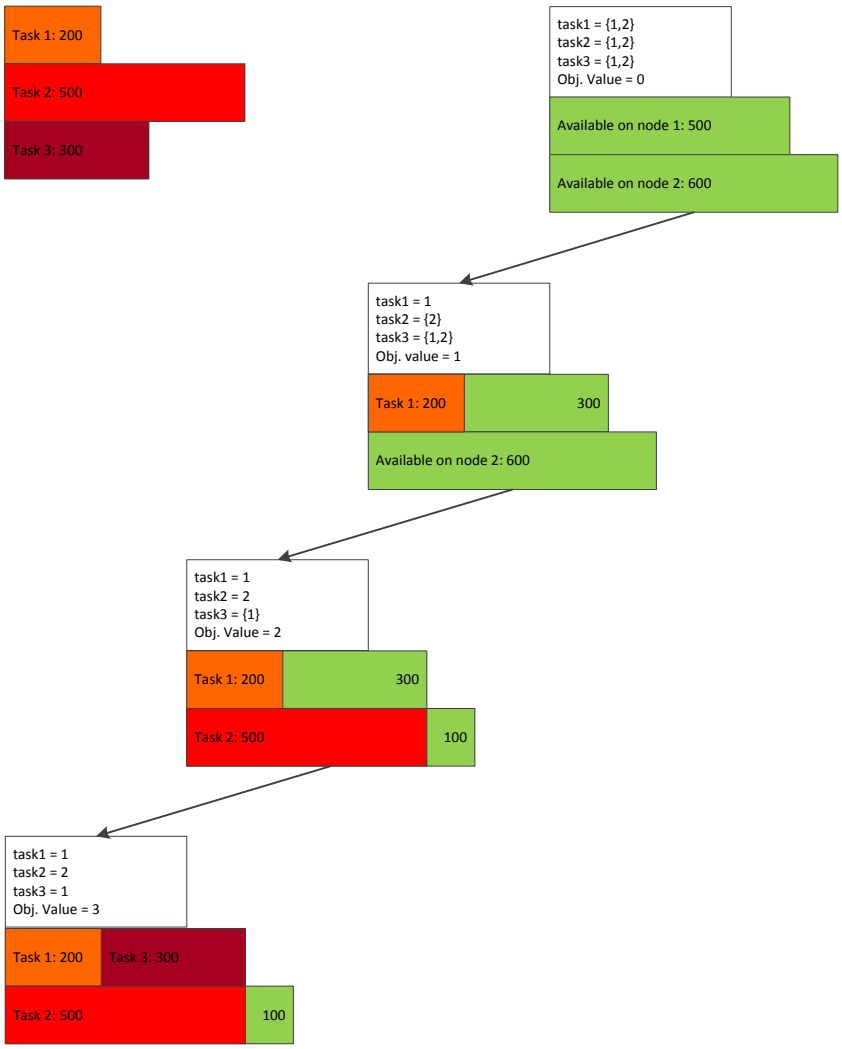


Figure 3.5: CSP search tree based on example described in subsection 5.4.1 with a single variable

The scheduling is straightforward, no backtracking is required. Assume another variable has to be considered in the scheduling to simulate backtracking. Figure 3.6 depicts the search tree for three tasks, two nodes and two variables. The task requirements are shown at the top-left and top-right positions of the figure.

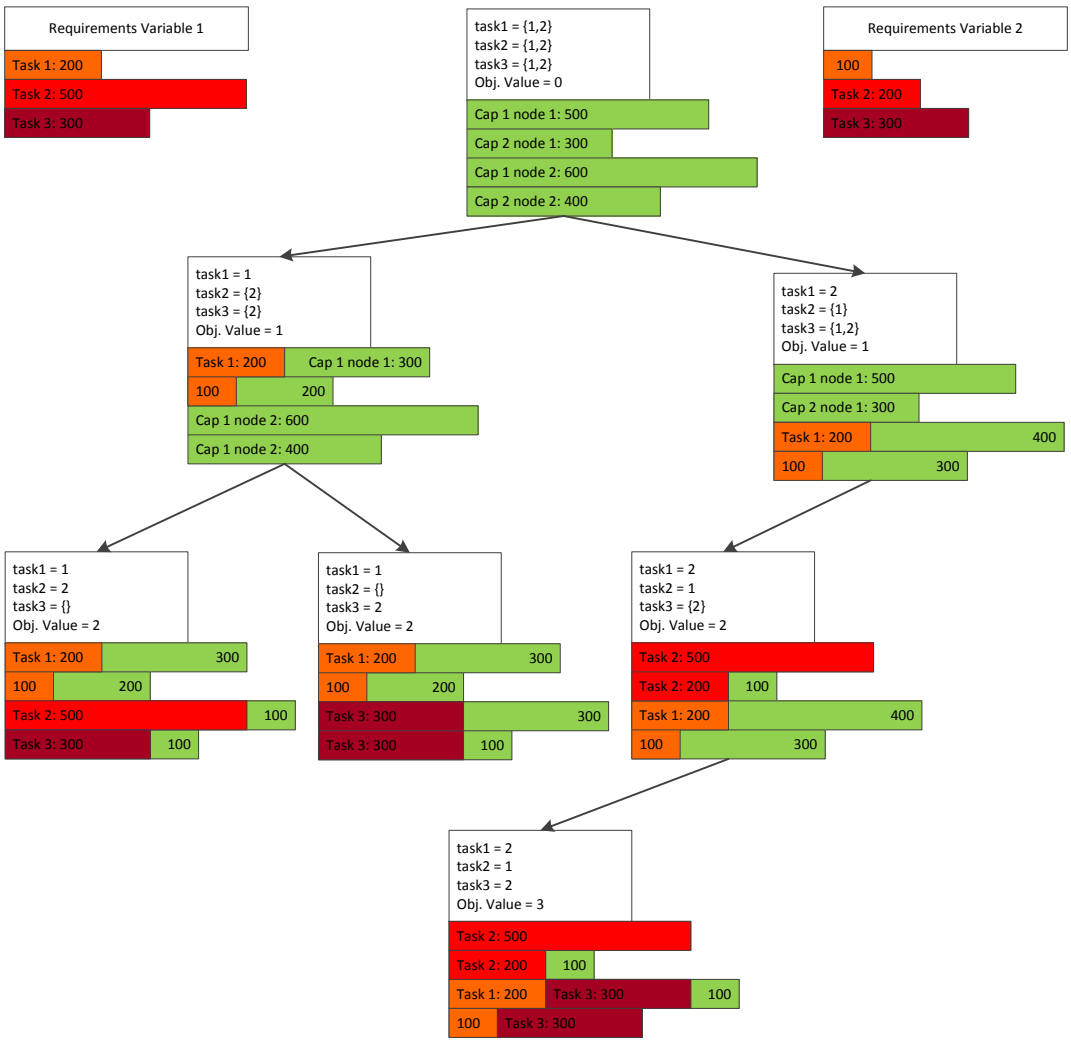


Figure 3.6: CSP search tree based on example described in subsection 5.4.1 with two variables

The left branch produces two leaf nodes with both an objective value of 2. While this is a feasible solution to the maximization problem, backtracking is performed to find a better objective value in the right branch. The right branch clearly demonstrates the propagation, only a single value can be chosen for the variables due to the capacity constraints.

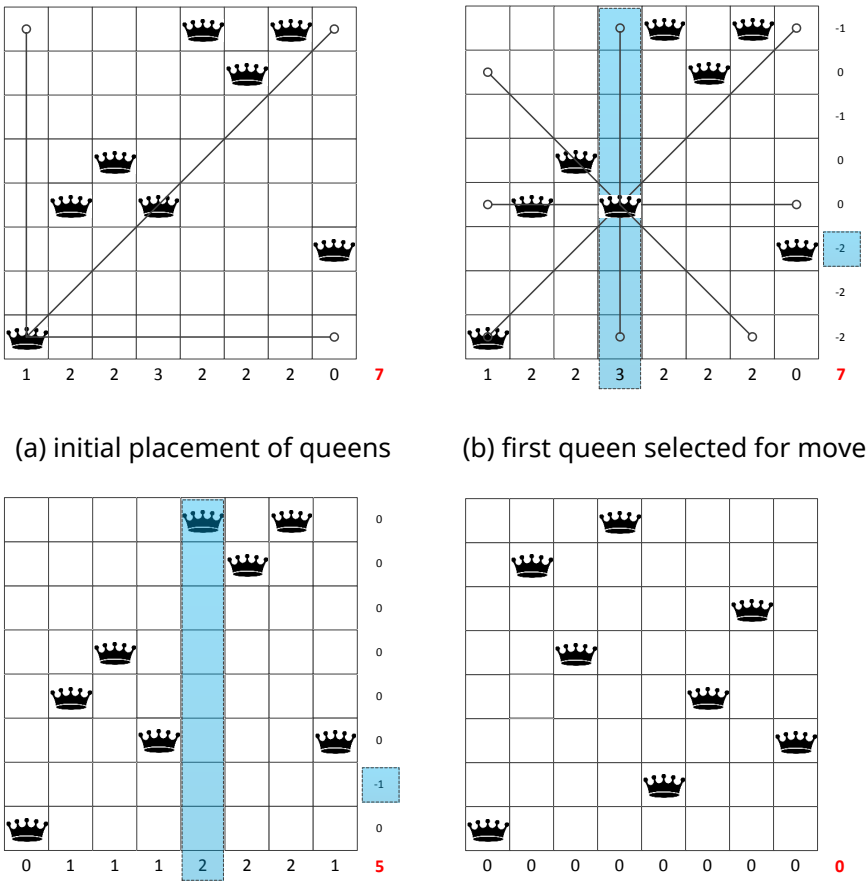
Constraint programming seems like a very good alternative to linear programming with respect to task scheduling. However, the search tree can grow significantly when more tasks, nodes and variables are added. The heuristic used in the search engine becomes important to prune as much values as possible at each level in the tree.

3.4 Local Search

Local search [15] is a technique to find local optima. Local optimum is the optimal value found in a neighborhood, which, in turn, is a subset of the entire problem. Local search starts with any arbitrary chosen configuration within the problem space. Many constraints are likely to be violated. Local search selects a neighborhood, obtains a local optima and updates the solution space accordingly. Local search repeatedly explores local neighborhoods until all options have been explored to yield the found solution. A neighborhood and local optima can be defined differently among problems.

The eight queens problem is a satisfaction problem, i.e. a single feasible placement of queens is sufficient. The neighborhood of the eight queens problem can be seen as the constraints of each queen. Recall that only one queen can be placed in each row, each column and each diagonal. The local optima, or local minima, can be seen as the number of constraint violations by the selected queen in its neighborhood. The idea is that a solution emerges when constraint violations are removed. No constraint violations implies that all queens are placed such that the problem is feasible.

Figure 3.7a depicts an arbitrary placement of queens onto the board's columns. The neighborhood of the queen located at the bottom-left is depicted by the horizontal, vertical and diagonal lines. The diagonal line overlaps a single queen which implies that one constraint is violated. The number of violations for each queen is shown below each column. The right most number, number 7 depicted in red, is the number of global constraints violated, i.e. a global constraint is violated when there are multiple queens on the same row, same column, or same diagonal. The objective is to minimize the number of global constraint violations, i.e. zero global constraint violations is a feasible solution. In figure 3.7b, the queen on the forth column is selected because the heuristic chosen selects the queen with the most local constraint violations. It is possible to move the queen to any other position on the same column. All possible options are explored and each option yields an increase, decrease, or remains the same in the number of constraints violated. According to the same heuristic, the selected queen should be moved down one row decreasing the number of local violations by two. The constraint violations have been recalculated and the next queen is selected, as depicted in figure 3.7c. Also, the number of global constraint violations has decreased. This process continues until a feasible solution is found or it is determined that the problem is infeasible. Infeasibility is obtained when all possible positions have been explored and all positions violate one or more constraints. In this case, after some steps, the feasible solution is found and is depicted in figure 3.7d.



(a) initial placement of queens

(b) first queen selected for move

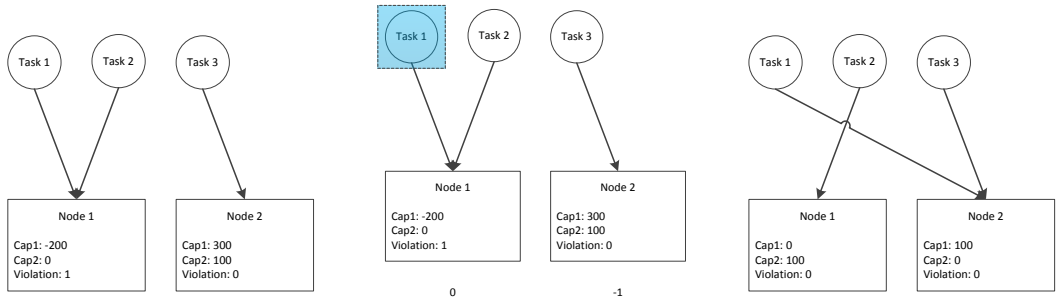
(c) first queen moved and second queen selected

(d) solution to the eight queens problem

Figure 3.7: Eight queens problem with local search

In context of task scheduling, a task's neighborhood can include a subset or all nodes. The global constraints are binary and represent whether or not each node's total capacity is exceeded. Local constraints are, in this case, redundant to the global constraints. The objective is to minimize the global constraints' violations. No constraint violations is optimal since it implies that all tasks are able to be scheduled.

Assume the example described in subsection 5.4.1 with two variables and variable requirements of 100, 200 and 300 for each node to illustrate a possible local search for task scheduling. An initial arbitrary allocation of tasks is depicted in figure 3.8a. The global constraint violations are shown inside the boxes, representing nodes. For reference, the node's capacity is also added inside the boxes. Based on the initial allocation, node 1's global constraint is violated by allocating task 1 and task 2.



(a) initial arbitrary allocation (b) task 1 selected for alternative allocation (c) final scheduling

Figure 3.8: Scheduling of example described in subsection 5.4.1 with local search and two variables

Figure 3.8b shows that a simple heuristic is used to select task 1 as a rescheduling candidate to reduce violations. Below the boxes, the effect on the number of global constraint violations is shown. In this case, task 1 should be scheduled to node 2 to obtain a decrease in the number of global constraint violations. Figure 3.8c shows that the rescheduling of task 1 produced an optimal scheduling without constraint violations.

The given example does not demonstrate the importance of heuristics because the problem space is not large enough. The example uses a simple but expensive heuristic to check every possible node. There are many possible heuristics; an alternative heuristic is to only consider a subset of neighboring nodes, e.g. only consider nodes that do not violate their corresponding global constraint. Chances are nodes that do not violate their global constraint have residual capacity.

A problem in local search is that finding local optima does not necessarily yield the best possible solution to a problem [15]. With some other combination it might be possible to yield a better solution, the global minima, or maxima depending on the problem. Depending on the heuristic, local search can be greedy and the algorithm will converge to a local minima with the least constraint violations. This local minima has to be escaped to obtain the global minima. Figure 3.9 illustrates the possible optimas.

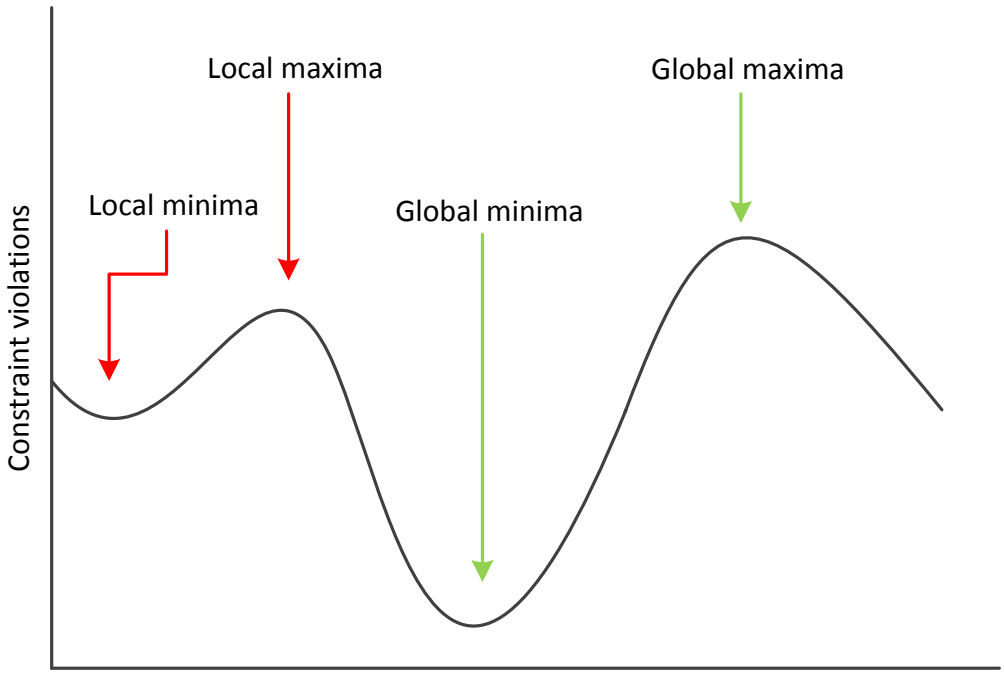


Figure 3.9: Local and global optima

Several metaheuristics are available to escape from a local optima and to obtain the global optima. Iterated local search, simulated annealing and tabu search are examples of metaheuristics. Iterated local search [15] starts from different, randomly chosen, starting positions, e.g. different combinations, to obtain local optima. After a number of iterations, the best local optima is considered the global optima. Simulated annealing [15] uses a temperature function to decide whether a combination is further explored. Low temperatures basically decides randomly, i.e. browse around, in the feasible field and high temperatures basically selects the first combination it encounters. Simulated annealing starts with a cold temperature that is increased with every decision. Several options are available such as reheating to explore an alternative combination. Tabu search [15] is a metaheuristic that keeps track of already explored combinations to force the algorithm to move away from a local optimum, i.e. climb over a hill to find other optima. The algorithm cannot move to a previous explored combination since that combination is marked 'tabu'.

Local search for task scheduling is challenging since a good (meta)heuristic has to be obtained to speed up the process, exploring every possible combination is too expensive. The given local search solution is not sufficient as a possible solution for task scheduling because every task has to be allocated in order to have no constraint violations. It is naive to say to remove one or more nodes from a global constraint violated node because there might be a potentially better solution.

3.5 Conclusion

Dynamic programming seems inefficient for task scheduling as its memory footprint is too high to obtain the scheduling and it is not flexible in terms of problem scaling. The memory footprint comes from the fact that the constructed table needs traversal in order to obtain the actual scheduling instead of just the objective value.

Branch and bound seems like a good alternative to linear programming in terms of task scheduling as long as efficient heuristics are used. Branch and bound's variant, branch and cut is commonly used as an integer linear programming problem solver. Branch and cut [10] is a combination of branch and bound and cutting planes, as discussed in chapter 4, to reduce the depth of the tree.

Constraint programming seems like a good alternative as long as the search space can be significantly pruned and the problems remain manageable in terms of size. Constraint programming is very similar to branch and bound in terms of constructing a search tree.

In local search, the (meta) heuristics are an important aspect for local search. Although local search seems possible to be part of a task scheduler, the discussed solution is not because all tasks have to be scheduled to eliminate all constraint violations.

Heuristics are very important in discrete optimization algorithms for searching and pruning. The discrete optimization algorithms discussed potentially have a large search space and therefor depend on a well-performing heuristic. Although linear programming does not require heuristics, integer linear programming does require a discrete algorithm to solve the problem. Branch and cut is almost used by all ILP solvers [11]. Since linear programming produces an optimal solution, branch and cut can be used to prune away suboptimal, feasible, solutions to limit tree depth. Linear programming provides more flexibility in terms of tree traversal heuristics and cutting planes can be added to a problem to limit the search space.

4 Linear Programming

A variety of mathematical methods are available to solve optimization problems. Optimization problems generally consist of a finite number of constraints and an objective function. The objective function is maximized or minimized while respecting each constraint.

Linear programming is one of the mathematical methods to solve and optimize an optimization problem. Linear programming can be used if and only if the objective function and the constraints are linear.

Introducing linear programming is done best by example [2], the diet problem is introduced in section 4.1. The standard form of describing LPs is introduced in section 4.2 and section 4.3 describes how to convert a LP into standard form. Section 4.4 describes the simplex algorithm to solve linear programming problems. The duality of a linear programming problem is described in section 4.5 and section 4.6 describes how to solve integer linear programming problems.

4.1 Diet Problem

The diet problem [2, 11] is a well-suited problem to demonstrate Linear Programming. The diet problem is about how much money a person must spend on food in order to get all the required energy, protein, and calcium that somebody needs every day. Six foods for the source of the nutrients are collected in table 4.1. It is assumed that a person requires 2,000 kcal of energy, 55 grams of protein, and 800 mg each day. Of course there are a lot more factors and nutritions to be taken into account but the problem is kept simple for illustrative purposes.

Food	Serving size	Energy (kcal)	Protein (g)	Calcium (mg)	Price per serving (cents)
Oatmeal	28 g	110	4	2	3
Chicken	100 g	205	32	12	24
Eggs	2 large	160	13	54	13
Whole milk	237 cc	160	8	285	9
Cherry pie	170 g	420	4	22	20
Pork with beans	260 g	260	14	80	19

Table 4.1: Nutritive value per serving

Theoretically, a person can eat multiple servings of one food to pass the minimal required amount of nutrients. Since this is not very practical, a servings-per-day limit is imposed on all six foods:

Oatmeal	at most 4 servings per day
Chicken	at most 3 servings per day
Eggs	at most 2 servings per day
Milk	at most 8 servings per day
Cherry pie	at most 2 servings per day
Pork with beans	at most 2 servings per day.

It is possible to come up with many possible combinations with each combination having a price. A trial and error approach is not particularly helpful in finding the best combination. The best solution, or combination, here is considered to be the cheapest menu while fulfilling the minimal required nutrients. A more systematic approach would be speculating about an undefined menu consisting of x_1 servings of oatmeal, x_2 servings of chicken, x_3 servings of eggs, x_4 servings of milk, x_5 servings of cherry pie, and x_6 servings of pork with beans. These undefined variables are the decision variables of the diet problem. The decision variables must respect the upper bound of servings per day:

$$\begin{aligned} 0 \leq x_1 \leq 4 \\ 0 \leq x_2 \leq 3 \\ 0 \leq x_3 \leq 2 \\ 0 \leq x_4 \leq 8 \\ 0 \leq x_5 \leq 2 \\ 0 \leq x_6 \leq 2 \end{aligned} \tag{4.1}$$

The required energy, protein, and calcium also have to be satisfied, which lead to the following inequalities

$$\begin{aligned} 110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 &\geq 2,000 \\ 4x_1 + 33x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 &\geq 55 \\ 2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 &\geq 800 \end{aligned} \tag{4.2}$$

If some values, numbers, for $x_1, x_2, \ldots, x_6$ satisfy inequalities (4.1) and (4.2), then a feasible menu is found. A found menu might be feasible but it might not be the optimal menu, i.e. the most economical menu. The price of a menu is specified by

$$3x_1 + 24x_2 + 13x_3 + 9x_4 + 20x_5 + 19x_6. \tag{4.3}$$

To find the most economical menu, values for $x_1, x_2, \ldots, x_6$ have to be found which satisfy inequalities (4.1) and (4.2), and minimize (4.3).

The diet problem can be written in mathematical form:

$$
\begin{array}{ll}
\text{minimize} & 3x_1 + 24x_2 + 13x_3 + 9x_4 + 20x_5 + 19x_6 \\
\text{s.t.} & 0 \leq x_1 \leq 4 \\
& 0 \leq x_2 \leq 3 \\
& 0 \leq x_3 \leq 2 \\
& 0 \leq x_4 \leq 8 \\
& 0 \leq x_5 \leq 2 \\
& 0 \leq x_6 \leq 2 \\
110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 & \geq 2,000 \\
4x_1 + 33x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 & \geq 55 \\
2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 & \geq 800
\end{array}
\tag{4.4}
$$

The linear inequality (4.3) has to be minimized and is the objective function of the LP. The objective function z of variables $x_1, x_2, \ldots, x_6$ is defined by

$$z(x_1, x_2, \ldots, x_6) = 3x_1 + 24x_2 + 13x_3 + 9x_4 + 20x_5 + 19x_6$$

Decision variable values that satisfy all constraints is considered a feasible solution. A feasible solution for the diet problem (4.3) is

$$x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 8, x_5 = 2, x_6 = 0$$

The corresponding objective value for this solution is

$$z = 3 \cdot 0 + 24 \cdot 0 + 13 \cdot 0 + 9 \cdot 8 + 20 \cdot 2 + 19 \cdot 0 = 112 \text{ cents}$$

However, a better feasible solution can be found. The optimal solution for the diet problem (4.3) turns out to be

$$x_1 = 4, x_2 = 0, x_3 = 0, x_4 = 4.5, x_5 = 2, x_6 = 0$$

The corresponding objective value for this solution is

$$z = 3 \cdot 4 + 24 \cdot 0 + 13 \cdot 0 + 9 \cdot 4.5 + 20 \cdot 2 + 19 \cdot 0 = 92.5 \text{ cents}$$

To recap, the variables $x_1, x_2, \ldots x_6$ are unknown decision variables. The objective function (4.3) is an equation and its goal is to either maximize or minimize z. The problem is constrained by linear inequalities and/or linear equalities. The goal of the problem is to find values for the decision variables such that the constraints are satisfied and, in the optimal case, the obtained value in the objective function is optimal. Decision variable solutions that produce a suboptimal objective solution are called feasible solutions.

In general, if $c_1, c_2, \ldots, c_n$ are real numbers, then the function f of real decision variables $x_1, x_2, \ldots, x_n$ defined by

$$f(x_1, x_2, \ldots, x_n) = c_1x_1 + c_2x_2 + \ldots + c_nx_n = \sum_{j=1}^{n} c_jx_j$$

is called a linear function.

If f is a linear function and if b is a real number, then the equality

$$f(x_1, x_2, \ldots, x_n) = b$$

is called a linear equality and the inequalities

$$f(x_1, x_2, \ldots, x_n) \leq b$$

$$f(x_1, x_2, \ldots, x_n) \geq b$$

are called linear inequalities. Linear equalities and linear inequalities are both referred to as linear constraints [2].

Not every LP problem has a unique optimal solution; some problems may have multiple optimal solutions and some problems have no optimal solution at all.

4.2 Standard Form

A single LP problem constraint can either be a linear equality or a linear inequality. For simplicity LP problems are restricted to the standard form. The terminology of the standard form is not unified; some authors also refer to this form as the canonical or symmetric form, and other authors might reserve this term for entirely different problems. The standard form is distinguished from other forms by only allowing linear less-than-or-equal inequality constraints, the objective function is maximized and every decision variable have to be non-negative, e.g. a non-negativity constraint is added for every decision variable. Negative decision values change the nature of a problem. A generalization of the standard form:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i = 1, 2, \ldots, m) \\ & x_j \geq 0 \quad (j = 1, 2, \ldots, n) \end{aligned} \tag{4.5}$$

Similarly:

$$\begin{aligned} \text{maximize} \quad & c_1 x_1 + \ldots + c_n x_n && \leq b_1 \\ \text{s.t.} \quad & a_{11} x_1 + \ldots + a_{1n} x_n && \leq b_2 \\ & \qquad \ddots && \vdots \\ & a_{m1} x_1 + \ldots + a_{mn} x_n && \leq b_m \\ & x_1, \ldots, x_n && \geq 0 \end{aligned} \tag{4.6}$$

where m is the number of constraints, n is the number of decision variables, c_j is the j-th coefficient corresponding to the j-th decision variable x_j in the objective function, a_{ij} is the j-th coefficient of the i-th linear constraint and b_i is the i-th coefficient of the constraint's right-hand side.

The standard form in matrix notation:

$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{s.t.} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned} \tag{4.7}$$

where

$$\mathbf{c} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ & \ddots & \\ a_{m1} & \dots & a_{mn} \end{pmatrix} \mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \tag{4.8}$$

4.3 Standard Form conversion

Other forms of LP problems are available. Conversion from other forms to the standard form requires simple steps. After converting to the standard form and solving the original problem in standard form, a reverse conversion is required to end up with the original form.

The objective function is maximized in the standard form. If the original's objective function is minimized, a conversion is required. The conversion from minimization to maximization is to inverse the sign of the function.

In general notation:

$$\text{minimize} \sum_{j=1}^{n} c_j x_j \xrightarrow{\text{change to}} \text{maximize} \sum_{j=1}^{n} -c_j x_j \tag{4.9}$$

in matrix notation:

$$\text{minimize } \mathbf{c}^T\mathbf{x} \xrightarrow{\text{change to}} \text{maximize } (-\mathbf{c}^T\mathbf{x}) \tag{4.10}$$

an example:

$$\text{minimize } -5x_1 + 4x_2 - 3x_3 \xrightarrow{\text{change to}} \text{maximize } 5x_1 - 4x_2 + 3x_3 \tag{4.11}$$

There are no equality constraints in the standard form. Linear equalities are substituted by two inequality constraints. The two inequality constraints are less-than-or-equal and greater-than-or-equal constraints. Both inequality constraints simulate an equality constraint.

in general notation:

$$a_i x_i = b_i \xrightarrow{\text{change to}} \begin{cases} a_i x_i \leq b_i \\ a_i x_i \geq b_i \end{cases} \tag{4.12}$$

in matrix notation:

$$\mathbf{a}_i^T\mathbf{x} = b_i \xrightarrow{\text{change to}} \begin{cases} \mathbf{a}_i^T\mathbf{x} \leq b_i \\ \mathbf{a}_i^T\mathbf{x} \geq b_i \end{cases} \tag{4.13}$$

an example:

$$2x_1 - 3x_2 + x_3 = 5 \xrightarrow{\text{change to}} \begin{cases} 2x_1 - 3x_2 + x_3 \leq 5 \\ 2x_1 - 3x_2 + x_3 \geq 5 \end{cases} \tag{4.14}$$

Less-than-or-equal linear inequalities do not have to be converted since the standard form requires all constraints to be in that form. Greater-than-or-equal inequalities have to be converted to their less-than-or-equal equivalent by changing the sign.

in general notation:

$$a_i x_i \geq b_i \xrightarrow{\text{change to}} -a_i x_i \leq -b_i \tag{4.15}$$

in matrix notation:

$$\mathbf{a}_i^T \mathbf{x} \geq b_i \xrightarrow{\text{change to}} -\mathbf{a}_i^T \mathbf{x} \leq -b_i \tag{4.16}$$

an example:

$$2x_1 - 3x_2 + x_3 \geq 5 \xrightarrow{\text{change to}} -2x_1 + 3x_2 - x_3 \leq -5 \tag{4.17}$$

In standard form, all decision variable values have to be non-negative. Non-negativity constraints have to be added to LP problems that lack the non-negativity constraint on one or more decision variables. It is not possible to simple add the non-negativity constraint $x_i \geq 0$ to the problem because then the whole meaning of the problem is changed. For every missing non-negativity constraint two new variables are introduced: x_i^+ and x_i^-. Every occurrence of x_i has to be replaced with the two newly introduced variables: $x_i \mapsto x_i^+ - x_i^-$. Any number of x_i can be written as the difference between two non-negative numbers. For example, the number 7 can be written as $7 - 0$ and the number -7 can be written as $0 - 7$. Non-negativity constraints are added to both decision variables x_i^+ and x_i^-.

The following example [11] combines all the described transformations:

$$\begin{array}{rrcl}
\text{minimize} & -5x_1 + 4x_2 - 3x_3 & & \\
\text{s.t.} & 2x_1 - 3x_2 + x_3 & = & 5 \\
& 4x_1 + x_2 + 2x_3 & \geq & 11 \\
& 3x_1 + 4x_2 + 2x_3 & \leq & 8 \\
& x_1 & \geq & 0
\end{array}$$

after conversion:

$$\begin{array}{rrcl}
\text{maximize} & 5x_1 - 4x_2^+ + 4x_2^- + 3x_3^+ - 3x_3^- & & \\
\text{s.t.} & 2x_1 - 3x_2^+ + 3x_2^- + x_3^+ - x_3^- & \leq & 5 \\
& -2x_1 + 3x_2^+ - 3x_2^- - x_3^+ + x_3^- & \leq & -5 \\
& -4x_1 - x_2^+ + x_2^- - 2x_3^+ - 2x_3^- & \leq & -11 \\
& 3x_1 + 4x_2^+ - 4x_2^- + 2x_3^+ - 2x_3^- & \leq & 8 \\
& x_1, \quad x_2^+, \quad x_2^-, \quad x_3^+, \quad x_3^- & \geq & 0
\end{array}$$

4.4 Simplex

The simplex method [2, 11, 15], invented by George Danzig, is a mathematical method to solve or determinate infeasibility of a linear program in a finite number of steps. In theory it is possible that solutions cycle which hinders converging to an optimal solution. Simplex is an iterative hill-climbing algorithm that tries to find a feasible solution after every iteration.

4.4.1 Slack variables

Given a linear inequality constraint, $\mathbf{a^T x} \leq b$, the left-hand side is at most the value of the right-hand side; often, there may be a difference between the two values. The difference between the values of the left-hand side and the right-hand side is called slack. With slack, it is possible to rewrite linear inequalities to linear equalities. From a computational perspective, linear equalities are easier to process.

$$\mathbf{a}^T\mathbf{x} \leq b \xrightarrow{\text{add slack}} \mathbf{a}^T\mathbf{x} + x_{slack} = b$$

For every linear constraint, one new slack variable is introduced. The value for x_{slack} has to be a non-negative value. A non-negativity constraint is added to the problem for each slack variable.

Consider the following problem [11]:

$$\begin{array}{rl} \text{maximize} & 5x_1 + 4x_2 + 3x_3 \\ \text{s.t.} & 2x_1 + 3x_2 + x_3 \leq 5 \\ & 4x_1 + x_2 + 2x_3 \leq 11 \\ & 3x_1 + 4x_2 + 2x_3 \leq 8 \\ & x_1,\ x_2,\ x_3 \geq 0 \end{array}$$

After changing the problem to a problem with added slack, the problem would look like

$$\begin{array}{rl} \text{maximize} & 5x_1 + 4x_2 + 3x_3 \\ \text{s.t.} & 2x_1 + 3x_2 + x_3 + x_4 = 5 \\ & 4x_1 + x_2 + 2x_3 + x_5 = 11 \\ & 3x_1 + 4x_2 + 2x_3 + x_6 = 8 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{array} \tag{4.18}$$

4.4.2 Dictionary

The dictionary is a notation, or data structure, which represents a state of the LP. The dictionary might reflect the initial dictionary or a (sub)optimal solution yielded by the simplex algorithm. A data structure is required for the simplex algorithm to perform operations on, to store on hard disk and for readability.

In matrix form, a dictionary is defined as:

$$\begin{array}{rll} \mathbf{x_B} & = \mathbf{b} & -\mathbf{A}\mathbf{x_I} \\ \hline z & = c_0 & +\mathbf{c}^T\,\mathbf{x}_i \end{array} \tag{4.19}$$

where $\mathbf{x_B}$ is the set of basic variables, $\mathbf{x_I}$ is the set of non-basic variables, and z represents the objective function with c_0 as its obtained value. Initially, objective value of z, c_0, is set to zero, the slack variables form the basic set and the decision variables form the non-basic set. Non-negativity constraints are implicit in the dictionary.

A more general form of a dictionary:

$$\begin{array}{rcl} x_{B1} & = & b_1 + a_{11}x_{I1} + \ldots + a_{1j}x_{Ij} + \ldots + a_{1n}x_{In} \\ & \vdots & \\ x_{Bk} & = & b_k + a_{k1}x_{I1} + \ldots + a_{kj}x_{Ij} + \ldots + a_{kn}x_{In} \\ & \vdots & \\ x_{Bm} & = & b_m + a_{m1}x_{I1} + \ldots + a_{mj}x_{Ij} + \ldots + a_{mn}x_{In} \\ \hline z & = & c_0 + c_1x_{I1} + \ldots + c_jx_{Ij} + \ldots + c_nx_{In} \end{array} \tag{4.20}$$

where m is the number of basic variables and is equal to the number of constraints, and where n is the number of non-basic constraints and is equal to the number of decision variables.

A solution can be read from a dictionary with ease by setting every non-basic variable to zero and plug the value of each basic variable into the objective function. The slack variables are initially basic because the objective function does not have terms for the slack variables and therefor the objective value is zero. In other words, each term in the objective function evaluates to zero because every decision variable is set to zero. Variables will be swapped among the basic and non-basic set at each simplex iteration.

The corresponding initial dictionary of problem (4.18) is:

$$\begin{array}{rl} x_4 &= 5 \ -2x_1 -3x_2 \ -x_3 \\ x_5 &= 11 -4x_1 -x_2 \ -2x_3 \\ x_6 &= 8 \ -3x_1 -4x_2 -2x_3 \\ \hline z &= 0 \ +5x_1 +4x_2 +3x_3 \end{array} \tag{4.21}$$

The variables and their corresponding values are:

$$\begin{aligned} x_1 &= 0 \\ x_2 &= 0 \\ x_3 &= 0 \\ x_4 &= 5 \\ x_5 &= 11 \\ x_6 &= 8 \end{aligned}$$

where $\mathbf{x_I} = \{x_1, x_2, x_3\}$
and $\mathbf{x_B} = \{x_4, x_5, x_6\}$

Plugging the values into the objective function yields:

$$\begin{array}{c} 5x_1 + 4x_2 + 3x_3 = 0 \\ = \\ 5 \cdot 0 + 4 \cdot 0 + 3 \cdot 0 = 0 \end{array}$$

Setting the value of every non-basic variable to zero also implies that the basic variables receive their corresponding $\mathbf{b}$ value because $-\mathbf{A x_I}$ evaluates to zero.

The dictionary represents a feasible solution if all values of vector $\mathbf{b}$, in any dictionary, are non-negative. Otherwise, the dictionary represents an infeasible solution.

4.4.3 Algorithm

The simplex algorithm consists of two phases, the initialization and optimization phase. The initialization phase is only required for linear programming problems that are initially infeasible. The initialization phase performs a finite amount of pivoting steps to obtain a feasible or infeasible dictionary. Once a feasible dictionary is found, the optimization phase continues to pivot a finite number of steps until a final dictionary is obtained. The final dictionary contains the optimal solution to the linear programming problem.

Pivoting is the method to move from one dictionary to the next dictionary in the simplex algorithm. A single pivot performs three tasks; selecting an entering variable, selecting a leaving variable, and perform substitutions. An entering variable is a variable that is non-basic and a leaving variable is a variable that is basic. The entering variable becomes basic and the leaving variable becomes non-basic. In essence, pivoting is the method of swapping two variables and updating the dictionary accordingly.

There are n non-basic variables to choose an entering variable from. Since that the objective function needs to be maximized, it is important to choose an entering variable which increases objective value z the most. To increase z, $c_j x_{Ij}$ (4.19) has to be increased, which is done by increasing the value of non-basic variable x_{Ij}. The value of z can only increase when $c_j > 0$. A decrease of z cannot be obtained due

to the non-negativity constraints imposed on all variables. Each x_{Ij} of the $c_j x_{Ij}$ term that increases z is a possible entering variable. When every $c_k x_{Ij} \leq 0$, the dictionary is final and the optimal solution has been found. In other words, there is no entering variable available to enter the basis and therefor the dictionary is final.

After choosing an entering value, a leaving variable has to be chosen. The leaving variable is the variable that limits the increase of x_{Ij} the most. In other words, x_{Ij} is increased as much as possible to increase z but x_{ij} cannot be increased infinitely. When x_{Ij} is increased, the value of each basic variable b_k should decrease to maximize the corresponding constraint. The value of each basic variable b_k can only decrease if and only if $a_{kj} < 0$, otherwise the constraint is unbounded and x_{Ij} can be set to $+\infty$. The constraint corresponding to a basic variable that restricts the increase of x_{Ij} the most is the leaving variable. The maximum increase of x_{Ij} for each constraint is calculated by $\frac{b_k}{-a_{kj}}$. When every $a_{kj} \geq 0$, the original problem is unbounded because the increase of x_{Ij} is not bounded and thus $+\infty$ can be used. There is no need to pivot further because every value of x_{Ij} is a feasible value. To recap, select the basic variable corresponding to x_{Ij} that has the most stringent increase as leaving variable:

$$x_{Ij} \leq \begin{cases} \frac{b_k}{-a_{kj}} (\text{if } a_{kj} < 0) \\ +\infty (\text{if } a_{kj} \geq 0) \end{cases} \tag{4.22}$$

After the entering and leaving variables are determined, the dictionary can be updated by performing a series of substitutions to yield a new dictionary. The dictionary row of the leaving variable has to be replaced by a new row for the entering variable. All other rows contain the entering variable; these variables have to be substituted in terms of the leaving variable. The new row of the entering variable is defined by

$$x_{Ij} = \frac{b_k}{-a_{kj}} + \frac{a_{k1}}{-a_{kj}} x_{I1} + \ldots + \frac{a_{kn}}{-a_{kj}} x_{In} + \frac{1}{a_{kj}} x_{Bk}. \tag{4.23}$$

All other basic variable rows have to be updated in terms of the leaving variable, defined by

$$x_{Bk} = b_k + a_{k1} x_{I1} + \ldots + a_{kj} x_{Ij} + \ldots + a_{kn} x_{In} \tag{4.24}$$

where x_{Ij} is substituted by equation (4.23) which can also be written as

$$x_{Bk} = \left(b_k - a_{kj} \frac{b_i}{a_{ij}} \right) + \left(a_{k1} + a_{kj} \frac{a_{i1}}{-aij} \right) x_{I1} + \cdots + \left(a_{kn} + a_{kj} \frac{a_{in}}{-a_{ij}} \right) x_{In} \tag{4.25}$$

where i is the i-th row index position in x_{Ij}.

For dictionary (4.21), the entering variable of this problem can either be one of the non-basic variables since every variable is able to increase the objective value. Assume x_1 as the entering variable, then a basic variable has to leave. Which basic variable has to leave depends on the constraint that is the most stringent on the entering variable's increase. The constraint on the increase for each basic variable is

$$\begin{array}{rcrcrcrcl} x_4 & = & 5 & -2x_1 & = & \frac{5}{2} & = & 2.5 & \rightarrow \quad x_1 \leq 2.5 \\ x_5 & = & 11 & -4x_1 & = & \frac{11}{4} & = & 2.75 & \rightarrow \quad x_1 \leq 2.75 \\ x_6 & = & 8 & -3x_1 & = & \frac{8}{3} & \approx & 2.667 & \rightarrow \quad x_1 \leq\sim 2.667. \end{array} \tag{4.26}$$

The entering variable x_1 cannot be increased more than the value 2.5 because otherwise x_4 would violate its non-negativity constraint. The variable x_4 constraints the increase of x_1 the most since x_5 and x_6 allow a higher value. Since x_4 is the most stringent basic variable, x_4 is chosen to be the leaving variable.

According to (4.23) x_4 is substituted in terms of x_1.

$$
\begin{aligned}
& x_4 = 5 - 2x_1 - 3x_2 - x_3 \\
= & \\
& x_1 = \tfrac{5}{2} - \tfrac{3}{2}x_2 - \tfrac{1}{2}x_3 - \tfrac{1}{2}x_4
\end{aligned}
$$

Other basic variables are updated according to (4.24), or (4.25), to substitute x_5 in terms of x_1

$$
\begin{aligned}
& x_5 = \quad 11 - 4(x_1) - x_2 - 2x_3 \\
= & \quad (4.24) \\
& x_5 = \quad 11 - 4(\tfrac{5}{2} - \tfrac{3}{2}x_2 - \tfrac{1}{2}x_3 - \tfrac{1}{2}x_4) - x_2 - 2x_3 \\
= & \\
& x_5 = \quad 11 - 10 + 6x_2 + 2x_3 + 2x_4 - x_2 - 2x_3 \\
= & \\
& x_5 = \quad 1 + 5x_2 + 0x_3 + 2x_4.
\end{aligned}
$$

After all the basic variable rows have been substituted, the resulting dictionary after one pivot is

$$
\begin{array}{rl}
x_1 & = \frac{5}{2} \; -\frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\
x_5 & = 1 \; +5x_2 \qquad\quad +2x_4 \\
x_6 & = \frac{1}{2} \; +\frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\
\hline
z & = \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4
\end{array}
\tag{4.27}
$$

From the dictionary, a solution can be read. However this is not a final dictionary because an entering variable can be found for another round of pivoting. When variable x_3 enters and variable x_6 leaves, the next dictionary will be the final dictionary

$$
\begin{array}{rl}
x_1 & = 2 \; -2x_2 - 2x_4 \; +x_6 \\
x_3 & = 1 \; +x_2 \; +3x_4 - 2x_6 \\
x_5 & = 1 \; +5x_2 + 2x_4 \\
\hline
z & = 13 - 3x_2 - x_4 \quad -x_6
\end{array}
\tag{4.28}
$$

The objective value for the final dictionary is $z = 13$ where $x_1 = 2$, $x_2 = 0$, $x_3 = 1$, $x_4 = 0$, $x_5 = 1$, and $x_6 = 0$. The objective value can be verified by the objective function and the variables:

$$
\begin{aligned}
& 5x_1 + 4x_2 + 3x_3 \\
= & \\
& 5 \cdot 2 + 4 \cdot 0 + 3 \cdot 1 \\
= & \\
& 10 + 0 + 3 \\
= & \\
& 13.
\end{aligned}
$$

4.4.4 Initialization phase

Linear programming problems that are initially infeasible require initialization, i.e. there is at least one negative value in the b column vector:

$$\begin{pmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_m \end{pmatrix} < 0.$$

An infeasible problem does not necessarily imply that the problem does not have a feasible solution. The feasible solution is simply not found by setting zeros for every decision variable. In other words, a feasible solution is not found at the origin $(0, 0)$ for a two dimensional problem. It is possible to modify the original problem such that the problem might become feasible. The modified problem is called an auxiliary problem. To set up the auxiliary problem, a new non-basic variable x_0 is introduced, with a corresponding non-negativity constraint. The purpose of x_0 is to find a value where $b_i + x_0$ becomes non-negative. After the variable x_0 is added, the simplex method is performed to find the minimal value for x_0 that is required for the problem to become feasible. This implies that the objective function is set to minimizing x_0 in the auxiliary problem.

Simplex is forced to let x_0 enter the basis in the initial auxiliary dictionary for the first pivoting step. The leaving basic variable is the variable with the corresponding b value that is most negative. The auxiliary dictionary becomes feasible after the first pivot. Simplex is performed until a final dictionary is obtained. Whenever x_0 is a possible leaving variable, x_0 will be preferred to leave. The original problem is feasible if and only if the final auxiliary dictionary has the objective value $z = 0$. The original problem is infeasible when $z < 0$ or x_0 is basic.

After finding the final feasible solution to the auxiliary problem, the auxiliary dictionary is used to form the original problem's initial dictionary for the optimization phase. The variable x_0 and the objective function is removed from the auxiliary dictionary which leaves the original problem's dictionary without the original problem's objective value and function. The objective function is constructed by substituting the original problem's objective function by its corresponding terms in the auxiliary dictionary. The resulting dictionary is now feasible and ready for the optimization phase.

Consider the infeasible initial dictionary [11] shown in (4.29)

$$\begin{array}{rl} x_3 &= -2 + 2x_1 - x_2 \\ x_4 &= 4 \quad + 0x_1 - x_2 \\ x_5 &= -2 - x_1 \quad + 2x_2 \\ x_6 &= 4 \quad - x_1 \quad + 0x_2 \\ \hline z &= 0 \quad + x_1 \quad + 2x_2 \end{array} \tag{4.29}$$

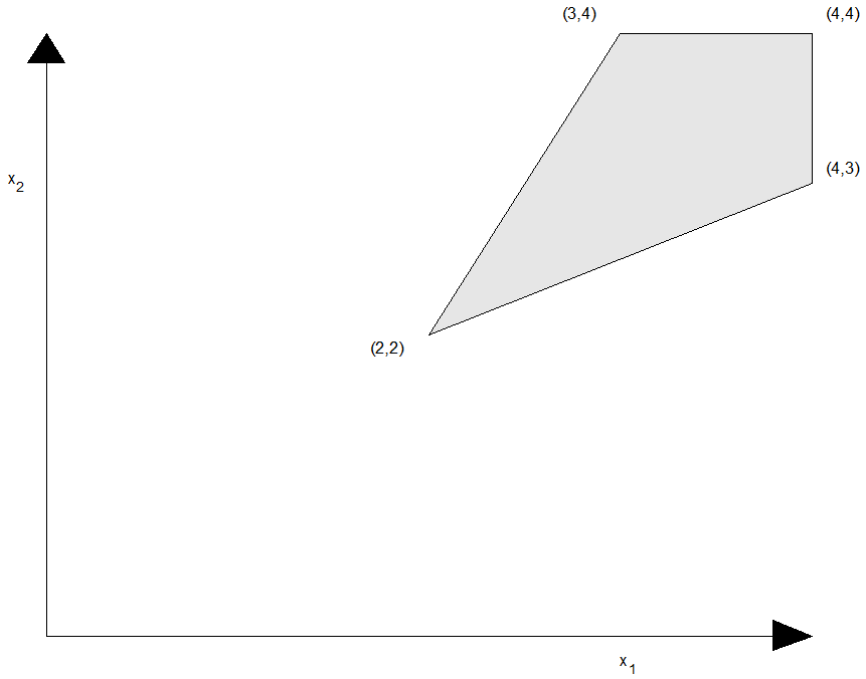


Figure 4.1: Example (4.29)'s geometry

The corresponding visual, geometry, representation of example (4.29) is show in figure 4.1. There is no solution for $x_1 = 0$ and $x_2 = 0$ because $x_3, x_5 < 0$. As the geometry shows, there is a feasible region but its origin is not at $x_1 = 0, x_2 = 0$. Initialization phase simplex is required for this problem to become feasible. The initial auxiliary dictionary of (4.29), in standard form, is:

$$\begin{array}{rl} x_3 &= -2 + x_0 + 2x_1 - x_2 \\ x_4 &= 4 \quad + x_0 + 0x_1 - x_2 \\ x_5 &= -2 + x_0 - x_1 \quad + 2x_2 \\ x_6 &= 4 \quad + x_0 - x_1 \quad + 0x_2 \\ \hline z &= 0 \quad - x_0 \end{array}$$

For the first pivoting step, x_0 is the entering variable. The leaving variable is chosen to be x_3 while x_5 could also be chosen. After the first pivoting step, the dictionary is as follows:

$$\begin{array}{rl} x_0 &= 2 \quad + x_3 - 2x_1 + x_2 \\ x_4 &= 6 \quad + x_3 - 2x_1 + 0x_2 \\ x_5 &= 0 \quad + x_3 - 3x_1 + 3x_2 \\ x_6 &= 6 \quad + x_3 - 3x_1 + x_2 \\ \hline z &= -2 - x_3 + 2x_1 - x_2 \end{array} \tag{4.30}$$

which results in a feasible, but not final, auxiliary dictionary. The b value of x_0 is set to 2 which implies that 2 is the minimal number required for every b_i to become

non-negative, $b_i + 2 \geq 0$. After two subsequent pivoting steps, x_0 has left the basis and the following final auxiliary dictionary is found:

$$\begin{array}{rl} x_1 &= 2 + \frac{2}{3}x_3 + \frac{1}{3}x_5 + 0x_0 \\ x_2 &= 2 + \frac{1}{3}x_3 + \frac{2}{3}x_5 + 0x_0 \\ x_4 &= 2 - \frac{1}{3}x_3 - \frac{2}{3}x_5 + 2x_0 \\ x_6 &= 2 - \frac{2}{3}x_3 + \frac{4}{3}x_5 + 2x_0 \\ \hline z &= 0 + 0x_3 + 0x_5 - x_0 \end{array} \tag{4.31}$$

The objective value $z = 0$ implies that there is a feasible solution to the original problem. After stripping x_0 and the objective function from the dictionary, the original objective function has to be substituted in:

$$\begin{array}{rl} z &= x_1 + 2x_2 \\ &= \left(2 + \frac{2}{3}x_3 + \frac{1}{3}x_5\right) + 2\left(2 + \frac{1}{3}x_3 + \frac{2}{3}x_5\right) \\ &= \left(2 + \frac{2}{3}x_3 + \frac{1}{3}x_5\right) + \left(4 + \frac{2}{3}x_3 + \frac{4}{3}x_5\right) \\ &= 6 + \frac{4}{3}x_3 + \frac{5}{3}x_5 \end{array} \tag{4.32}$$

The first feasible dictionary for the original problem is:

$$\begin{array}{rl} x_1 &= 2 + \frac{2}{3}x_3 + \frac{1}{3}x_5 \\ x_2 &= 2 + \frac{1}{3}x_3 + \frac{2}{3}x_5 \\ x_4 &= 2 - \frac{1}{3}x_3 - \frac{2}{3}x_5 \\ x_6 &= 2 - \frac{2}{3}x_3 + \frac{4}{3}x_5 \\ \hline z &= 6 + \frac{4}{3}x_3 + \frac{5}{3}x_5 \end{array} \tag{4.33}$$

which can be used in the optimization phase. By looking at the geometry of figure (4.1), it is easy to see that the obtained dictionary is in the feasible region of the original problem.

4.4.5 Entering and leaving variable heuristics

Often there is more than one entering or leaving variable to choose for pivoting. Which entering or leaving variable is chosen depends on the heuristic used.

The largest objective coefficient heuristic [11] is an entering variable heuristic that chooses the maximal coefficient c_j. The idea is simple: the larger the value of c_j, the more increase in z per unit increase of x_{Ij}. The heuristic does not consider the net increase that can be obtained, which depends on the chosen leaving variable, i.e. for any $c_k < c_j$, there might be larger x_{Ik} that yields a better increase of z.

The greedy choice heuristic [11] explores all possible options. All possible entering and leaving variable combinations are examined to obtain the combination that yields the most increase in z. The greedy choice heuristic might increase significantly in the first step but z's increase might flatten after the next step. Also, there could be a better yield of z after the second step when choosing a suboptimal combination in the first step.

Bland's rule [2, 11] is a heuristic that always chooses the lowest possible index value for both the entering and the leaving variable. The entering variable is x_{Ij} where j is the lowest possible index and the leaving variable is x_{Bk} where k is the lowest possible index.

Bland's rule is not the best performing heuristic since it does not look at the problem's context but Bland's rule guarantees termination [11]. The greedy choice heuristic tends to perform best for practical problems [11]. It is possible to use more

than one heuristic to solve a problem. For example, it is possible to switch to Bland's rule after a cycle is detected with the greedy choice heuristic.

Consider the following dictionary [11]:

$$\begin{array}{rl} x_3 &= 1 \ +3x_1 -25x_2 \\ x_4 &= 11 +0x_1 -x_2 \\ x_5 &= 3 \ -x_1 \ +x_2 \\ x_6 &= 3 \ -x_1 \ +0x_2 \\ \hline z &= 0 \ +x_1 \ +25x_2 \end{array} \tag{4.34}$$

Possible entering variables are x_1 and x_2. When x_1 enters, possible leaving variables are x_5 and x_6, and when x_2 enters, possible leaving variables are x_3 and x_4.

The largest coefficient heuristic chooses x_2 as the entering variable since $25 > 1$. The greedy choice heuristic examines all possible combinations. Examining x_1 as the entering variable, the maximum increase is 3 for either x_5 or x_6 as leaving variable since $x_5 \leq 1/3$ and $x_6 \leq 1/3$. Examining x_2 as the entering variable, the maximum increase is $1/25$ when x_3 enters and $11/1$ when x_4 enters. The greedy choice heuristic will choose x_1 as entering variable and arbitrary choose x_5 or x_6 as leaving variable. Bland's rule will choose x_1 as entering variable and x_5 as leaving variable because of their indexes.

4.4.6 Degeneracy, cycling and stalling

During the optimization phase of simplex, a finite number of dictionaries are produced. The objective value z cannot decrease because of each pivoting step, but the objective value does not strictly have to increase. It is possible for simplex to yield the same objective value after a pivoting step. Dictionaries that yield the same objective value after the next pivoting step are called degenerate.

The objective value of the next dictionary is determined by

$$z' = z + c_j \left(\frac{b_i}{-a_{ij}} \right) \tag{4.35}$$

where c_j cannot have a negative value because of the entering variable analysis. It is implied that term $\frac{b_i}{-a_{ij}}$ has to be non-negative to yield a zero or an increase. To yield a zero, either b_i or $-a_{ij}$ will be zero. The variable a_{ij} is strictly negative because of the leaving variable analysis. The value of b_i has to be zero for the problem to yield $z = z'$. In other words, a dictionary is degenerate when a $b_j = 0$.

Simplex does not necessarily terminate in a finite number of steps. Since simplex does not generate infinitely many dictionaries, the algorithm must cycle through previously generated dictionaries to prevent termination. In other words, dictionaries D_i to D_j are repeating. Cycling dictionaries $D_i, ..., D_j$ are degenerate dictionaries, i.e. there is no increase in the objective value while cycling. Cycling depends on the chosen heuristic in entering and leaving variable analysis. Bland's rule is a heuristic that prevents cycling [11].

When not using Bland's rule, detecting cycling by storing each obtained dictionary and comparing dictionaries is very inefficient. A possible method to overcome cycling is to keep track of the number of subsequent degenerate dictionaries and switch to Bland's rule after a k-specified number of subsequent degenerate dictionaries.

Like cycling, stalling is the notion of a sequence of dictionaries that are degenerate but unlike cycling, stalling breaks the cycle after a finite number of pivoting steps and increases the objective value. Cycling and stalling only occurs in highly degenerate problems and are considered more theoretical than practical [11].

4.4.7 Complexity

The worst-case complexity of the simplex algorithm is exponential in theory. In practice, the average complexity is polynomial [11].

The choice of entering variable is $\mathcal{O}(n)$ and the choice of leaving variable is $\mathcal{O}(m)$. The complexity of a single pivot operation is $\mathcal{O}((m+n)^3)$. The upper bound on the number of dictionaries is $\frac{(n+m)!}{m!n!}$ [11].

V. Klee and G.J. Minty (1972) have shown that the simplex method goes through $2^n - 1$ iterations for any number of decision variables n for a specific problem. An example dictionary of the Klee-Minty problem [2] with $n = 3$ is:

$$\begin{array}{rlll} x_4 & = 1 & -x_1 & \\ x_5 & = 100 & -20x_1 & -x_2 \\ x_6 & = 10,000 & -200x_1 & -20x_2 - x_3 \\ \hline z & = 0 & +100x_1 & +10x_2 + x_3 \end{array} \tag{4.36}$$

with its corresponding geometry representation:

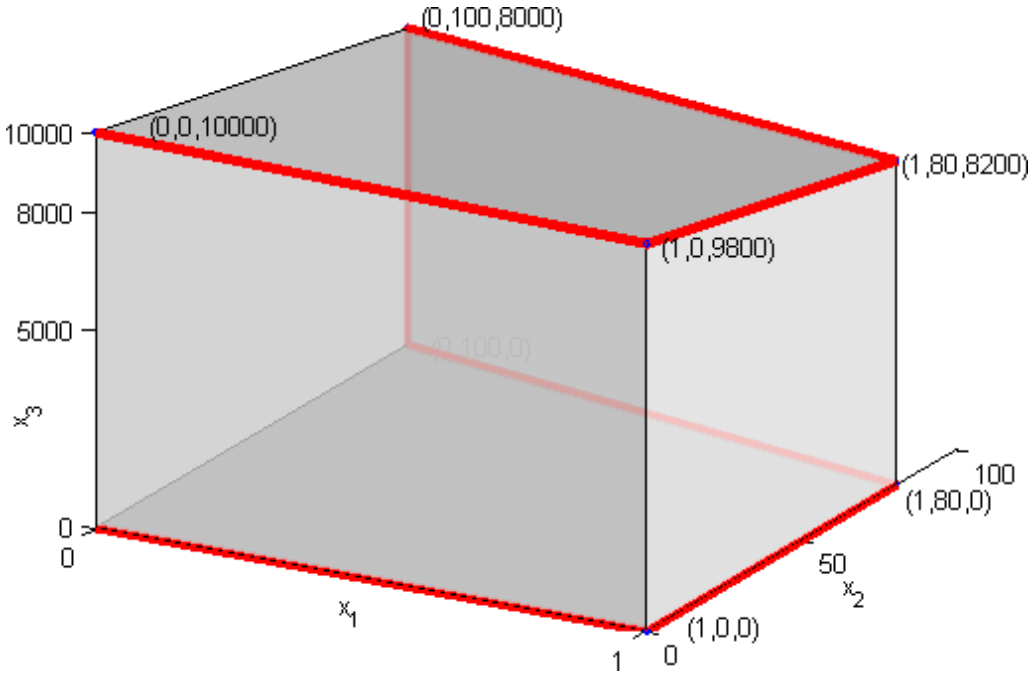


Figure 4.2: Example of a Klee-Minty "distorted" cube

The heuristic chosen significantly influences the number of iterations required in order to obtain the optimal value. The largest objective coefficient is used in Klee-Minty examples; its behavior is shown in figure 4.2. With the largest objective coefficient heuristic, the simplex algorithm will yield the optimal solution after seven iterations, as shown in red, while the greedy heuristic would have yielded the optimal

solution after only one iteration. It is visible in figure 4.2 that the solution at point (0,0,10000) could be reached after one iteration, with a greedy heuristic.

The storage complexity of simplex with dictionaries is $\mathcal{O}(m \cdot n)$ floating point numbers per dictionary. Storage complexity can be reduced by reconstructing the dictionary at every pivoting step [11].

4.5 Duality

Every maximization LP problem in the standard form has a corresponding minimization problem, the dual problem. The dual problem finds upper bounds on the original's problem. The original LP is commonly referred to as the primal problem. The objective of the dual problem is to find the most stringent upper bound. The dual LP problem is solved like the primal LP problem using the simplex algorithm. The optimal solution to the dual problem is optimal solution of the primal LP.

The dual defined as:

$$\begin{aligned} \text{minimize} \quad & \mathbf{b}^T\mathbf{y} \\ \text{s.t.} \quad & A^T\mathbf{y} \geq \mathbf{c} \\ & \mathbf{y} \geq 0 \end{aligned} \tag{4.37}$$

Note that the definition is not in standard form because it is a minimization problem and the linear inequalities differ. Furthermore, the primal's data is transposed and reused. The real difference between the primal and dual is that the dual decision variables are indicated by $\mathbf{y}$.

Consider the infeasible problem shown in figure 4.29:

$$\begin{aligned} \text{maximize} \quad & x_1 + 2x_2 \\ \text{s.t.} \quad & -2x_1 + x_2 \leq -2 \\ & +x_2 \leq 4 \\ & x_1 - 2x_2 \leq -2 \\ & x_1 \leq 4 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{4.38}$$

Given the definition of the primal problem (4.7), the primal's problem data is:

$$\mathbf{c} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \mathbf{b} = \begin{bmatrix} -2 \\ 4 \\ -2 \\ 4 \end{bmatrix} A = \begin{bmatrix} -2 & 1 \\ 0 & 1 \\ 1 & -2 \\ 1 & 0 \end{bmatrix} \tag{4.39}$$

According to the definition of the dual problem (4.37), its corresponding dual problem is:

$$\begin{aligned} \text{minimize} \quad & -2y_1 + 4y_2 - 2y_3 + 4y_4 \\ \text{s.t.} \quad & -2y_1 + y_3 + y_4 \geq 1 \\ & y_1 + y_2 - 2y_3 \geq 2 \\ & y_1, y_2, y_3, y_4 \geq 0 \end{aligned} \tag{4.40}$$

The definition of the dual problem in standard form is

$$\begin{aligned} \text{maximize} \quad & -\mathbf{b}^T\mathbf{y} \\ \text{s.t.} \quad & -A^T\mathbf{y} \leq -\mathbf{c} \\ & \mathbf{y} \geq 0 \end{aligned} \tag{4.41}$$

The decision variables in $\mathbf{y}$ have to be non-negative because otherwise the multiplier changes the polarity of the problem constraints.

The dual dictionary becomes:

$$\begin{array}{rl} y_1 & = -1 - 2y_3 \qquad\quad + y_5 \;\; + y_6 \\ y_2 & = -2 + y_3 \;\; + y_4 \;\; - 2y_5 \\ \hline z & = 0 \quad + 2y_3 - 4y_4 + 2y_5 - 4y_6 \end{array}$$

Like the primal, the dual problem requires initialization since there is a $c_j < 0$. The primal and the dual problems are two different problems but are related to each other, the primal finds an upper bound while the dual finds a lower bound. All possible primal-dual combinations are:

		Dual		
		Optimal	Infeasible	Unbounded
Primal	Optimal	Possible	Impossible	Impossible
	Infeasible	Impossible	Possible	Possible
	Unbounded	Impossible	Possible	Impossible

Table 4.2: Primal-Dual combinations

The primal problem has an optimal solution if and only if the dual problem has an optimal solution. The dual of the dual problem is the primal problem.

4.6 Integer Linear Programming

Integer Linear Programming (ILP) problems are similar to regular linear programming problems, they differ in the requirement that every decision variable is an integer. The integrality constraint $x_1, \ldots, x_n \in \mathbb{Z}$ is added to the LP to obtain integer solutions.

For an ILP to be feasible, values for $x_1, x_2, \ldots, x_n$ are found such that the solution satisfies every linear constraint and the integrality constraint. To find a feasible ILP solution, the problem is relaxed to a LP relaxation problem. LP relaxation temporarily drops the integrality constraint.

The optimal objective value of an ILP is always less than or equal to the optimal solution of a LP relaxation. This implies that the feasible region of an ILP is a subset of the LP feasible region. The optimal objective value of the LP relaxation is a real value and the optimal objective value of the ILP is an integer value. It is not possible for the ILP to yield a better optimal objective value than the optimal value of the LP relaxation; the ILP will yield the same value or a value less than the LP relaxation's optimal objective value.

Consider the following linear program [11] to illustrate the difference between an ILP and the LP relaxation:

$$\begin{aligned} &\text{maximize } x \\ &\quad \text{s.t. } 0.5 \leq x \leq 5.5 \\ &\qquad x \in \mathbb{Z} \end{aligned} \tag{4.42}$$

The corresponding feasible region is:

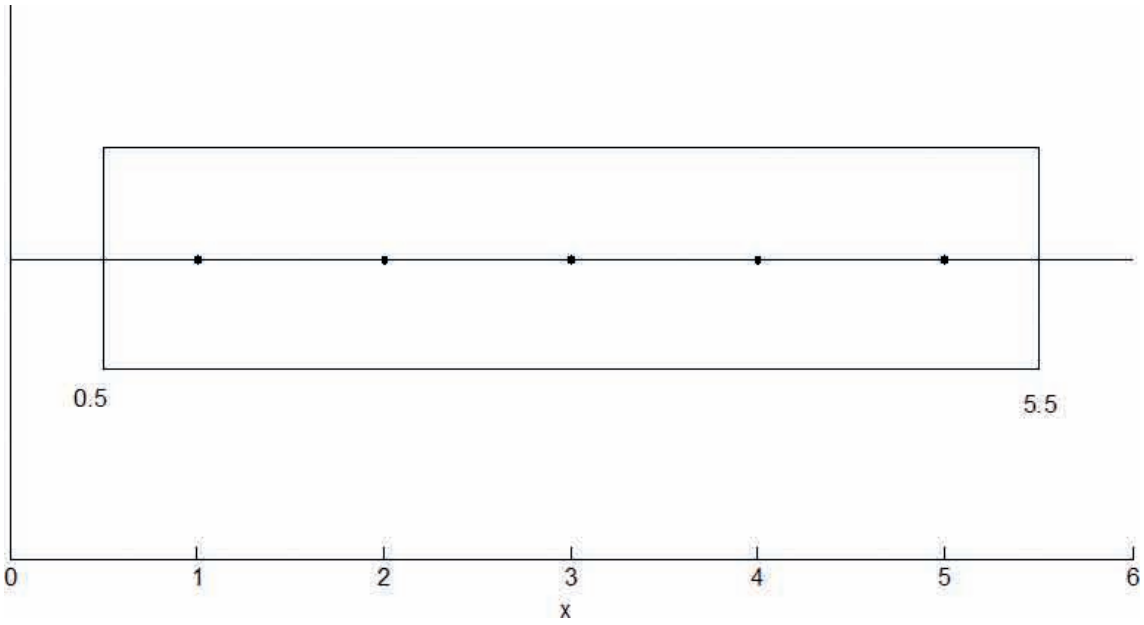


Figure 4.3: Geometry / feasible region

In the example, the real values for decision variable x can range from 0.5 up to 5.5. The only integer points in the feasible region are {1,2,3,4,5}. The feasible points for the ILP will be {1,2,3,4,5} while the feasible points for the LP relaxation are all real values in [0.5, 5.5]. Therefore, the optimal objective value for the ILP is $x = 5$ and the optimal objective value for the LP relaxation is $x = 5.5$.

There are multiple algorithms to solve ILPs. ILPs are NP-complete and algorithms tend to perform reasonable, however, each algorithm has exponential worst-case complexity [11]. Next subsections discuss branch and bound, cutting plane, and branch and cut algorithms to solve integer linear problems.

4.6.1 Branch and Bound

Branch and bound is an algorithm for solving ILPs by partitioning the feasible region. Partitioning is only required when the LP relaxation yields fractional values for its decision variables. The partition is made on the two nearest integer points for a fractional valued decision variable. The branch and bound algorithm grows a tree of LP relaxed problems in order to find the optimal ILP solution, having integer values for its decision variables. The root node of the tree resembles the optimal solution of the original LP relaxation problem given by the simplex algorithm. If a decision variable has a fractional solution, that decision variable is branched upon. Each branch operation yields two new child nodes, constructing a binary tree. Each node takes the parent's LP relaxation problem and adds an additional constraint. Let x_j be the fractional decision variable to branch on and let s_j be its corresponding value, one child node adds the constraint $x_j \leq \lfloor s_j \rfloor$ and the other child node adds the constraint $x_j \geq \lceil s_j \rceil$. Depending on the heuristic chosen, an unexplored node is solved and the process continues.

The optimal solution of each node, or problem, has one of possible three outcomes:

- Infeasible
- Integral solution
- Fractional solution

A LP relaxation problem that is not the original problem, or the problem represented by the root node, cannot be unbounded because the initial problem would have been unbounded; constraints are added to tighten the feasible region. When the node's problem is infeasible, branching can stop on that node and the algorithm can continue processing another unexplored node. Branching continues when a fractional value is found.

For nodes, problems, that yield integer optimal objective values are not necessary optimal in the entire tree. A global variable keeps track of the best integer optimal value found, $bestObjective := max(foundOptimum, bestObjective)$, where the initial value of $bestObjective := -\infty$.

For each unexplored node, the corresponding LP relaxation problem is solved by the simplex algorithm. The yielded optimal objective value is compared to the $bestObjective$ variable. When the optimal objective value of the LP relaxation problem is lower than the value of $bestObjective$, no branching is required because the particular branch cannot yield better values than a previously discovered optimum. This is the bound part of the branch and bound algorithm and is called pruning.

The overall goal is to minimize the number of explored nodes in a branch and bound tree. Often, there are multiple unexplored nodes to explore at the same time. Unexplored nodes that yield an integral solution or improve the value of $bestObjective$ are preferred. Several heuristics are available to try to achieve the desired behavior. There is no golden heuristic, a heuristic might perform excellent in one problem domain but not in the other. Several heuristics have to be experimented with in a certain problem domain in order to decide which heuristic to be used for the problems within that domain. Among others; depth-first, best LP relaxation solution and breadth-first are heuristics that can be used.

The depth-first node heuristic prefers unexplored nodes that have the largest depth in the tree. Breadth-first prefers unexplored nodes that have the least depth in the

tree, examining one level at a time. The best LP relaxation solution heuristic prefers to branch on nodes that have yielded the best LP relaxed objective value.

Once an unexplored node is chosen, a decision variable has to be chosen to branch on. When there is only one fractional value to choose from, the choice is simple. Heuristics are available to choose from multiple fractional valued decision variables. A possible heuristic is to choose the decision variable which value is closest to an integer value. Another heuristic might be to prefer binary decision variables because there are only two possible values. A random choice can also be considered as a possible heuristic. Like the unexplored node heuristics, there is no golden heuristic in choosing the decision variable to branch on.

Branching yields two new problems, a single constraint is added to the problem at every level of the tree. The problem's dictionary is updated accordingly but becomes infeasible. The new problem is infeasible because the fractional point in the feasible region is in between two integer points, the two branching constraints, and does not lie in the feasible region of the new problem. The initialization phase simplex might be performed on the infeasible problem. However, the infeasible problem is dual feasible and non-final. Performing simplex on its dual problem yields a final dual solution which can be converted back to a primal form obtaining the final, optimal, primal solution.

Potentially many constraints are added to the problem's dictionary during branch and bound. As an alternative to the standard form, the general form [16] avoids adding a new constraint to the dictionary. In general form, variable bounds are not included in the dictionary but are treated special and the pivoting operation is modified to respect these bounds. In general form, instead of adding constraints, the variable bounds are adjusted.

Consider the following ILP example to demonstrate branch and bound [11]:

$$
\begin{aligned}
\text{maximize} \quad & -x_1 - 2x_2 - 0.5x_3 - 0.2x_4 - x_5 + 0.5x_6 \\
s.t. \quad & x_1 + 2x_2 \geq 1 \\
& x_1 + x_2 + 3x_6 \geq 1 \\
& x_1 + x_2 + x_6 \geq 1 \\
& x_3 - 3x_4 \geq 1 \\
& x_3 - 2x_4 - 5x_5 \geq 1 \\
& x_4 + 3x_5 - 4x_6 \geq 1 \\
& x_2 + x_5 + x_6 \geq 1 \\
& x_1, x_2, x_3, x_4, x_5, x_6 \in \mathbb{Z}
\end{aligned}
\tag{4.43}
$$

The problem is not in standard form but the problem nicely demonstrates the use of branch and bound and heuristics. With arbitrary choice of decision variable branching, the following tree has been yielded:

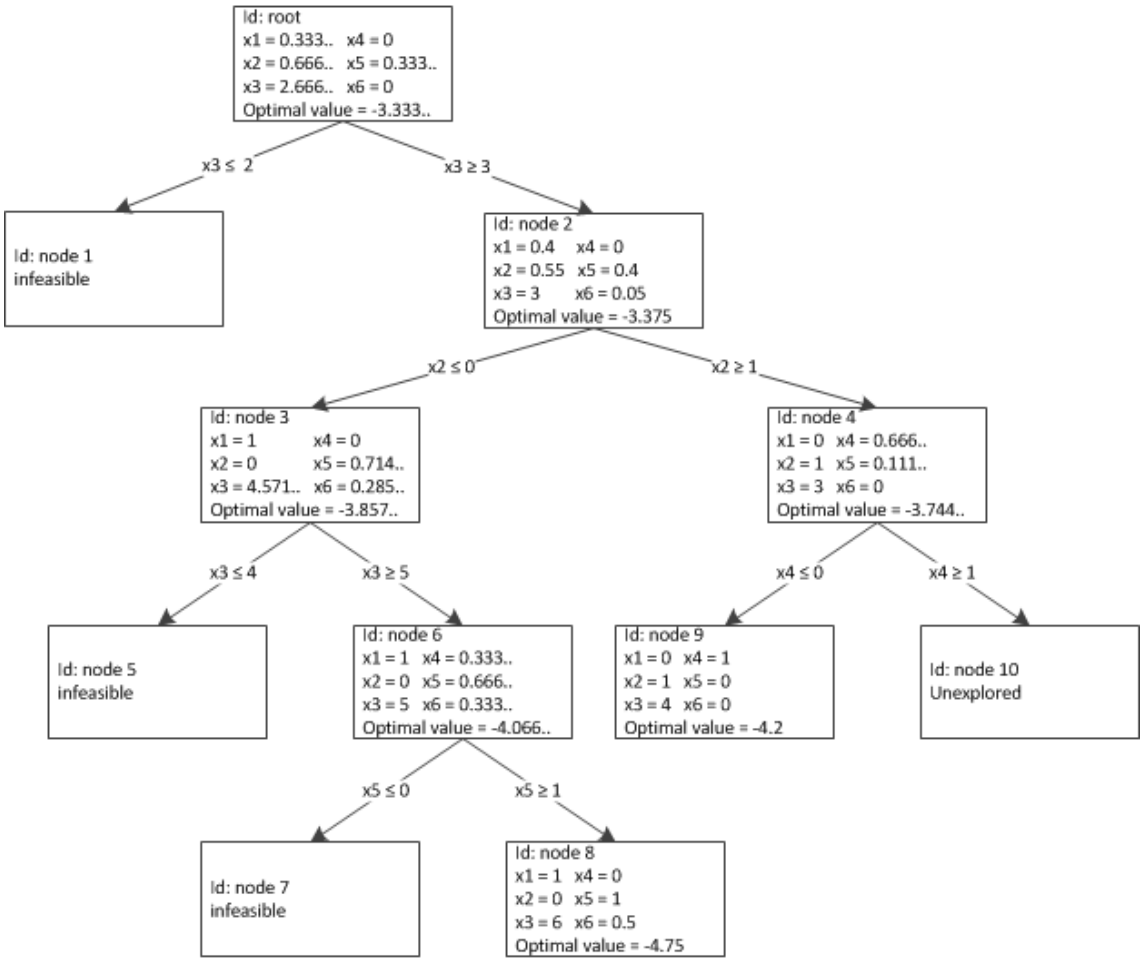


Figure 4.4: Problem (4.43)'s branch and bound tree

Each node of the branch and bound tree contains all decision variables with its value and the optimal objective value yielded by solving the corresponding problem. The root node contains the solution of the initial LP relaxation problem. The root node's possible branch variables are $\{x_1, x_2, x_3, x_5\}$ since x_4 and x_6 are already integral. In this case, x_3 is arbitrarily chosen. Assume that exploring node 3 was preferred over node 4. While branching continues after node 3, no integer solution was found in that particular branch up to node 7 and 8. Node 4's branch does yield an integer solution in node 9. In this case, the breath-first heuristic would have yielded an integer solution earlier than the depth-first heuristic. The optimal value of node 9 would have been set to $bestObjective$ and node 8 would have been pruned away because the optimal value of node 8 is -4.75 which is less than the best ILP solution found by node 9.

Consider branching on a different decision variable right at the start of branch and bound, at the root node. Choosing x_5 instead of x_3 yields the optimal integer solution in node 1, as shown in image 4.5. Only three rounds of simplex were required to yield the solution to the problem. Node 2 is pruned away by not having enough potential to yield a better optimum.

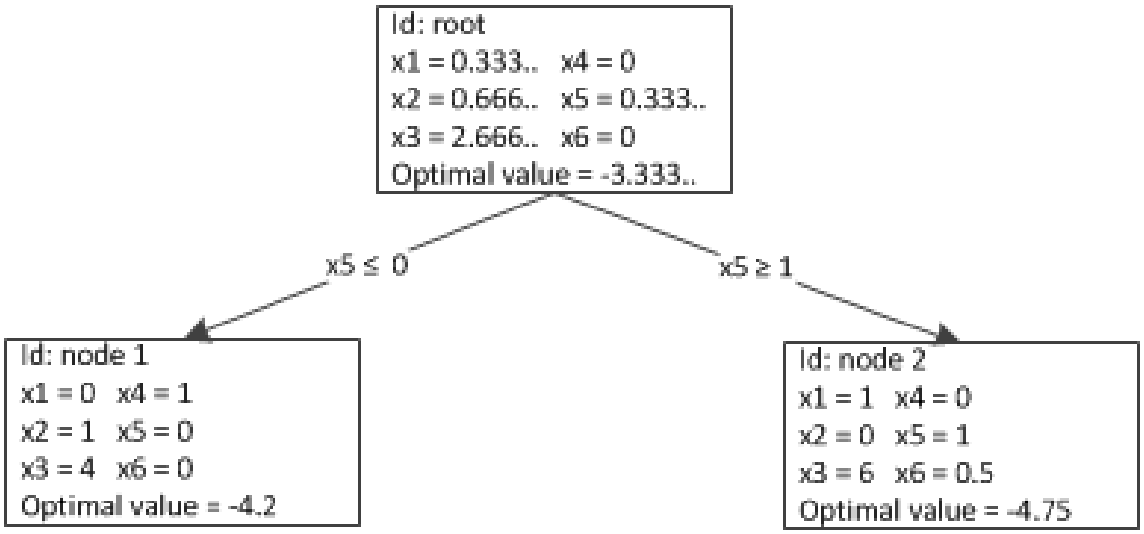


Figure 4.5: Branch and bound tree with a different starting branch variable for problem (4.43)

4.6.2 Cutting Planes

Unlike branch and bound, the cutting plane method cuts of portions of the feasible region. Like branch and bound, all integer points are preserved when cutting of portions of the feasible region. These cuts are Gomory-Chvâtal cuts [11, 15].

The LP relaxation of the problem is solved and checked whether the solution is integral. When the solution is integral, the problem has been solved. When the solution is not integral, a new cutting plane constraint is added to the problem. The process of adding new cutting plane constraints continues until the solution yields an integral solution.

The first step is to identify dictionary rows that have fractional values in vector b of the LP relaxed solution. All rows with fractional slack are candidates for a cutting plane constraint. Each coefficient in the constraint is split up into an integer and fractional part. The fractional part forms the basis for the cutting plane constraint.

Based on the definition (4.20), assume $b_k \notin \mathbb{Z}$ in dictionary row:

$$x_{Bk} = b_k + a_{k1}x_{I1} + \ldots + a_{kj}x_{Ij} + \ldots + a_{kn}x_{In}$$

The cutting plane becomes:

$$\text{frac}(-a_{k1})x_{I1} + \ldots + \text{frac}(-a_{kn})x_{In} \geq \text{frac}(b_k) \tag{4.44}$$

where

$$\text{frac}(x) = x - \lfloor x \rfloor \tag{4.45}$$

The sign of matrix A's coefficients are inversed because the cutting plane constraint is in form $A\mathbf{x} \geq \mathbf{b}$.

The cutting plane constraint can be added to the dictionary resulting in an infeasible problem. The problem is primal infeasible but dual feasible and non-final. Switching between the primal and dual version of the dictionary overcomes the need of initialization. All possible cutting plane constraints can be added to the dictionary at each step.

Consider the following final LP relaxed dictionary [11] to demonstrate the process of adding cutting planes:

$$\begin{array}{rl} x_1 & = 1.2 - 3.1x_2 + 4.3x_3 - 0.5x_5 \\ x_4 & = 1 \quad - x_2 \quad + x_3 \quad - x_5 \\ x_6 & = 2.5 + 1.3x_2 - 2.1x_3 + x_5 \\ \hline z & = 1.7 - 1.2x_2 - 2.3x_3 - 2.1x_5 \end{array} \tag{4.46}$$

Candidate cutting planes are the constraints corresponding to basic decision variables x_1 and x_6 since their corresponding b values are fractional. For x_1 a cutting plane is derived as:

$$\begin{aligned} & x_1 = 1.2 - 3.1x2 + 4.3x_3 - 0.5x_5 \\ =& (A\mathbf{x} \geq \mathbf{b} \text{ form}) \\ & x_1 + 3.1x_2 - 4.3x_3 + 0.5x_5 = 1.2 \\ =& (\text{by}(4.44)) \\ & \text{frac}(3.1)x_2 + \text{frac}(-4.3)x_3 + \text{frac}(0.5)x_5 \geq \text{frac}(1.2) \\ =& (\text{by}(4.45)) \\ & (3.1 - \lfloor 3.1 \rfloor)x_2 + (-4.3 - \lfloor -4.3 \rfloor)x_3 + (0.5 - \lfloor 0.5 \rfloor)x_5 \geq (1.2 - \lfloor 1.2 \rfloor) \\ =& \\ & 0.1x_2 + 0.7x_3 + 0.5x_5 \geq 0.2 \end{aligned} \tag{4.47}$$

Adding the cutting plane constraint to the dictionary, the dictionary becomes:

$$\begin{array}{rl} x_1 & = 1.2 \quad - 3.1x_2 + 4.3x_3 - 0.5x_5 \\ x_4 & = 1 \quad\quad - x_2 \quad + x_3 \quad - x_5 \\ x_6 & = 2.5 \quad + 1.3x_2 - 2.1x_3 + x_5 \\ x_7 & = -0.2 + 0.1x_2 + 0.7x_3 + 0.5x_5 \\ \hline z & = 1.7 \quad - 1.2x_2 - 2.3x_3 - 2.1x_5 \end{array} \tag{4.48}$$

where x_7 is represents the added constraint. The dictionary is primal infeasible but dual feasible and non-final. Solving the corresponding dual problem yields a new primal solution on which the process can continue.

4.6.3 Branch and Cut

It is possible to mix branch and bound and cutting planes together, resulting in branch and cut [10]. Cutting plane constraints are added to tighten the LP relaxation after solving the initial LP relaxed problem. The process of adding constraints and solving the LP relaxation is done repetitively before each branch operation. The idea of tightening the LP relaxation is to obtain a better bounding to yield an integral solution faster and to obtain a minimal number of nodes in the tree. Branch and cut is almost used by all solvers [11].

Solving integer linear programming problems with branch and cut does not have a single implementation for all problem classes, heuristics differ. For example, not every added cut is valid for another sub-problem, it is a choice to lift these cuts [10]. There are various techniques available to optimize the algorithms. One technique is preprocessing; removing unnecessary constraints, identifying fixed variables and simplifying the problem in other ways [10].

5 Task Scheduling with LP

Given the problem statement in chapter 2, this section discusses linear programming as a general solution to task scheduling. Linear programming is a mathematical technique to maximize, or minimize, a given cost function while respecting a set of linear constraints.

This chapter discusses whether linear programming is a suitable technique for the scheduler function, defined by (2.5). The scheduler function requires a set of nodes and a set of tasks as its input. Sections 5.2 and 5.3 describe possible ways to obtain task and node variable values. The return value of the scheduler function is a topology which describes which task is scheduled on which node. Section 5.4 describes a possible linear programming model that can be used as a scheduler. Given the variations is section 2.5, section 5.6 discusses possible solutions with the linear programming model given in section 5.4.

5.1 Overview

The scheduling of tasks onto nodes is done by the scheduler component. The scheduler is responsible for the execution of the scheduler function. The scheduler accepts a set of tasks and nodes to transform its inputs into a topology, or scheduling.

A client component is added to each node to measure certain variables to reflect the available capacity at that point in time. Nodes register to the scheduler. The scheduler retrieves measurement variables from each node specifying the capacity of a node. Based on the scheduler's inputs, a topology is formed by solving a linear programming problem and the resulting topology is distributed among registered nodes. The nodes execute tasks with respect to the received topology. The process of retrieving information and scheduling is a continuous process.

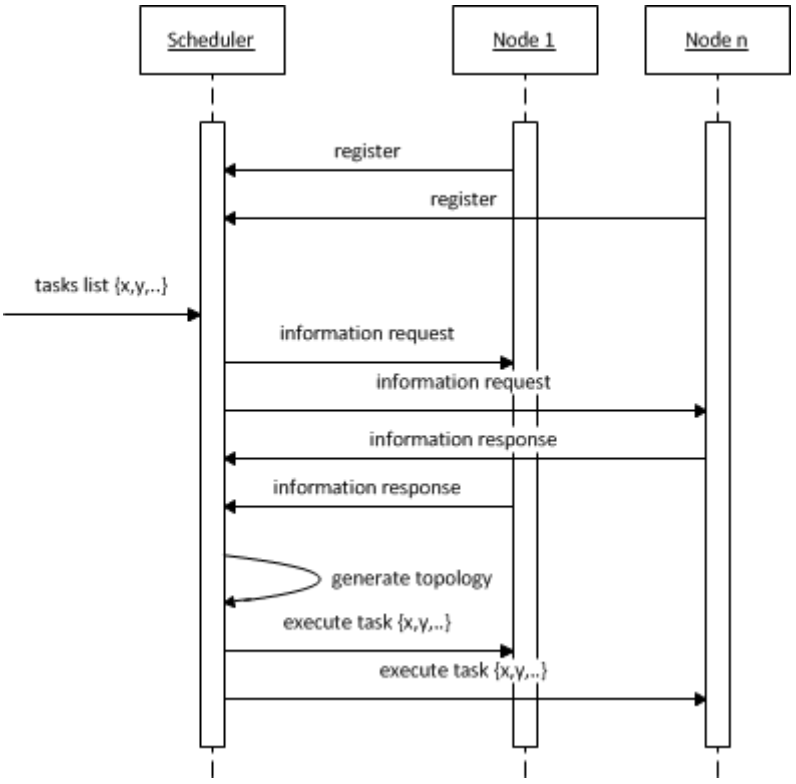


Figure 5.1: Scheduling process

Figure 5.1 depicts a coarse-grained interaction between the nodes and the scheduler. A node registers itself with the scheduler to be considered in a scheduling. Once registered, the scheduler continuously requests variable information from the node. The node sends back the requested information. The scheduler receives a list of tasks, solves the linear programming problem and produces a scheduling. When the node is included in the scheduling, the task is send to the node for execution. A node is unregistered just before termination of the client application. Only registered nodes are considered in scheduling. Algorithm 2 describes the coarse-grained process of a node's client application in pseudocode.

Algorithm 2: Node's client application

```
1  register node with scheduler;
2  while True do
3      if information request is received then
4          V ← set of measured variables;
5          send V to scheduler;
6      end
7      if execution request is received then
8          T ← received topology from scheduler;
9          execute each task in T;
10     end
11 end
12 unregister node;
```

The node's client application measures the available variables when the scheduler requests them. When all variables are measured, the set of measurements are send back to the scheduler. The scheduler decides whether or not the node is included

in the scheduling by solving the linear programming problem. When the node is included in the scheduling, the set of tasks is received by the node for execution. A node can only be registered to a single scheduler. Multiple schedulers might incur overscheduling of tasks to a single node since all schedulers are informed of the same available capacity.

The scheduler is responsible for managing nodes, receiving tasks and producing a topology. Algorithm 3 describes the coarse-grained process of the scheduler in pseudocode.

Algorithm 3: Scheduler

```
1  let N be a list of available, registered, nodes;
2  let S be a set of tasks;
3  while True do
4      if node wants to register then
5          add node to N;
6      end
7      if node want to unregister then
8          remove node from N;
9      end
10     if set of tasks is received then
11         add each task to S;
12     end
13     T ← generate topology using linear programming, scheduler function (N, S);
14     send execute request to each node in topology T;
15 end
```

The information request also acts as a heartbeat mechanism to the scheduler. A heartbeat mechanism ensures that a node is responsive and ready to process tasks. When a node becomes unresponsive, that node can be unregistered for efficiency. A topology will be suboptimal when unresponsive nodes are included in a scheduling because tasks are scheduled to a node that will not be executed.

The scheduler sends task execution requests to a node and the node executes these tasks. After a task has been executed, the node sends the scheduler a confirmation of this fact. The confirmation is important because it cannot be guaranteed that a task is executed, it is possible for a node to crash leaving tasks unexecuted. The scheduler can reschedule unexecuted tasks after a certain timeout period or when the node is unresponsive to information requests.

5.2 Obtaining task requirements

Predicting required task resources is an active field of research. There are models available to predict required node resources based on historical data [9]. These models focus heavily on the total required resources of a node. However, with process accounting [3] and other monitoring tools, it is possible to obtain task/process specific measurements to generate historical data for a possible prediction model.

Prediction models based on historical data are suitable when tasks are executed repetitively. However, some tasks do not have such information, i.e. a specific task has not been executed before. Further research is needed to obtain information

about the resource requirements for these types of tasks. Although it is not in the scope of this document, the following possibilities can be thought of to obtain the required information:

- system administrator knowledge;
- task developer knowledge;
- task input knowledge;
- task output knowledge.

The system administrator may have insights in similar tasks. An educated guess can be made by the system administrator to obtain task requirements based on past experiences. The person, or persons, responsible for task development, testing and execution may provide additional insights in the task's requirements. Also, the testing phase, development testing or acceptance testing can yield requirement measurements. It is possible to obtain historical data with every executed test. Most tasks are tested before they are put in production environments, therefor the test suite can be modified to obtain measurements.

It is also possible to look at the inputs and outputs of a specific task to get a sense of the required resources. Depending on task design, a data set is usually a task input which is streamed to the node and stored on disk or read directly into memory. When the data set is read directly into memory, it is possible to say that a portion of required memory is equal to its corresponding input data set. Similar can be said about task outputs; input aggregation tasks yield a preformatted output where the size of the intermediate results and the end results are known.

5.3 Obtaining Node Variables

Obtaining the current operating state of a node can be done by the monitoring application that is installed on the node. The monitoring application is able to monitor variable changes in real-time. The measured variables can be used by the scheduler. However, every variable corresponding to the required task variables has to be measured for comparison.

5.4 Linear Programming Model

Typically, there are a number of tasks to be scheduled to a number of nodes. The number of tasks is denoted by n while the number of nodes is denoted by m. The objective of the problem is to maximize the number of tasks scheduled to nodes. The decision variables are defined as binary variables with values of zero or one; that is, $x_{ij} \in \{0, 1\}$. When $x_{ij} = 1$, it indicates the assignment of task i to node j. Otherwise, when $x_{ij} = 0$, task i is not assigned to node j. There are $n \cdot m$ decision variables to form a topology. The mapping between decision variables, tasks and nodes is shown in table 5.1.

Decision variable	Corresponding task	Corresponding node
x_{11}	task 1	node 1
x_{12}	task 1	node 2
	$\vdots$	
x_{1m}	task 1	node m
x_{21}	task 2	node 1
x_{21}	task 2	node 2
	$\vdots$	
x_{2m}	task 2	node m
	$\vdots$	
x_{nm}	task n	node m

Table 5.1: Decision variable mapping

The objective function of the model is:

$$\text{maximize} \sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij} \tag{5.1}$$

Summing each decision variable's value yields the number of scheduled tasks. It is only possible to schedule a single task once, to a single node. A task's corresponding decision variables have to be constrained. Multiple decision variables for a single task implies that at most one decision variable has value one while other variables have value zero. All decision variable value corresponding to a single task sum up to zero when the task is not scheduled. The constraint is defined as:

$$\sum_{j=1}^{m} x_{ij} \leq 1 \qquad (i = 1, \ldots, n) \tag{5.2}$$

To respect a node's maximum variable capacity, constraints have to be added to ensure that no capacity is exceeded while scheduling tasks to nodes. For every required task variable there has to be a corresponding node capacity variable. Since multiple tasks can be scheduled to a single node, all required values of allocated tasks have to be taken into account to not exceed the remaining node's capacity. The constraint for the v-th variable is defined as:

$$\left.\begin{array}{c} R_1^v x_{11} + R_2^v x_{21} + \ldots + R_i^v x_{i1} + \ldots + R_n^v x_{n1} \leq NCAP_1^v \\ R_1^v x_{12} + R_2^v x_{22} + \ldots + R_i^v x_{i2} + \ldots + R_n^v x_{n2} \leq NCAP_2^v \\ \vdots \\ R_1^v x_{1j} + R_2^v x_{2j} + \ldots + R_i^v x_{ij} + \ldots + R_n^v x_{nj} \leq NCAP_j^v \\ \vdots \\ R_1^v x_{1m} + R_2^v x_{2j} + \ldots + R_i^v x_{im} + \ldots + R_n^v x_{nm} \leq NCAP_m^v \end{array}\right\} 1, \ldots, v, \ldots, k \text{ variables} \tag{5.3}$$

The required capacity must be less than or equal to the remaining capacity of node j for variable v. The required capacity for task i is R_i^v for variable v. There are k

variables and $k \cdot m$ capacity constraints. The $NCAP$ variable represents the remaining capacity for variable v on node j. $NCAP$ is defined as:

$$NCap_j^v = capacity_j^v - used_j^v \tag{5.4}$$

Where $capacity$ is node j's total, catalog, capacity for the v-th variable and $used$ is the amount of resources currently in use for the v-th variable on node j.

Matrix R defines the v-variable cost of scheduling task i. There are k variables for each task i. R is defined as:

$$R = \begin{pmatrix} cost_1^1 & cost_1^2 & \dots & cost_1^v & \dots & cost_1^k \\ cost_2^1 & cost_2^2 & \dots & cost_2^v & \dots & cost_2^k \\ & & \vdots & & & \\ cost_i^1 & cost_i^2 & \dots & cost_i^v & \dots & cost_i^k \\ & & \vdots & & & \\ cost_n^1 & cost_n^2 & \dots & cost_n^v & \dots & cost_n^k \end{pmatrix} \tag{5.5}$$

A problem where all decision variables are binary is often referred to as a binary integer linear programming problem. The decision variables are bounded by:

$$x_{ij} \in \{0, 1\} \quad (i = 1 \dots n, j = 1 \dots m) \tag{5.6}$$

To solve an integer linear program, the problem has to be relaxed and solved by an integer linear programming algorithm as described in section 4.6. The constraints on the decision variables have to be relaxed to obtain the LP relaxed problem, as shown in (5.7). In other words, the decision variables accept real values between zero and one; that is, $0 \leq x_{ij} \leq 1$. The relaxed decision variables constraints are made explicit by (5.7). However, the decision variables are implicitly constrained by (5.2).

$$x_{ij} \in [0, 1] \quad (i = 1 \dots n, j = 1 \dots m) \tag{5.7}$$

The following subsections illustrate the modelling on problems with arbitrary chosen number of nodes, number of tasks, and capacity values. The example problems are solved by GLPK [8].GLPK uses simplex to produce the optimal solution. The following examples are discussed:

# tasks	# nodes	# variables	Section	Page number
3	2	1	5.4.1	56
4	2	1	5.4.2	59
3	2	2	5.4.3	60

5.4.1 Example: 3 tasks, 2 nodes, 1 variable

Consider an example of three tasks, two nodes and a single capacity variable. Table 5.2 describes the tasks' capacity requirements and nodes' available capacity.

Task	Capacity required
	Variable 1
Task 1	200
Task 2	500
Task 3	300
Node	Capacity
	Variable 1
Node 1	500
Node 2	600

Table 5.2: Tasks and Nodes

The decision variables, with their corresponding mapping of tasks and nodes, are listed in table 5.3.

Decision variable	Corresponding task	Corresponding node
x_{11}	task 1	node 1
x_{12}	task 1	node 2
x_{21}	task 2	node 1
x_{22}	task 2	node 2
x_{31}	task 3	node 1
x_{32}	task 3	node 2

Table 5.3: Decision variable mapping

Based on the information given in table 5.2 and table 5.3, it is possible to formulate the primal linear program, by definition (4.7). With the given information, the linear program terms are as follows:

$$
\begin{gathered}
R = \begin{bmatrix} 200 \\ 500 \\ 300 \end{bmatrix} \mathbf{NCAP} = \begin{bmatrix} 500 \\ 600 \end{bmatrix} \\
\mathbf{c}^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \\
\mathbf{x} = \begin{bmatrix} x_{11} & x_{12} & x_{21} & x_{22} & x_{31} & x_{32} \end{bmatrix} \\
A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 200 & 0 & 500 & 0 & 300 & 0 \\ 0 & 200 & 0 & 500 & 0 & 300 \end{bmatrix} \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 500 \\ 600 \end{bmatrix}
\end{gathered}
\tag{5.8}
$$

Matrix R and vector $\mathbf{NCAP}$ are used to form the capacity constraints, by definition (5.3). The objective function is to maximize the number of scheduled tasks, by definition (5.1):

$$\text{maximize } x_{11} + x_{12} + x_{21} + x_{22} + x_{31} + x_{32} \tag{5.9}$$

The constraints to ensure that each task is scheduled at most once are, by definition (5.2):

$$\begin{aligned} x_{11} + x_{12} &\leq 1 \\ x_{21} + x_{22} &\leq 1 \\ x_{31} + x_{32} &\leq 1 \end{aligned} \tag{5.10}$$

The capacity constraints for node 1 and node 2 are, by definition (5.3):

$$\begin{aligned} 200x_{11} + 500x_{21} + 300x_{31} &\leq 500 \\ 200x_{12} + 500x_{22} + 300x_{32} &\leq 600 \end{aligned} \tag{5.11}$$

The decision variable constraints are, by (5.6):

$$x_{11}, x_{12}, x_{21}, x_{22}, x_{31}, x_{32} \in \{0, 1\} \tag{5.12}$$

The full binary integer linear programming problem model is:

$$\begin{array}{rrrrrrrrl} \text{maximize} & x_{11} & +x_{12} & +x_{21} & +x_{22} & +x_{31} & +x_{32} & & \\ \text{s.t.} & x_{11} & +x_{12} & & & & & \leq & 1 \\ & & & x_{21} & +x_{22} & & & \leq & 1 \\ & & & & & x_{31} & +x_{32} & \leq & 1 \\ & 200x_{11} & & +500x_{21} & & +300x_{31} & & \leq & 500 \\ & & 200x_{12} & & +500x_{22} & & +300x_{32} & \leq & 600 \\ & x_{11}, & x_{12}, & x_{21}, & x_{22}, & x_{31}, & x_{32} & \in & \{0, 1\} \end{array} \tag{5.13}$$

To solve the linear programming problem with an integer linear programming algorithm, the LP relaxation is used as described in section 4.6. The full LP relaxed problem is:

$$\begin{array}{rrrrrrrrl} \text{maximize} & x_{11} & +x_{12} & +x_{21} & +x_{22} & +x_{31} & +x_{32} & & \\ \text{s.t.} & x_{11} & +x_{12} & & & & & \leq & 1 \\ & & & x_{21} & +x_{22} & & & \leq & 1 \\ & & & & & x_{31} & +x_{32} & \leq & 1 \\ & 200x_{11} & & +500x_{21} & & +300x_{31} & & \leq & 500 \\ & & 200x_{12} & & +500x_{22} & & +300x_{32} & \leq & 600 \\ & x_{11}, & x_{12}, & x_{21}, & x_{22}, & x_{31}, & x_{32} & \in & [0, 1] \end{array} \tag{5.14}$$

Solving the relaxed linear programming problem yields the following results:

Decision variable	Corresponding task	Corresponding node
$x_{11} = 1$	task 1	node 1
$x_{12} = 0$	task 1	node 2
$x_{21} = 0$	task 2	node 1
$x_{22} = 1$	task 2	node 2
$x_{31} = 1$	task 3	node 1
$x_{32} = 0$	task 3	node 2

In this particular case, the solution to the LP relaxation problem is equal to the ILP solution since every decision variable's value is integral. The topology is as follows: node 1 executes tasks 1 and 3 while node 2 executes task 2. The capacity required on both nodes is:

$$
\begin{aligned}
& 200x_{11} + 500x_{21} + 300x_{31} && \leq 500 \\
& 200x_{21} + 500x_{22} + 300x_{32} && \leq 600 \\
= & && \\
& 200 \cdot 1 + 500 \cdot 0 + 300 \cdot 1 && = 500 \\
& 200 \cdot 0 + 500 \cdot 1 + 300 \cdot 0 && = 500
\end{aligned}
\tag{5.15}
$$

5.4.2 Example: 4 tasks, 2 nodes, 1 variable

Consider an example of four tasks, two nodes and a single variable. Table 5.4 describes the tasks' capacity requirements and nodes' available capacity.

Task	Capacity required
	Variable 1
Task 1	200
Task 2	500
Task 3	300
Task 4	700
Node	Capacity
	Variable 1
Node 1	500
Node 2	600

Table 5.4: Tasks and Nodes

Based on the information given in table 5.4, it is possible to formulate the primal linear program, by definition (4.7). Clearly task 4 cannot be scheduled since there is no node with sufficient capacity to execute task 4. With the given information, the linear program terms are as follows:

$$
\begin{gathered}
R = \begin{bmatrix} 200 \\ 500 \\ 300 \\ 700 \end{bmatrix} \mathbf{NCAP} = \begin{bmatrix} 500 \\ 600 \end{bmatrix} \\
\mathbf{c}^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \\
\mathbf{x} = \begin{bmatrix} x_{11} & x_{12} & x_{21} & x_{22} & x_{31} & x_{32} & x_{41} & x_{42} \end{bmatrix} \\
A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 200 & 0 & 500 & 0 & 300 & 0 & 700 & 0 \\ 0 & 200 & 0 & 500 & 0 & 300 & 0 & 700 \end{bmatrix} \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 500 \\ 600 \end{bmatrix}
\end{gathered}
\tag{5.16}
$$

Matrix R and vector $\mathbf{NCAP}$ are used to form the capacity constraints, by definition (5.3). The full binary linear programming problem is:

$$
\begin{array}{rrrrrrrrrrl}
\text{maximize} & x_{11} & +x_{12} & +x_{21} & +x_{22} & +x_{31} & +x_{32} & +x_{41} & +x_{42} & & \\
\text{s.t.} & x_{11} & +x_{12} & & & & & & & & \leq 1 \\
& & & x_{21} & +x_{22} & & & & & & \leq 1 \\
& & & & & x_{31} & +x_{32} & & & & \leq 1 \\
& & & & & & & x_{41} & +x_{42} & & \leq 1 \\
& 200x_{11} & & +500x_{21} & & +300x_{31} & & +700x_{41} & & & \leq 500 \\
& & 200x_{12} & & +500x_{22} & & +300x_{32} & & +700x_{42} & & \leq 600 \\
& x_{11}, & x_{12}, & x_{21}, & x_{22}, & x_{31}, & x_{32}, & x_{41}, & x_{42} & & \in \{0,1\}
\end{array}
$$

(5.17)

The relaxed LP problem is obtained by relaxing the decision variable constraints; that is, $x_{ij} \in [0, 1]$. The solution to the relaxed LP problem is:

Decision variable	Corresponding task	Corresponding node
$x_{11} = 1$	task 1	node 1
$x_{12} = 0$	task 1	node 2
$x_{21} = 0$	task 2	node 1
$x_{22} = 1$	task 2	node 2
$x_{31} = 1$	task 3	node 1
$x_{32} = 0$	task 3	node 2
$x_{41} = 0$	task 4	node 1
$x_{42} = 0.142857$	task 4	node 2

The yielded objective value is: 3.142857. The solution is fractional due to x_{42}'s value. That x_{42} has a fractional value is because node 2 has some residual capacity when the other tasks are scheduled, a fraction of task 4 can be scheduled. x_{42} yields to 0 after an ILP algorithm has further processed the problem.

5.4.3 Example: 3 tasks, 2 nodes, 2 variables

Consider an example of four tasks, two nodes and a single variable. Table 5.5 describes the tasks' capacity requirements and nodes' available capacity.

Task	Capacity required	
	Variable 1	Variable 2
Task 1	200	100
Task 2	500	300
Task 3	300	200
Node	Capacity	
	Variable 1	Variable 2
Node 1	500	300
Node 2	600	200

Table 5.5: Tasks and Nodes

Based on the information given in table 5.5, it is possible to formulate the primal linear program, by definition (4.7). With the given information, the linear program terms are as follows:

$$
\begin{gathered}
R = \begin{bmatrix} 200 & 100 \\ 500 & 300 \\ 300 & 200 \end{bmatrix} \mathbf{NCAP} = \begin{bmatrix} 500 & 300 \\ 600 & 200 \end{bmatrix} \\
\mathbf{c}^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \\
\mathbf{x} = \begin{bmatrix} x_{11} & x_{12} & x_{21} & x_{22} & x_{31} & x_{32} \end{bmatrix} \\
A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 200 & 0 & 500 & 0 & 300 & 0 \\ 0 & 200 & 0 & 500 & 0 & 300 \\ 100 & 0 & 300 & 0 & 200 & 0 \\ 0 & 100 & 0 & 300 & 0 & 200 \end{bmatrix} \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 500 \\ 600 \\ 300 \\ 200 \end{bmatrix}
\end{gathered}
\tag{5.18}
$$

Matrix R and vector $\mathbf{NCAP}$ are used to form the capacity constraints, by definition (5.3). The full binary linear programming problem is:

$$
\begin{aligned}
\text{maximize} \quad & x_{11} + x_{12} + x_{21} + x_{22} + x_{31} + x_{32} \\
\text{s.t.} \quad & x_{11} + x_{12} \leq 1 \\
& x_{21} + x_{22} \leq 1 \\
& x_{31} + x_{32} \leq 1 \\
& 200x_{11} + 500x_{21} + 300x_{31} \leq 500 \\
& 200x_{12} + 500x_{22} + 300x_{32} \leq 600 \\
& 100x_{11} + 300x_{21} + 200x_{31} \leq 300 \\
& 100x_{12} + 300x_{22} + 200x_{32} \leq 200 \\
& x_{11}, x_{12}, x_{21}, x_{22}, x_{31}, x_{32} \in \{0, 1\}
\end{aligned}
\tag{5.19}
$$

The relaxed LP problem is obtained by relaxing the decision variable constraints; that is, $x_{ij} \in [0, 1]$. The solution to the relaxed LP problem is:

Decision variable	Corresponding task	Corresponding node
$x_{11} = 1$	task 1	node 1
$x_{12} = 0$	task 1	node 2
$x_{21} = 0$	task 2	node 1
$x_{22} = 0.666667$	task 2	node 2
$x_{31} = 1$	task 3	node 1
$x_{32} = 0$	task 3	node 2

The yielded objective value is: 2.666667. After an ILP algorithm has further processed the problem, the ILP result is:

Decision variable	Corresponding task	Corresponding node
$x_{11} = 0$	task 1	node 1
$x_{12} = 0$	task 1	node 2
$x_{21} = 1$	task 2	node 1
$x_{22} = 0$	task 2	node 2
$x_{31} = 0$	task 3	node 1
$x_{32} = 1$	task 3	node 2

The yielded objective value is 2. Remarkable to state is that task 2 is scheduled partly on node 2 in the LP relaxation to yield a better objective value. However, node 2 has insufficient capacity for the second variable to execute task 2 and therefor task 2 is scheduled to node 1 at expense of task 1 during ILP processing.

5.5 Dual Linear Programming Model

Every primal relaxed linear programming problem has an equivalent dual problem to find upper bounds to the primal problem, as described in section 4.5. By definition (4.37), the dual version of the primal problem is defined as:

$$\begin{aligned} \text{minimize} \quad & \mathbf{b}^T\mathbf{y} \\ \text{s.t.} \quad & A^T\mathbf{y} \geq \mathbf{c} \\ & \mathbf{y} \geq 0 \end{aligned} \tag{4.37}$$

The primal's corresponding dual linear program terms are:

$$\mathbf{b}^T = \begin{pmatrix} \mathbf{1}_n^T & \overbrace{\mathbf{NCAP}_1^v \dots \mathbf{NCAP}_m^v}^{1,\dots,v,\dots,k} \end{pmatrix}$$

$$\mathbf{y} = \begin{pmatrix} y_1 \dots y_{n+km} \end{pmatrix}$$

$$\mathbf{c} = \mathbf{1}_{nm}$$

$$A^T = \begin{pmatrix} 1 & 0 & 0 & \overbrace{R_1^v}^{1,\dots,v,\dots,k} & 0 & & 0 \\ 1 & 0 & 0 & 0 & R_1^v & & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & 0 & 0 & & R_1^v \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 1 & 0 & R_i^v & 0 & & 0 \\ 0 & \cdots & 1 & \cdots & 0 & 0 & R_i^v & & 0 \\ \vdots & & \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & 0 & 0 & 0 & & R_i^v \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 1 & R_n^v & 0 & & 0 \\ 0 & 0 & 1 & 0 & R_n^v & & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & 0 & 0 & & R_n^v \end{pmatrix} \tag{5.20}$$

The $\mathbf{b}^T$ vector consists of n ones due to n constraints in the primal problem to ensure that a task is only scheduled once, by definition (5.2). The vector $\mathbf{b}^T$ also consists of km $\mathbf{NCAP}$-values, where k is the number of task/node variables, due to the nodes' capacity constraints, by definition (5.3). The vector $\mathbf{b}^T$ has $n + km$ values, or coefficients, which correspond to $n + km$ dual decision variables, i.e. the vector $\mathbf{y}$. The column vector $\mathbf{c}$ consists of the primal's objective function coefficients, which consists of nm ones.

The A^T matrix is a $nm \times n + km$ matrix, mentally, composed of two blocks. The first block is a sparse matrix of size $nm \times n$ with ones around the diagonal. There are m consecutive ones in each column. The first block forces the dual problem to consider a single task in each constraint. The first block is the transposed version of definition (5.2) to ensure that a task is scheduled at most once. The second block is a $nm \times km$ matrix representing the tasks' variable requirements and is the transposed version of definition (5.3) to ensure the capacity requirements.

The objective function is, in matrix notation:

$$\text{minimize } \mathbf{1}_{1,\dots,n}\mathbf{y}_{1,\dots,n}, \overbrace{\mathbf{NCAP}_1^v \dots \mathbf{NCAP}_m^v}^{1,\dots,v,\dots,k}\mathbf{y}_{n+1,\dots,km} \tag{5.21}$$

The dual constraints have the following form:

$$
\begin{array}{ccccccccccc}
 & & & & & \overbrace{\hphantom{R_1^v y_{n+1} \quad 0y_{n+2} \quad \ddots \quad 0y_{n+m}}}^{1,\ldots,v,\ldots,k} & & & & \\
1y_1 & & 0y_i & & 0y_n & R_1^v y_{n+1} & 0y_{n+2} & & 0y_{n+m} & \geq 1 \\
1y_1 & & 0y_i & & 0y_n & 0y_{n+1} & R_1^v y_{n+2} & & 0y_{n+m} & \geq 1 \\
\vdots & & \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
1y_1 & & 0y_i & & 0y_n & 0y_{n+1} & 0y_{n+2} & & R_1^v y_{n+m} & \geq 1 \\
\vdots & & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0y_1 & \ldots & 1y_i & \ldots & 0y_n & R_i^v y_{n+1} & 0y_{n+2} & & 0y_{n+m} & \geq 1 \\
0y_1 & & 1y_i & & 0y_n & 0y_{n+1} & R_i^v y_{n+2} & & R_i^v y_{n+m} & \geq 1 \\
\vdots & & \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0y_1 & & 0y_i & & 1y_n & R_n^v y_{n+1} & 0y_{n+2} & & 0y_{n+m} & \geq 1 \\
0y_1 & & 0y_i & & 1y_n & 0y_{n+1} & R_n^v y_{n+2} & & 0y_{n+m} & \geq 1 \\
\vdots & & \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0y_1 & & 0y_i & & 1y_n & 0y_{n+1} & 0y_{n+2} & & R_n^v y_{n+m} & \geq 1
\end{array}
\tag{5.22}
$$

The constraint on each dual decision variable is:

$$y_i \in [0,1] \quad (i = 1, \ldots, n + km) \tag{5.23}$$

The goal is to minimize the objective function while respecting that every inequality has to yield at least the value one. The dual LP relaxation problem is most likely to yield fractional values for most dual decision variables. Dual decision variables where the value equals zero indicates that the corresponding constraint is inactive in the primal problem. The dual problem cannot be a binary or an integer linear program, only the primal relaxed linear programming problem does. To illustrate that the dual decision variables cannot be binary, inspect the first block in matrix A^T. The dual binary problem is expected to yield ones in the first n dual decision variables because a one in the last km decision variables incurs a much higher objective value which is not optimal. Also, setting the value one in the first n decision variables will satisfy all constraints. A binary dual problem has no meaning and results in an optimal value equal to the number of tasks n which differs from the primal binary problem's optimal value.

The following subsections illustrate the dual versions of the primal example problems introduced in 5.4:

# tasks	# nodes	# variables	Section	Page number
3	2	1	5.5.1	65
4	2	1	5.5.2	66
3	2	2	5.5.3	67

5.5.1 Example: 3 tasks, 2 nodes, 1 variable

Consider the example illustrated in subsection 5.4.1 of three tasks, two nodes and a single variable. The corresponding dual linear programming problem terms are:

$$\begin{aligned} \mathbf{b}^T &= \begin{bmatrix} 1 & 1 & 1 & 500 & 600 \end{bmatrix} \\ \mathbf{y} &= \begin{bmatrix} y_1 & y_2 & y_3 & y_4 & y_5 \end{bmatrix} \\ A^T &= \begin{bmatrix} 1 & 0 & 0 & 200 & 0 \\ 1 & 0 & 0 & 0 & 200 \\ 0 & 1 & 0 & 500 & 0 \\ 0 & 1 & 0 & 0 & 500 \\ 0 & 0 & 1 & 300 & 0 \\ 0 & 0 & 1 & 0 & 300 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{aligned} \tag{5.24}$$

By definition (4.37), the dual equivalent of the primal's relaxed problem is formulated as:

$$\begin{aligned} \text{minimize} \quad & y_1 + y_2 + y_3 + 500y_4 + 600y_5 \\ \text{s.t.} \quad & y_1 + 200y_4 \geq 1 \\ & y_1 + 200y_5 \geq 1 \\ & y_2 + 500y_4 \geq 1 \\ & y_2 + 500y_5 \geq 1 \\ & y_3 + 300y_4 \geq 1 \\ & y_3 + 300y_5 \geq 1 \\ & y_1, \; y_2, \; y_3, \; y_4, \; y_5 \in [0, 1] \end{aligned} \tag{5.25}$$

Solving the dual version of the primal's LP relaxed problem yields:

Decision variable	value
y_1	1
y_2	1
y_3	1
y_4	0
y_5	0
y_6	0

The dual's objective value is 3 which is equal to the primal's relaxed objective value.

5.5.2 Example: 4 tasks, 2 nodes, 1 variable

Consider the example illustrated in subsection 5.4.2 of four tasks, two nodes and a single variable. The corresponding dual linear programming problem terms are:

$$
\begin{aligned}
\mathbf{b}^T &= \begin{bmatrix} 1 & 1 & 1 & 1 & 500 & 600 \end{bmatrix} \\
\mathbf{y} &= \begin{bmatrix} y_1 & y_2 & y_3 & y_4 & y_5 & y_6 \end{bmatrix} \\
A^T &= \begin{bmatrix} 1 & 0 & 0 & 0 & 200 & 0 \\ 1 & 0 & 0 & 0 & 0 & 200 \\ 0 & 1 & 0 & 0 & 500 & 0 \\ 0 & 1 & 0 & 0 & 0 & 500 \\ 0 & 0 & 1 & 0 & 300 & 0 \\ 0 & 0 & 1 & 0 & 0 & 300 \\ 0 & 0 & 0 & 1 & 700 & 0 \\ 0 & 0 & 0 & 1 & 0 & 700 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
\end{aligned}
\tag{5.26}
$$

By definition (4.37), the dual equivalent of the primal's relaxed problem is formulated as:

$$
\begin{aligned}
\text{minimize} \quad & y_1 + y_2 + y_3 + y_4 + 500y_5 + 600y_6 \\
\text{s.t.} \quad & y_1 + 200y_5 \geq 1 \\
& y_1 + 200y_6 \geq 1 \\
& y_2 + 500y_5 \geq 1 \\
& y_2 + 500y_6 \geq 1 \\
& y_3 + 300y_5 \geq 1 \\
& y_3 + 300y_6 \geq 1 \\
& y_4 + 700y_5 \geq 1 \\
& y_4 + 700y_6 \geq 1 \\
& y_1, \; y_2, \; y_3, \; y_4, \; y_5, \; y_6 \in [0, 1]
\end{aligned}
\tag{5.27}
$$

Solving the dual version of the primal's LP relaxed problem yields:

Decision variable	value
y_1	0.714285714285714
y_2	0.285714285714286
y_3	0.571428571428571
y_4	0
y_5	0.00142857142857143
y_6	0.00142857142857143

The objective value is 3.142857, by:

$$
\begin{aligned}
& y_1 + y_2 + y_3 + y_4 + 500y_5 + 600y_6 \\
= \; & 0.714285714285714 + 0.285714285714286 + 0.571428571428571 + 0 \\
& + 500 \cdot 0.00142857142857143 + 600 \cdot 0.00142857142857143 \\
= \; & 3.14285714285714
\end{aligned}
\tag{5.28}
$$

5.5.3 Example: 3 tasks, 2 nodes, 2 variables

Consider the example illustrated in subsection 5.4.3 of three tasks, two nodes and two variables. The corresponding dual linear programming problem terms are:

$$\mathbf{b}^T = \begin{bmatrix} 1 & 1 & 1 & 500 & 600 & 300 & 200 \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 0 & 0 & 200 & 0 & 100 & 0 \\ 1 & 0 & 0 & 0 & 200 & 0 & 100 \\ 0 & 1 & 0 & 500 & 0 & 300 & 0 \\ 0 & 1 & 0 & 0 & 500 & 0 & 300 \\ 0 & 0 & 1 & 300 & 0 & 200 & 0 \\ 0 & 0 & 1 & 0 & 300 & 0 & 200 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{5.29}$$

By definition (4.37), the dual equivalent of the primal's relaxed problem is formulated as:

$$\begin{array}{rl} \text{minimize} & y_1 + y_2 + y_3 + 500y_4 + 600y_5 + 300y_6 + 200y_7 \\ \text{s.t.} & \begin{array}{rrrrrrrl} y_1 & & & +200y_4 & & +100y_6 & & \geq 1 \\ y_1 & & & & +200y_5 & & +100y_7 & \geq 1 \\ & y_2 & & +500y_4 & & +300y_6 & & \geq 1 \\ & y_2 & & & +500y_5 & & +300y_7 & \geq 1 \\ & & y_3 & +300y_4 & & +200y_6 & & \geq 1 \\ & & y_3 & & +300y_5 & & +200y_7 & \geq 1 \\ y_1, & y_2, & y_3, & y_4, & y_5, & y_6, & y_7 & \in [0, 1] \end{array} \end{array} \tag{5.30}$$

Solving the dual version of the primal's LP relaxed problem yields:

Decision variable	value
y_1	0.666666666666667
y_2	0
y_3	0.333333333333333
y_4	0
y_5	0
y_6	0.00333333333333333
y_7	0.00333333333333333

The objective value is 2.666667, by:

$$\begin{aligned} & y_1 + y_2 + y_3 + 500y_4 + 600y_5 + 300y_6 + 200y_7 \\ = \; & 0.666666666666667 + 0 + 0.333333333333333 + 0 + 500 \cdot 0 + 600 \cdot 0 \\ & + 300 \cdot 0.00333333333333333 + 200 \cdot 0.00333333333333333 \\ = \; & 2.66666666666667 \end{aligned} \tag{5.31}$$

5.6 Variations

Given the linear programming model in section 5.4, this section discusses the feasibility of each variation given in section 2.5. The following subsections discuss a possible solution to each variation.

5.6.1 Task dependencies

Tasks that need to be executed in serial ordering can be scheduled to a single or multiple nodes for execution. Tasks are required to have an order, or dependency, specified. The scheduler schedules a single task per scheduling operation with respect to the order specified. Scheduling all serial ordered tasks at once is a waste of node resources because only a single task is able to be executed at the time.

The scheduler is able to schedule all serial ordered tasks to a single node when desired. Scheduling serial ordered tasks to a single node can be expensive in terms of time since the scheduler cannot guarantee whether enough resources free up in a timely manner for the next task in order. Scheduling depending tasks to a specific node can be done by assigning a tag to the tasks and node, as discussed in subsection 5.6.5.

For efficiency, order depending tasks have to be placed in a separate queue to ensure a proper execution order; the scheduler should only consider a single task at the time. The client application, that is responsible for task execution, has to report back to the scheduler indicating the task has finished. Upon receiving the finished message, the scheduler considers the next task from the ordered queue for scheduling.

Whether scheduling serial ordered tasks to a single nodes or to multiple nodes does not make a difference in terms of operational complexity since every node has to report that the task is finished before considering the next task in order. The only difference is that extra constraints are required when the ordered tasks have to be scheduled on a single node, a tag constraint.

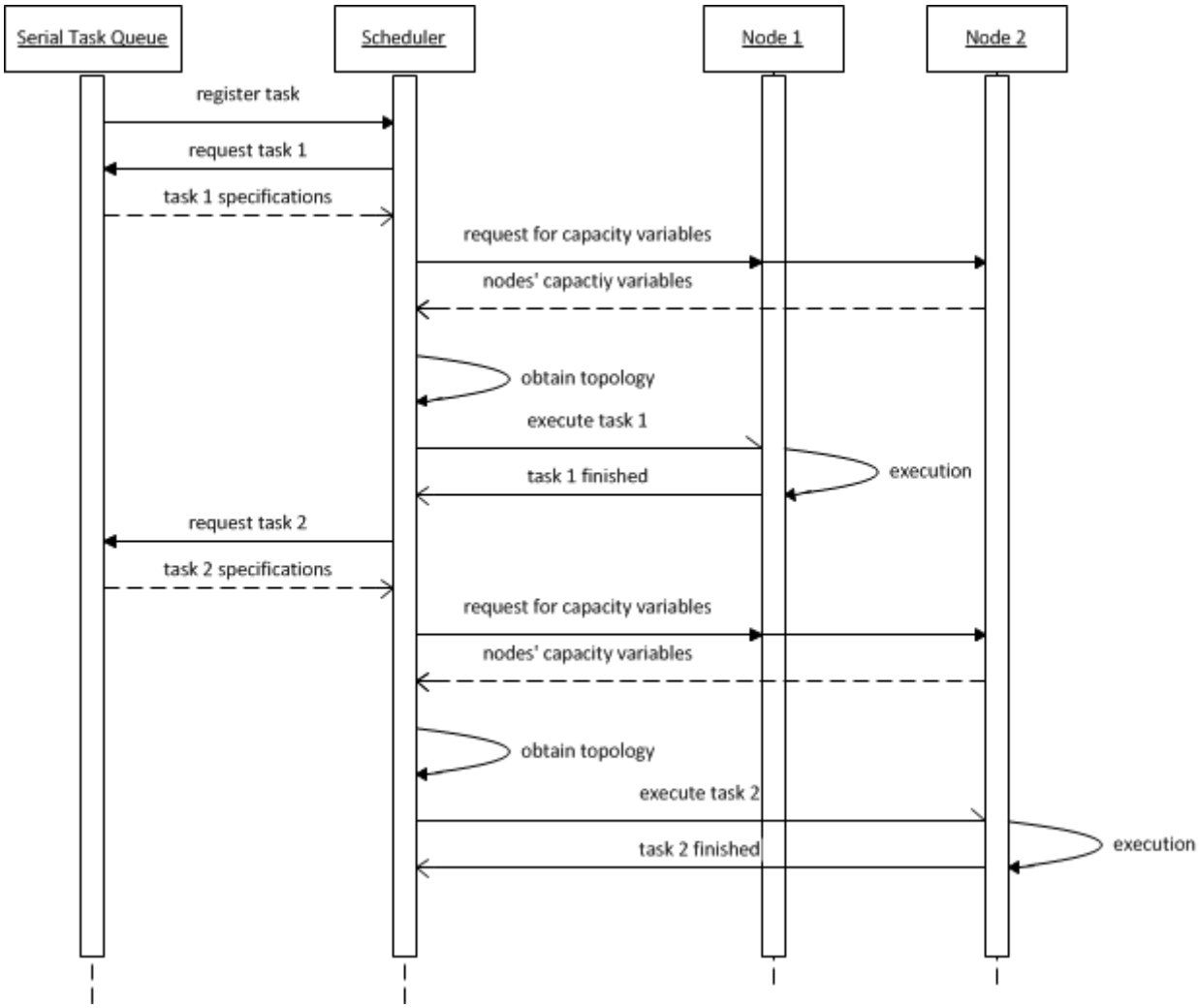


Figure 5.2: Scheduling process of ordered tasks

Figure 5.2 depicts a sequence diagram to illustrate scheduling serial ordered tasks. The serial task queue contains every task that is required to be executed in serial order. The scheduler is the process of collecting task requirements, collecting node variables, scheduling and producing a topology with solving the linear programming problem. The first, highest priority, task is requested by the scheduler and scheduled for execution on node 1 in this particular case. After task 1 has completed execution, node 1 returns a message to the scheduling process indicating that the task is finished. The scheduler requests the next task from the task queue for a next round of scheduling. Task 2 is scheduled for execution on node 2. This process continues until the task queue is empty, indicating that all serial ordered tasks have been executed.

5.6.2 Point in time

Scheduling to execute a task in the future requires two timestamps, the current time and the desired time of execution. Based on both timestamp values, the scheduler is able to compare both values and decide whether or not the desired execution time has been reached. Once the desired execution time has been reached, the scheduler can assign the task to an available node.

A possible solution to schedule tasks that are required to be executed at a future

point in time is to queue tasks and consider them only when the point in time is reached. This can be done similarly to handling task dependencies, as discussed in subsection 5.6.1. The tasks are queued until their desired execution time is reached. Once a task's execution time is reached, the scheduler function receives the task as if the task were a normal task. In other words, tasks are preprocessed and point in time tasks are set aside, in a queue, until their desired point in time is reached.

Another possible solution is to add a constraint to the linear programming model. The constraint forces the task's corresponding decision variables to zero when the desired timestamp is not yet reached. Intuitively, the constraint can be modeled as:

$$TS \cdot x_{i1} + \ldots + TS \cdot x_{ij} + \ldots + TS \cdot x_{im} \geq PTS \tag{5.32}$$

Where TS is the current timestamp and PTS is the task's preferred execution timestamp. A timestamp is a positive integer representing a point in time. At most one corresponding decision variable can be set to one at all times, due to the existing constraints in the standard problem, forcing the left-hand side of the inequality to yield the TS value. The inequality becomes $TS \geq PTS$. However, consider the possibility that the task is not scheduled at all, i.e. all corresponding decision variables are set to zero. When all corresponding decision variables are zero valued, the constraint becomes:

$$\begin{aligned} & TS \cdot x_{i1} + \ldots + TS \cdot x_{ij} + \ldots + TS \cdot x_{im} \geq PTS \\ = \\ & TS \cdot 0 + \ldots + TS \cdot 0 \geq PTS \\ = \\ & 0 \geq PTS \end{aligned} \tag{5.33}$$

Where the constraint evaluates to false since the preferred timestamp is a positive integral number, which is always greater than the value zero. The constraint has failed and the whole model has become infeasible. This behavior is incorrect since it should be possible to not schedule a particular task, e.g. by having insufficient resources available for the task to be scheduled at that point in time.

To allow zero valued decision variables, the constraint has to be adjusted. Similar to the left-hand side of the inequality, the right-hand side also has to depend on the decision variables. The improved modelling of the constraint is:

$$TS \cdot x_{i1} + \ldots + TS \cdot x_{ij} + TS \cdot x_{im} \geq PTS \cdot x_{i1} + PTS \cdot x_{ij} + \ldots + PTS \cdot x_{im} \tag{5.34}$$

In case the decision variables are set to zero, both the left-hand side and the right-hand side of the inequality evaluate to zero.

The inequality has to be converted to standard, dictionary, form for processing in the scheduler function. The final inequality constraint is transformed to:

$$\begin{aligned} & TS \cdot x_{i1} + \ldots + TS \cdot x_{ij} + TS \cdot x_{im} \geq PTS \cdot x_{i1} + PTS \cdot x_{ij} + \ldots + PTS \cdot x_{im} \\ = & \text{ (move right-hand side decision variables to left-hand side)} \\ & \begin{matrix} TS \cdot x_{i1} + \ldots + TS \cdot x_{im} \\ -PTS \cdot x_{i1} - \ldots - PTS \cdot x_{im} \end{matrix} \quad \geq \quad \begin{matrix} PTS \cdot x_{i1} + \ldots + PTS \cdot x_{im} \\ -PTS \cdot x_{i1} - \ldots - PTS \cdot x_{im} \end{matrix} \\ = & \\ & TS \cdot x_{i1} + \ldots + TS \cdot x_{im} - PTS \cdot x_{i1} - \ldots - PTS \cdot x_{im} \geq = 0 \\ = & \text{ (} \geq \rightarrow \leq \text{)} \\ & -TS \cdot x_{i1} - \ldots - TS \cdot x_{im} + PTS \cdot x_{i1} + \ldots + PTS \cdot x_{im} \leq 0 \end{aligned} \tag{5.35}$$

To illustrate the time constraint, consider the example described in subsection 5.4.1. Suppose task 2 has to be executed at a certain point in time. Assume the preferred execution timestamp is 1040, which is arbitrary chosen but it can be thought of as 10:40 AM, and assume the current timestamp is 1039. Based on assumptions, the constraint is:

$$
\begin{aligned}
& 1039 \cdot x_{21} + 1039 \cdot x_{22} \geq 1040 \cdot x_{21} + 1040 \cdot x_{22} \\
= & \text{ (move right-hand side decision variables to left-hand side)} \\
& 1039 \cdot x_{21} + 1039 \cdot x_{22} - 1040 \cdot x_{21} - 1040 \cdot x_{22} \geq 1040 \cdot x_{21} + 1040 \cdot x_{22} - 1040 \cdot x_{21} - 1040 \cdot x_{22} \\
= & \\
& 1039 \cdot x_{21} + 1039 \cdot x_{22} - 1040 \cdot x_{21} - 1040 \cdot x_{22} \geq = 0 \\
= & \text{ (} \geq \rightarrow \leq \text{)} \\
& -1039 \cdot x_{21} - 1039 \cdot x_{22} + 1040 \cdot x_{21} + 1040 \cdot x_{22} \leq 0 \\
= & \\
& x_{21} + x_{22} \leq 0
\end{aligned}
\tag{5.36}
$$

Since timestamps $1039 < 1040$, the task should not be scheduled. The constraint is only true when $x_{21} = 0$ and $x_{22} = 0$. Any other combination evaluates to false; $1 \leq 0$. This forces the model to set the corresponding decision variables to zero.

Suppose the current timestamp has moved to 1040, the task should be evaluated for scheduling. The constraint becomes:

$$
\begin{aligned}
& -1040 \cdot x_{21} - 1040 \cdot x_{22} + 1040 \cdot x_{21} + 1040 \cdot x_{22} \leq 0 \\
= & \\
& 0 \cdot x_{21} + 0 \cdot x_{22} \leq 0
\end{aligned}
\tag{5.37}
$$

All of the terms will evaluate to zero. In every case the constraint evaluates to true.

Suppose the current timestamp has moved to 1045, the constraint becomes:

$$
\begin{aligned}
& -1045 \cdot x_{21} - 1045 \cdot x_{22} + 1040 \cdot x_{21} + 1040 \cdot x_{22} \leq 0 \\
= & \\
& -5 \cdot x_{21} + -5 \cdot x_{22} \leq 0
\end{aligned}
\tag{5.38}
$$

For every $TS > PTS$, the entire left-hand side is strictly negative. Each term can be either negative or zero. Strictly negative results causes the constraint to evaluate to true.

It is possible to compare the two possible solutions. Preprocessing tasks does not burden the linear programming algorithm, or simplex, while adding constraints does. Adding more constraints to the linear programming model implies more constraints to be verified and satisfied. Introducing additional constraints might tighten the solution space beforehand in cases where $TS < PTS$. However, the time to solve the model by the simplex algorithm might be slower in cases where $TS \geq PTS$, because the outcome is the same whether or not the constraint is added and an extra constraint has to be evaluated and satisfied. When other variations are already used, or desirable to be used, such as task dependencies, tasks are already preprocessed. Extending preprocessing to serve point in time tasks is not too far-fetched.

5.6.3 Duration

The scheduler has the ability to allocate tasks to nodes given its inputs. A node measures variables which represent a node's capacity. Based on the scheduler's inputs, the scheduler produces a topology as its output. Some tasks are services and are required to be executed over a certain period of time.

A possible solution is to specify that a task requires certain resources over a given period of time. The scheduler schedules the task to a node that meets the task's requirements. The node's application, which measures capacity variables, takes the task's duration into account. Resources are kept reserved even when resources are freed up by other tasks and system processes. Since a node has sufficient capacity to execute the task once, the node should have at least the same amount of capacity during the entire period of time.

Scheduling a duration task is no problem for the scheduler, such tasks are to be scheduled as any normal task. However, it is a challenge to determine the actual reserved capacity once the task is executed on a node. It is naive to say that the required capacity is equal to the reserved capacity for the given task within the duration of execution since the task is already consuming resources. The available capacity for a single variable v is:

$$capacity_v = free_v - (reserved_v - consumed_v) \tag{5.39}$$

Where the current consumed resources is subtracted from the reserved resources, which resembles the free resources that are available/reserved to the duration task. In turn, the reserved resource number is subtracted from the total free measurement to yield the remaining capacity. The remaining capacity is one of the scheduler's inputs for the next round of scheduling.

5.6.4 Task priority

Giving preference to a task over another task can be done by adjusting the objective function. The objective function of the general problem maximizes the number of scheduled tasks. It is possible to change the objective function's perspective by saying that every task has equal priority by imagining that there is a coefficient with value one assigned to every decision variable. The perspective can be used to give a decision variable some other weight, or priority.

Given the general problem's objective function definition (5.1), the prioritized objective function becomes:

$$\text{maximize} \sum_{i=1,j=1}^{n,m} prio_i x_{ij} \tag{5.40}$$

Where $prio_i$ is the priority value assigned to task i, n is the number of tasks, and m is the number of nodes. The coefficient $prio_i$ determines the direction of the priority and its weight. There are three possible directions; lower, equal and higher priority. The priority given by $prio_i$ is defined as:

$$priority(prio_i) = \begin{cases} \text{lower priority} & 0 < prio_i < 1 \\ \text{equal priority} & prio_i = 1 \\ \text{higher priority} & 1 < prio_i \end{cases} \tag{5.41}$$

Priority coefficient values $prio_i \leq 0$ are no entering variable candidates since it does not increase the objective function, the result is that the task is not scheduled.

Adjusting the coefficients corresponding to a task's decision variables changes the meaning of the objective value. The original problem's objective value represents the number of scheduled tasks whereas the prioritized objective function's value does not.

5.6.5 Tags

Tags are useful when tasks need to be executed on a specific set of nodes. Once a task has a tag defined, the task cannot be scheduled to a node with a different tag.

Intuitively, tags can be associated with a unique integral number. Integer identifiers can be used to compare which task has to be scheduled to which node. The integer values can be used as coefficients of the decision variables.

$$tag_{node_1} \cdot x_{i1} + \ldots + tag_{node_j} \cdot x_{ij} + \ldots + tag_{node_m} \cdot x_{im} = tag_{task_i} \tag{5.42}$$

Where tag_{node_j} is the tag associated to node j and tag_{task_i} is the tag associated to task i. Every tagged task has a constraint in the described form. There are at most $2m$ constraints per task, where m is the number of nodes and $2m$ because of the equal sign has to be changed to a less-or-equal-than and greater-equal-than constraint in standard form. There are at most $2m$ constraints per task since it is not mandatory for all nodes to have a tag.

To illustrate the constraint, consider the example described in subsection 5.4.1. Assume that there are two datacenters and some tasks have to be executed in one specific data center, abbreviated by DC. For the example, the following distribution is used:

Task	Tag	Integer value
Task 1	DC1	1
Task 2	DC2	2
Task 3	DC1	1
Node	Tag	Integer value
Node 1	DC1	1
Node 2	DC2	2

Table 5.6: Assignment of tags to nodes and tasks

The constraint for the first task is:

$$\begin{aligned} & tag_{node_1} \cdot x_{i1} + \ldots + tag_{node_j} \cdot x_{ij} + \ldots + tag_{node_m} \cdot x_{im} = tag_{task_i} \\ = \\ & 1 \cdot x_{11} + 2 \cdot x_{12} = 1 \end{aligned} \tag{5.43}$$

Where x_1 is the decision variable defining the assignment of the task to node 1 and x_4 is the decision variable that corresponds to the task's scheduling on node 2. The

constraint's outcome will match the desired value if and only if x_1 has value one. The constraint for the second task is:

$$\begin{aligned} & tag_{node_1} \cdot x_1^{task_1} + \ldots + tag_{node_m} \cdot x_m^{task_n} = tag_{task_t} \\ & tag_{node_1} \cdot x_{i1} + \ldots + tag_{node_j} \cdot x_{ij} + \ldots + tag_{node_m} \cdot x_{im} = tag_{task_i} \\ = \\ & 1 \cdot x_{21} + 2 \cdot x_{22} = 2 \end{aligned} \tag{5.44}$$

Where the constraint can only be satisfied when x_{22} has value one. All constraints in the example are:

$$\begin{aligned} & 1 \cdot x_{11} + 2 \cdot x_{21} = 1 \quad \text{(task 1)} \\ & 1 \cdot x_{21} + 2 \cdot x_{22} = 2 \quad \text{(task 2)} \\ & 1 \cdot x_{31} + 2 \cdot x_{32} = 1 \quad \text{(task 3)} \\ = & \text{ (inequality form)} \\ & 1 \cdot x_{11} + 2 \cdot x_{12} >= 1 \\ & 1 \cdot x_{11} + 2 \cdot x_{12} <= 1 \\ & 1 \cdot x_{21} + 2 \cdot x_{22} >= 2 \\ & 1 \cdot x_{21} + 2 \cdot x_{22} <= 2 \\ & 1 \cdot x_{31} + 2 \cdot x_{32} >= 1 \\ & 1 \cdot x_{31} + 2 \cdot x_{32} <= 1 \end{aligned} \tag{5.45}$$

However, the described constraint is a constraint for a perfect world. There are two problems that need to be addressed; tasks and nodes can have at most one tag and the constraint expects all tasks to be scheduled. It is not possible to have multiple copies of the constraint for every other tag because every constraint has to be satisfied for the model to be feasible. The right-hand side of the inequality cannot be adjusted to not allow a task to be scheduled, similar to the solution in subsection about point in time tasks, because it is required to have a fixed value, i.e. a tag is a unique integer that has to match exactly.

Another possible method becomes clear when changing perspective. Instead of forcing decision variables to set certain values, it is possible to prevent decision variables to set a certain value. Adding constraints where certain decision variables have to be set to zero prevents a task to be scheduled on certain nodes. Tags that do not match one or more node tags required by the task cannot be considered in the scheduling. A constraint is added to exclude the decision variable corresponding the scheduling of the task to a certain node. The constraint is defined as:

$$x_{ij} \leq 0 \tag{5.46}$$

Where x_{ij} is the decision variable that cannot be considered in the scheduling. From a practical point of view, one decision variable per constraint is inefficient while there is a high probability that multiple decision variables are required to be set to zero. Such decision variables can be combined into a single constraint. The constraints for the example described in subsection 5.4.1 are:

$$\begin{aligned} & x_{12} \leq 0 \quad \text{(task 1 cannot be scheduled to node 2)} \\ & x_{21} \leq 0 \quad \text{(task 2 cannot be scheduled to node 1)} \\ & x_{32} \leq 0 \quad \text{(task 3 cannot be scheduled to node 2)} \\ = & \text{ (combining constraints)} \\ & x_{12} + x_{21} + x_{32} \leq 0 \end{aligned} \tag{5.47}$$

5.6.6 Node's resource protection

For a node to perform optimally, it is good practice to not utilize all of its free capacity. Maintaining a resource buffer on nodes offers the opportunity to handle spikes without a potential performance loss. What the actual buffer size should be is a point of debate; it might be a percentage or a fixed value depending on the variable.

The buffer size must be respected by the client application, which measures and sends the variables to the scheduler. When the application measures a value, a percentage or a fixed value has to be subtracted to yield the usable capacity for that variable. The limited variable is send as an input to the scheduler for scheduling.

5.6.7 Custom objective function

The current objective function, or cost function, is maximized to schedule as much tasks as possible. However, maximizing the number of scheduled tasks is not always optimal for the scheduling problem at hand. The problem can be remodeled to suit another objective function. An objective function can either be minimized or maximized. The following requirements have to be met when remodeling the objective function:

- a topology has to be derived from the optimal result;
- schedule all or a subset of available tasks.

In assignment problems, maximizing the objective function is not very useful because assignment problems aim to find a lower bound with respect to some cost. In task scheduling the aim is to optimize the number of scheduled tasks based on an upper bound. It is not possible to satisfy both.

Minimization problems are challenging when it comes to respecting the requirements. Minimization implies that an absolute minimum cost has to be found while, at the same time, requiring to maximize the number of scheduled tasks. It is not possible to optimize both at the same time. Minimizing the current model yields a topology of zero scheduled tasks. The resulting topology is optimal because it is possible to schedule zero scheduled tasks. Generally, minimization has at least one greater-than-or-equal inequality. The general problem's model has only less-than-or-equal inequalities.

Minimized objective functions are possible in the case where there is at least one-greater-than-or-equal inequality added to the model. Assume that we live in a perfect world where every task is able to get scheduled on a node. In a perfect world it is possible to come up with a cost function of our choosing while constraining the model to schedule all tasks. The following constraint can be used to force scheduling of task i:

$$x_{i1} + x_{i2} + \ldots + x_{ij} + \ldots + x_{im} = 1 \tag{5.48}$$

Since all decision variables are binary, at least one decision variable has to be set to one. This forces the task to be scheduled. Consider the example described in subsection 5.4.1 and the following costs of scheduling tasks to nodes to illustrate the constraint:

	node 1	node 2
Task 1	100	200
Task 2	100	200
Task 3	100	200

Table 5.7: Cost of scheduling tasks to nodes

Scheduling tasks to node 2 is twice as expensive as scheduling a task to node 1. The objective function is minimized because the objective is to reduce costs as much as possible. The linear programming model becomes:

$$
\begin{array}{rl}
\text{minimize} & 100x_{11} + 100x_{12} + 100x_{21} + 200x_{22} + 200x_{31} + 200x_{32} \\
\text{s.t.} & x_{11} + x_{12} = 1 \\
& x_{21} + x_{22} = 1 \\
& x_{31} + x_{32} = 1 \\
& 200x_{11} + 500x_{21} + 300x_{31} \leq 500 \\
& 200x_{12} + 500x_{22} + 300x_{32} \leq 600 \\
& x_{11}, x_{12}, x_{21}, x_{22}, x_{31}, x_{32} \geq 0
\end{array}
\tag{5.49}
$$

Solving the LP problem yields the following topology:

Decision variable	Corresponding task	Corresponding node
$x_{11} = 1$	task 1	node 1
$x_{12} = 0$	task 1	node 2
$x_{21} = 0$	task 2	node 1
$x_{22} = 1$	task 2	node 2
$x_{31} = 1$	task 3	node 1
$x_{32} = 0$	task 3	node 2

Table 5.8: Resulting topology of the cost minimized LP

Based on the result, only task 2 is scheduled to node 2 because node 1's capacity limit is reached with the allocation of task 1 and task 3.

In a less perfect world, it is a challenge to obtain a linear programming model with a minimized objective function where not all tasks can be scheduled, e.g. due to insufficient node capacity. A possible inefficient solution is to expand the solution space with every iteration. Every task is forced to get scheduled in the first iteration. At every subsequent iteration, constraints are removed. It is possible to schedule only a subset of available tasks by removing constraints. Iteration continues until a solution is found.

6 Evaluation

This chapter evaluates the linear programming model described in section 5.4 by benchmarking several inputs in a certain test environment. The test environment is described in section 6.1. Inputs are specified by a varying number of tasks, nodes and variables. The test parameters are described in section 6.2. The binary linear programming problem and the relaxed variant are benchmarked. The results of the relaxed variant are discussed in section 6.3 and the results of the binary problem are discussed in section 6.4. The raw results are gathered in appendix A.

6.1 Test environment

The linear programming problem, solved by the scheduler function, is solved by GLPK [8]. GLPK's defaults are used, which includes the use of the simplex algorithm. The test environment, to execute the scheduler function, consists of a single cloud instance with 15GB of internal, including swap, memory and a 2GHz processor. A clean installation of the operating system Ubuntu 13.10 is installed.

The test configurations are bound to 12 hours of processing time, to have a feasible period of waiting time and a total of 15GB internal memory available. The processing time and memory usage is measured by Valgrind's Massif memory heap profiler [14]. Note that Massif has some overhead, due to snapshotting and measuring. A test's processing time includes reading the problem from file, constructing the problem, solving the problem and snapshotting.

6.2 Test parameters

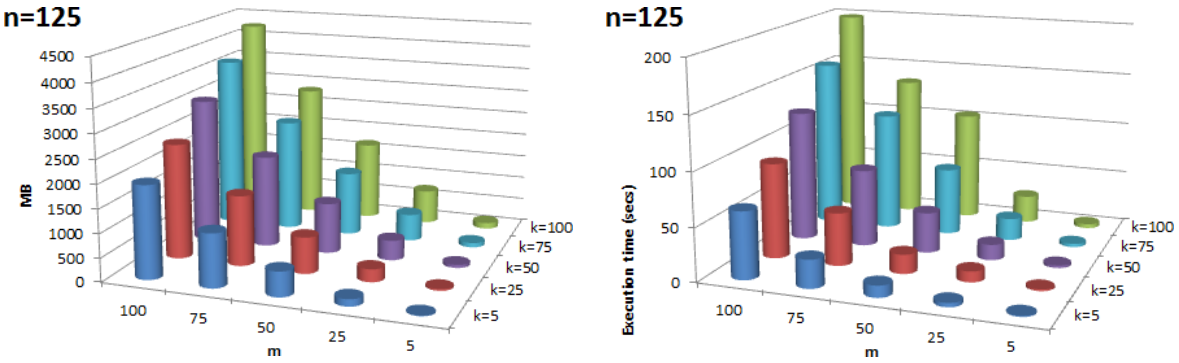
The binary, or integer, linear programming problem is solved with a varying number of tasks, nodes and variables. For comparison, the relaxed integer linear programming problem is also solved with a variety of inputs. All test inputs are pregenerated such that the ILP and the LP relaxation problems only differ in their integrality constraint. A task's requirement is a randomly chosen value between 1 and $1,000$ for each variable. A node's capacity is a randomly chosen value between 500 and $10,000$ for each variable.

The number of tasks, n, starting at 25 and every subsequent test has an increment of 25 tasks. The number of tasks for each test is $[25, 50, 75, 100, 125, \ldots]$. The number of nodes, m, should not be higher than the number of tasks because then every task is able to be scheduled. The number of nodes start at 5 and have an increment of 25 per subsequent test after $m = 25$. The number of nodes is limited to 100 as a fair assumption. The number of nodes for each test is $[5, 25, 50, 75, 100]$. Also, the number of nodes is limited to $5 \leq m \leq n$. The number of variables starts at 5 and have an increment of 25 per subsequent test. The number of variables is limited to 100 as a fair assumption. The number of variables for each test is $[5, 25, 50, 75, 100]$.

In real-world problems, the number of variables $k = 100$ is quite high. For instance, the amount of internal memory available might be the first variable, the second variable might be the available hard disk space, the third variable might be available network throughput and so on. Although the values of k are quite high, they nicely demonstrate the boundaries of the model. The upper limit of nodes, $m = 100$ seems high but are not uncommon in a cloud environment, e.g. it is possible to have 100 low capacity nodes instead of having a single node with high capacity.

6.3 Results LP relaxation

Several trial and error tests were executed to obtain an upper bound to the LP relaxation problem. The n number of tasks tested were $[25, 50, 75, 100, 125, 150, 175, 200, 225, 250, 275, 300, 325, 350, 375]$.



(a) Memory usage

(b) Execution time

Figure 6.1: LP relaxation, $n = 125$

Figure 6.1 depicts the memory usage, in megabytes, and execution time, in seconds, for the tests where the number of tasks are $n = 125$. The x-axis shows the values for the number of nodes m and the y-axis shows the values for the number of variables k. The z-axis shows the amount of memory or execution time consumed by each individual test. The test where $m = 100$ and $k = 5$ resulted in a memory consumption of $\sim$1900MB and an execution time of almost 65 seconds. There are bars for all test configurations in this figure, which implies that all test are run successfully within the limitations of the test environment.

There are $n+km$ constraints and nm decision variables. For a problem where $n = 125$, $m = 100$ and $k = 5$, there are $n + km = 125 + 100 \cdot 5 = 625$ constraints and $mn = 100 \cdot 125 = 12,500$ decision variables. In a dictionary data structure, the number of decision variable values are 7, 812 and 500 by multiplying the number of constraints by the number of decision variables.

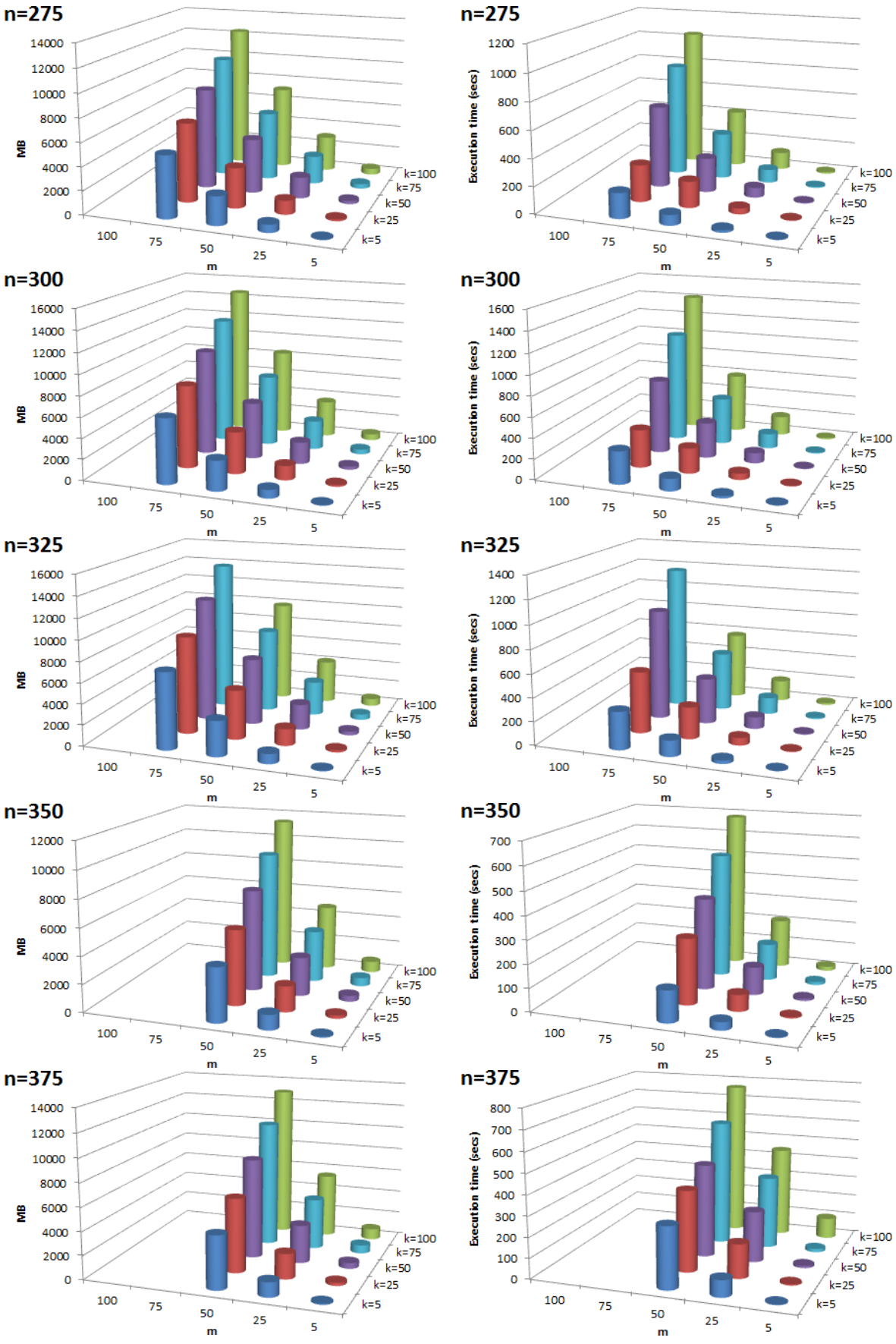


Figure 6.2: LP relaxation results, $n = [275, \ldots, 375]$

Figure 6.2 shows test results for values of n between 275 and 375. Tests start to become memory bounded after $n \geq 275$ where $m = 100$, this can be seen on test results where $n = 275$ and where $n = 300$. The test where $n = 325$, $m = 75$ and $k = 100$ is also memory bounded, notice the green bar missing in the back of the figure. Tests where $n \geq 350$ start to become even more memory bounded, as can be seen from tests where $m = 75$.

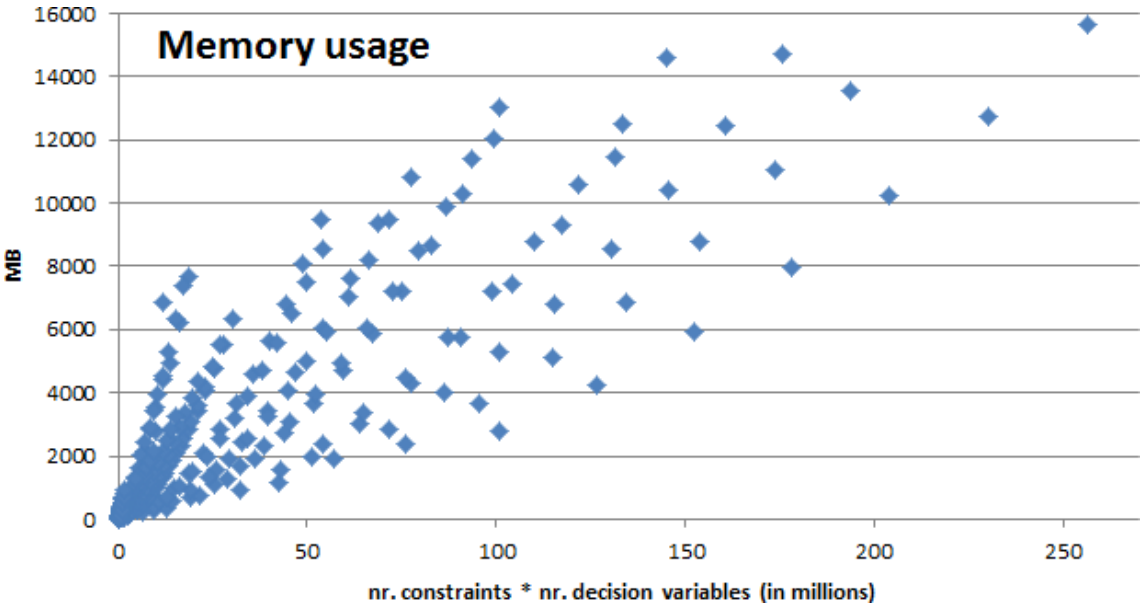


Figure 6.3: LP relaxation results, memory usage

Figure 6.3 shows the memory usage of every test performed. The point on the top right is the closest to the memory limit in the test environment. That point belongs to the test $n = 250$, $m = 100$ and $k = 100$. This particular test has 10,250 constraints and 25,000 decision variables.

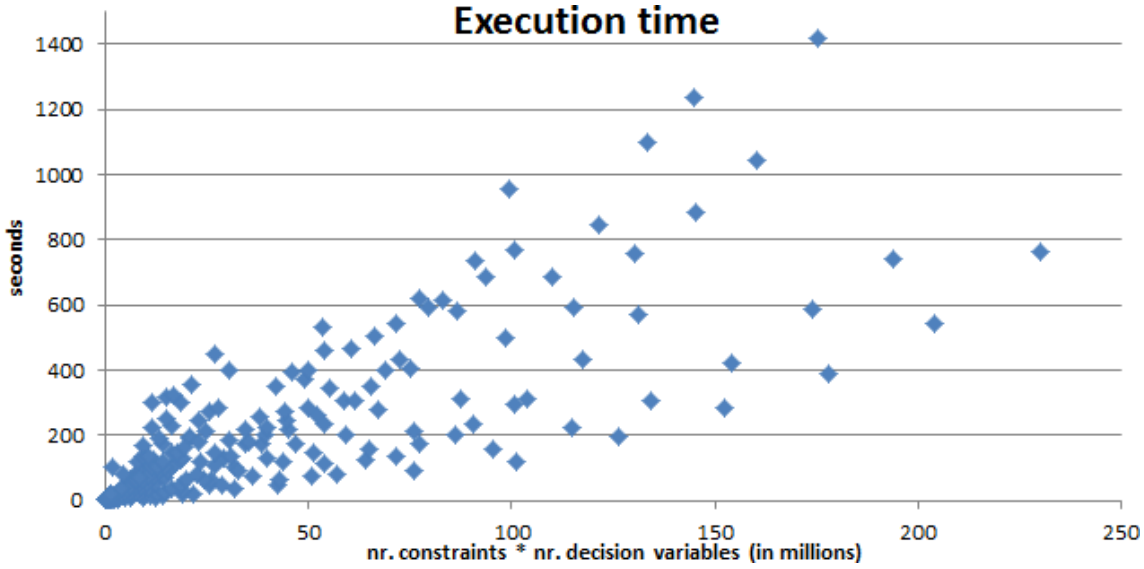


Figure 6.4: LP relaxation results, execution time

The LP relaxed variant seems to be memory bounded in this particular test environment and for the test configurations processed. The highest execution time measured is ~1416 seconds, which is nearly 24 minutes and is well below the

maximum allowed execution time of 12 hours. The test with the highest execution time is test configuration where $n = 300$, $m = 75$ and $k = 100$. The execution time of each test is shown in figure 6.4.

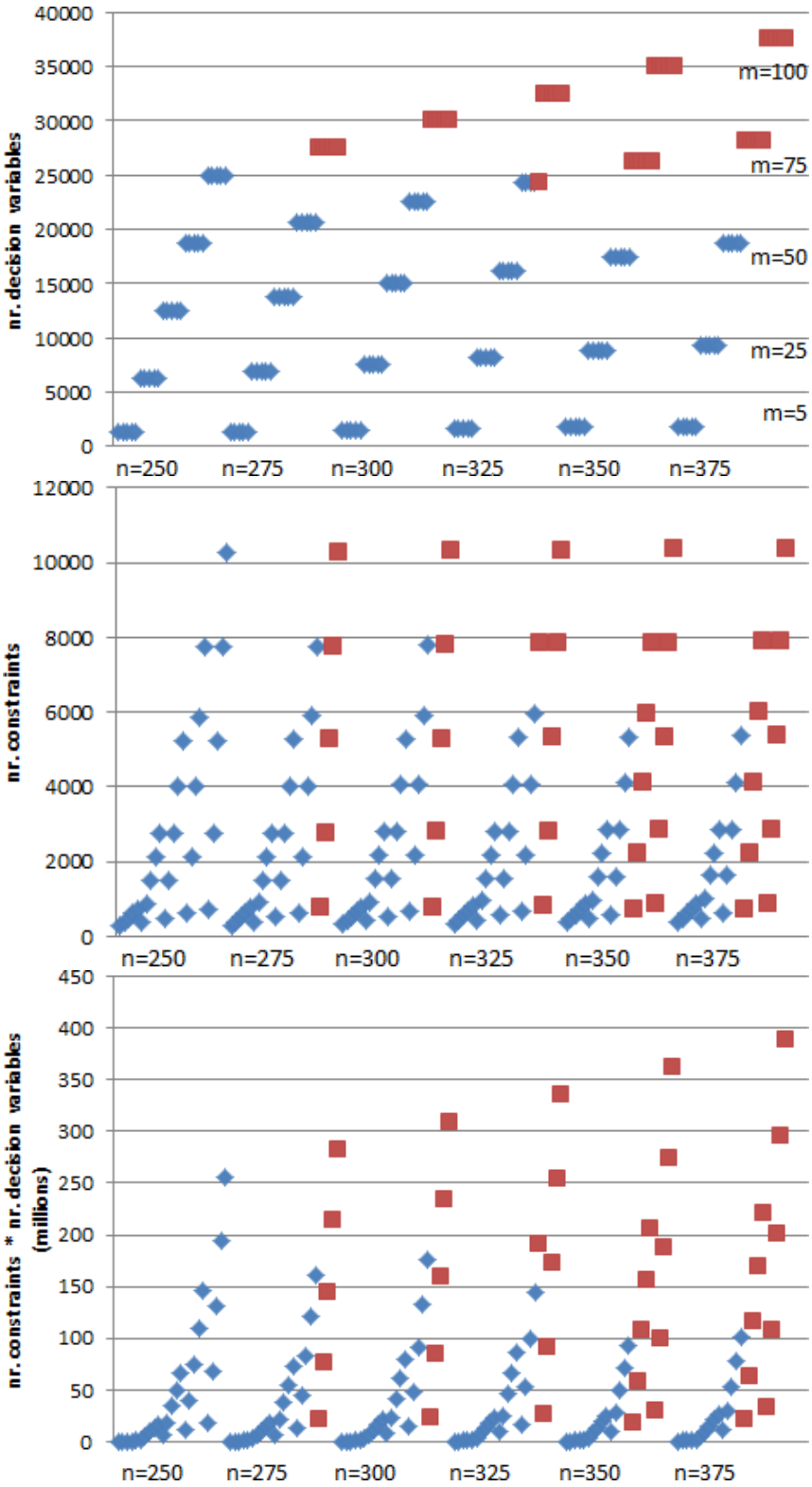


Figure 6.5: LP relaxation results, $n = [250, \ldots, 375]$ characteristics

For a single test, the number of decision variables, the number of constraints and the multiplication of both can provide insights in finding an upper bound to the LP relaxation within the bounds of the test environment. Figure 6.5 shows these characteristics for tests where $n = [250, \ldots, 375]$. The results shown in the graph are grouped per value of n. Per group, the left most dot is the test where $m = 5$ and $k = 5$ and the right most dot is the test where $m = 100$ and $k = 100$. The blue dots represent tests that succeeded while the red dots represent tests that are memory bounded.

Based on the number of decision variables, the flipping point of tests becoming memory bound is around 25,000 decision variables. However, this is not a hard line due to tests where $n = 250$ did succeed while tests where $n = 325$ did not.

Based on the number of constraints, there is no clear distinction. The tests where $n \geq 275$ and $m = 100$ failed. For these failed tests, there are a relatively low number of constraints. In other words, a problem can be found with a relatively low number of constraints and still fail.

Based on the multiplication of the number of constraints and the number of decision variables, a clearer distinction can be made; values around 260 million and above tend to fail. The value represents the amount of numbers in a data structure that is used to solve the problem. Each number is, in turn, a data type that consumes a certain amount of bytes. The test environment only has a certain number of bytes available to facilitate the data structure, hence the problem is memory bounded.

6.4 Results Binary LP

The n number of tasks tested for the binary LP's evaluation equals to the n number of tasks tested in the relaxed LP's evaluation, i.e. $n = [25, 50, 75, 100, 125, 150, 175, 200, 225, 250, 275, 300, 325, 350, 375]$. A smaller granularity for the number of variables was used to show feasibility as a bare minimum, i.e. $k = [1, 5, 25, 50, 75, 100]$. The full list of test configurations and results can be found in Appendix A.

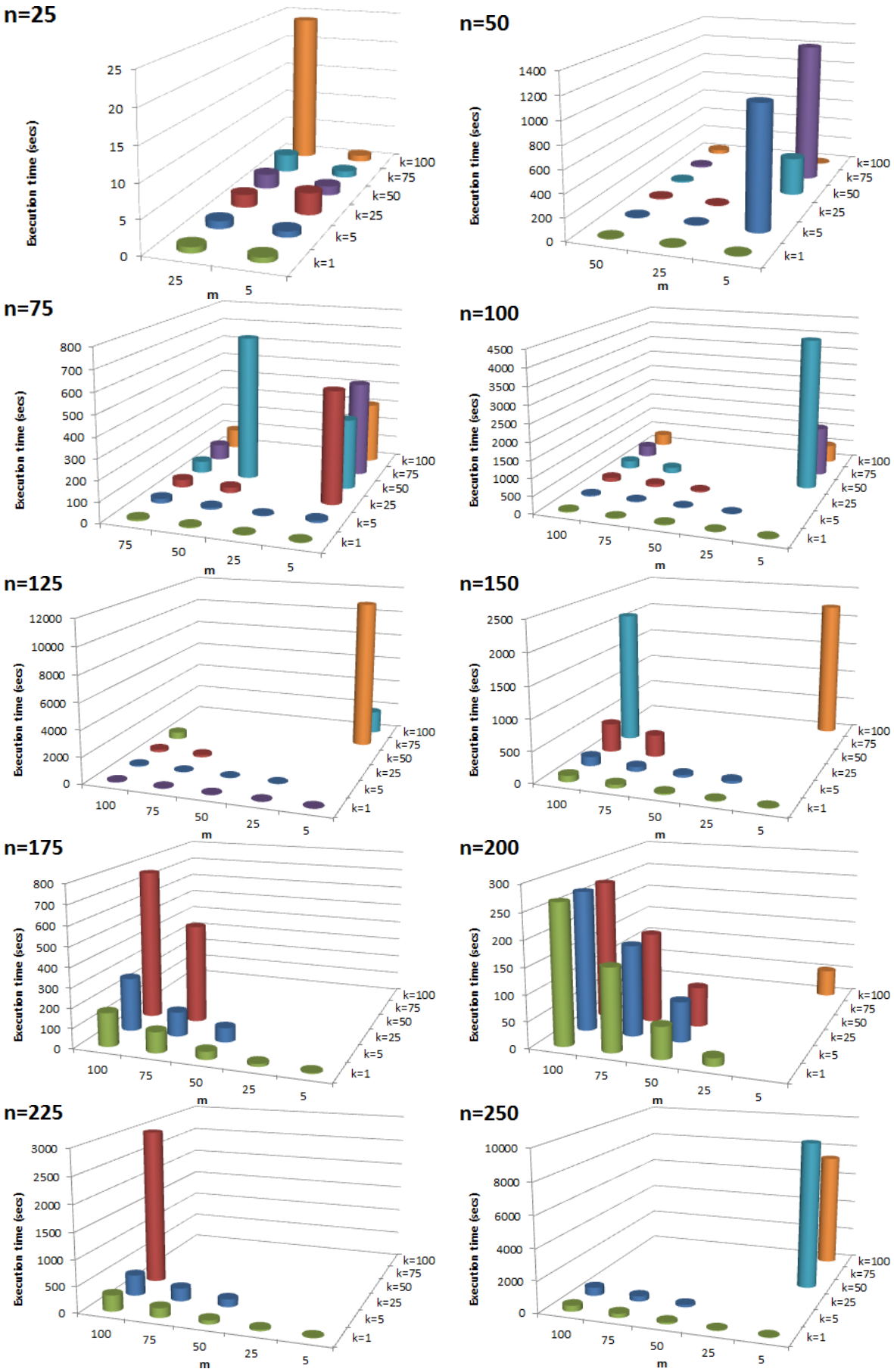


Figure 6.6: ILP results, $n = [25, \ldots, 250]$, execution times

Figure 6.6 shows test results, execution times, for values of n between 25 and 250. All $n = 25$ tests succeeded. However, as expected, far more tests failed in comparison to the LP relaxation tests; tests started to fail as soon as values of n go beyond 25. Based on the results, tests tend to fail on higher values of k with lower values of m. Higher values of m implies that there are more nodes to schedule tasks to which leads to a less stringent search space. It is interesting to see that some tests succeed on higher values of k on tests where $m = 5$, most probably; the search space can be significantly reduced due to the amount of constraints a higher number of k imposes. All failed ILP test configurations up to $n = 250$ exceeded the execution time limit of 43200 seconds, or 12 hours.

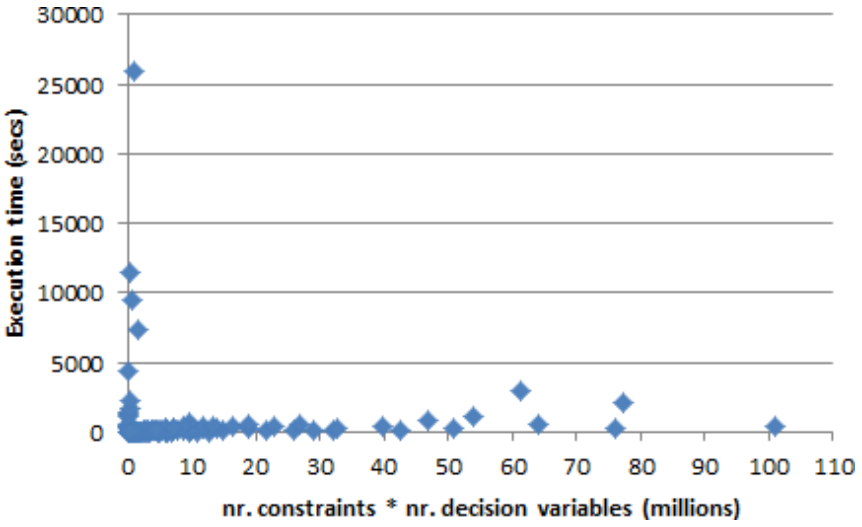


Figure 6.7: ILP results, $n = [25, \ldots, 275]$, execution times vs nr. constraints * nr. decision variables

Figure 6.7 shows the execution times of all test configurations that succeeded. The median execution time is ~53 seconds. The highest execution time measured is ~25,927 seconds, or ~7 hours, for test configuration $n = 275$, $m = 5$ and $k = 100$. The second highest execution time measured is ~11,426 seconds, or ~3 hours, for test configuration $n = 125$, $m = 5$ and $k = 75$. Neither the median nor the highest measured execution times, peaks, are near the execution time limit. A possible reason for these peaks is inefficient heuristics, GLPK might have chosen a suboptimal branch in the search tree, finding a local optimum and there was insufficient time left to exhaust every possible branch. A simple test was executed to demonstrate the importance of heuristics. Test configurations $n = 75$, $m = 25$ and $k \geq 25$ failed with GLPK's default settings. When choosing GLPK's depth-first search heuristic, test with $k = 75$ succeeded in ~87 seconds and test with $k = 100$ succeeded in ~303 seconds. Despite of tests $k < 75$ failed, it shows that a different heuristics can make a difference.

Although the execution time limit is reached for most of the test configurations, GLPK reported a suboptimal solution relatively fast but did not obtain the optimal value within the time window. Also, in some occurrences, GLPK reported that the suboptimal solution found only differed by one in comparison to the expected optimal solution. However, it is not guaranteed that the solution is found within minutes to come due to inefficient heuristics. For example, the test configurations $n = 175$, $m = 5$ and $k \geq 50$ demonstrate the described behavior. Given enough time, GLPK will eventually find an optimal result or become memory bounded.

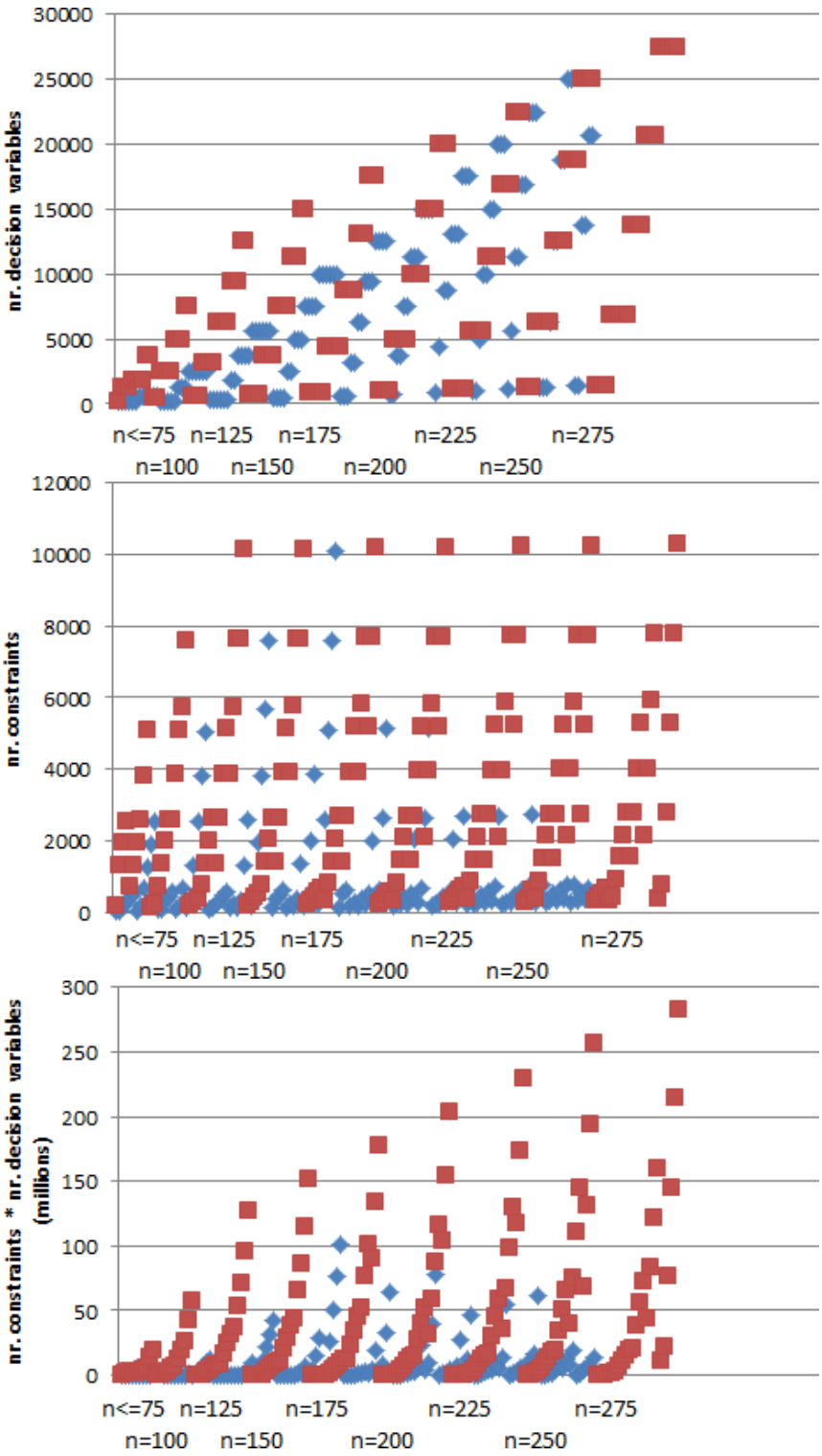


Figure 6.8: ILP results, $n = [25, \ldots, 275]$, characteristics

Figure 6.8 shows the characteristics of all test configurations. Similar to the LP relaxation results, blue points correspond to feasible tests and red points correspond to infeasible tests. However, an upper boundary is not that obvious for the ILP test results as that was in the LP relaxation test results. The highest value in the A-matrix, the number of constraints times the number of decision variables, of a feasible solution is 101 million, of test configuration $n = m = k = 100$, which seems like a reasonable upper bound for this particular test environment. The randomness of the generated test sets might have an influence in the results shown in figure

6.8 because a random value in a problem's constraint may be more stringent than another random value reducing the search space faster. A more uniform data set might provide better insights into the upper bounds of the ILP problem.

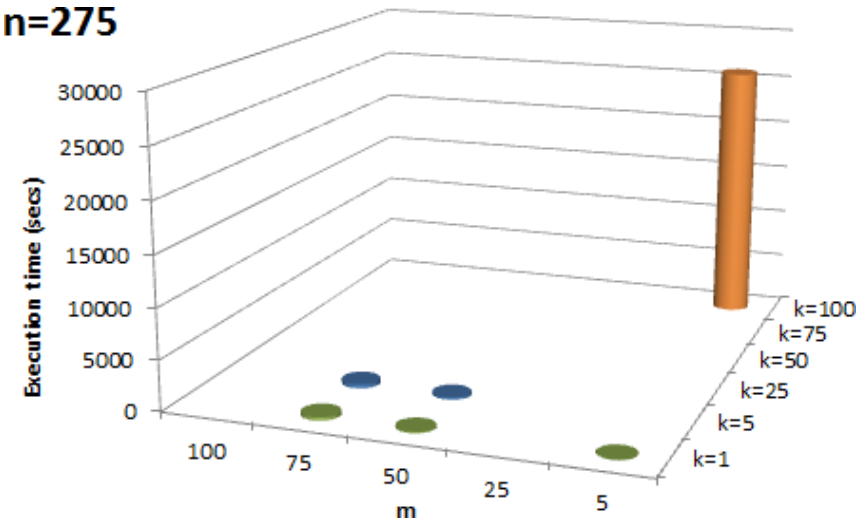


Figure 6.9: ILP results, $n = 275$, execution times

The memory usage of ILP tests is roughly similar to the memory usage of the LP relaxation tests. For example, test configuration $n = 250, m = 100$ and $k = 5$ consumed 7.497GB for the ILP problem versus 7.498GB for the LP relaxation problem. This implies that ILPs are both memory bound, inherited from the LP relaxation, and time bound. Figure 6.9 shows that all test configurations where $n = 275$ and $m = 100$ failed due to the memory boundary while the other failed $n = 275$ tests are execution time bound.

7 Discussion

Based on the proposed linear programming model, common variations were discussed. Task dependencies should not be included in the LP model due to efficiency. For instance, two tasks depend on each other; at point in time t both tasks are considered in the scheduling. The second task is never scheduled since the first task has not been executed. The second task should be included in the LP problem when the first task has been executed. Tasks that should be executed at some future point in time t can be included in the LP problem with some additional constraints. However, efficiency is questionable because the scheduler produces several topologies for different points in time; to use preprocessing to only consider these tasks is not too far-fetched. Tasks that require to be executed for a certain period of time are to be scheduled as regular tasks. The only difference is that the node's application, responsible for the node's variable measurements, has to respect the period of time when reporting the available resources to the scheduler. The client application can also reserve capacity to protect a node's operating system from exhausting capacity to operate. Prioritized tasks can be added to the LP problem by changing the objective function. Adjusting the objective function is not very flexible; maximizing the number of tasks scheduled must be respected. Scheduling tasks to nodes with equal tags can be done by adding constraints that force the decision variables to take on certain values.

Node variable values can be obtained by the client application. The client application is installed on each node. The client application measures the current state of each variable of the node at request of the scheduler.

Task variables can be obtained by using models that depend on historical data. However, not all tasks are similar and/or repetitively executed to obtain historical data. When constructing the task, the knowledge of the system administrator and/or task developer can be used to form task variable values. Also, task variables can be obtained before a task is scheduled for execution; one or more test passes have to be done to ensure a task behaves as expected and produces the expected result. The test/acceptance environment can be adjusted such that historical data is obtained.

The efficiency of linear programming as part of a task scheduler is questionable. Test results show that only small, or very stringent, problems yield a scheduling. Large problems were either memory or time bound. Although linear programming will produce an optimal solution, it can, and most probably will, take sufficient amount of time and memory to do so. However, the processing time of large problems will incur too much overhead for the entire scheduling to be efficient, i.e. it would be faster to arbitrarily schedule tasks instead of using linear programming.

The fairness of a scheduling is debatable. For instance, the objective function's objective is to maximize the number of the to be scheduled tasks. However, how does the objective function affect the scheduling of tasks with high valued variables in comparison to tasks with low valued variables? Assume a worst case scenario where there is one high demanding task and an infinite amount of low demanding tasks. As a result, the high demanding task is not scheduled unless the scheduler function, responsible of solving the linear programming problem, forces the task to be scheduled by adding a constraint in the form $x_{i0} + \ldots + x_{ij} + \ldots + x_{im} = 1$.

However, this can lead to an infeasible problem because there might not exist a node with sufficient capacity to execute the high demanding task.

The dynamicity of the task scheduler is provided by the scheduler function requesting node variables and the client application providing the node variable values. The client application provides variable values for the scheduler to exploit the full extent of the available resources in the entire distributed environment.

The task scheduler's transparency is ensured by the scheduler function. The user only provides a set of tasks to the scheduler. There is no need to provide a list of nodes; the client application is installed on all available nodes and the client application registers the node with the scheduler. The scheduler requests variable measurements from all nodes and produces a topology at point in time t. The scheduler only sends a specific task to a specific node when the task is scheduled to that node.

In conclusion, linear programming is only feasible for manageable sized problems due to the efficiency of the scheduler. Although linear programming can consume a significant amount of memory, the time consumed to yield a solution is a major concern. For instance, a manageable sized problem is a problem where there are at most $k = 5$ variables since test configurations where $k = 1$ and $k = 5$ yielded a solution most often.

While the bounded random test sets are well suited to evaluate the relaxed linear programming model, the same could not be said for the integer linear programming model. Although the results for the integer linear programming model do show boundaries in terms of execution time, some theoretically worse problems yielded a result while a relatively simpler test did not. For example, test sets where $n = 50, m = 5$ and $k \in \{1, 5, 50, 75, 100\}$ yielded a result while test set where $n = 50, m = 5$ and $k = 25$ did not. For future test work it is recommended to generate one, and only one, problem set where only a subset is used for every test to have uniform test data across all tests, e.g. a problem with 75 nodes will have the same 75 nodes as the first 75 in a 100 nodes test. Of course, the best test inputs are to have a problem set that is similar, or exact, to a practical, real life, environment.

GLPK was used in the evaluation of test sets. However, for future work, it might be interesting how other methods for solving integer linear programming problems perform in comparison to simplex. Other methods for solving linear programming problems are, among others, the Hungarian method and the interior point method. GLPK is only able to solve the relaxed linear programming problem with the interior point method [8]. Although GLPK is a generic solver, a dedicated solution, with sophisticated heuristics, can be researched to solve the task scheduling problem more efficiently. Heuristics are important to bound the search space and the traversal of the search tree can make a difference in terms of execution time. Also, special ordered sets [1], type 1, should be considered in future work. A special ordered set is a set of decision variables that allows at most one strictly positive value while all other values, in the same set, being at zero. Special ordered sets are well suited to all decision variables that correspond to a single task.

A Raw results

A.1 LP relaxation

The following table shows the raw results of every executed LP relaxation test configuration.

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
25	5	5	50	125		1	1
25	5	25	150	125		3	1
25	5	50	275	125		5	1
25	5	75	400	125		8	1
25	5	100	525	125	8.692	11	1
25	25	5	150	625		8	1
25	25	25	650	625		18	1
25	25	50	1275	625		31	2
25	25	75	1900	625		43	2
25	25	100	2525	625		56	3
50	5	5	75	250		3	1
50	5	25	175	250	25.725	8	1
50	5	50	300	250	14.962	15	1
50	5	75	425	250	15.453	21	1
50	5	100	550	250	11.414	28	2
50	25	5	175	1250		27	1
50	25	25	675	1250		54	2
50	25	50	1300	1250	50	88	4
50	25	75	1925	1250	50	122	5
50	25	100	2550	1250	50	155	6
50	50	5	300	2500		91	3
50	50	25	1300	2500		145	4
50	50	50	2550	2500		212	7
50	50	75	3800	2500	50	280	9
50	50	100	5050	2500	50	347	11
75	5	5	100	375	37.631	5	1
75	5	25	200	375	18.697	15	1
75	5	50	325	375	15.883	28	1
75	5	75	450	375	13.962	41	2
75	5	100	575	375	12.773	53	2
75	25	5	200	1875	75	57	2
75	25	25	700	1875	75	108	4
75	25	50	1325	1875	74.183	171	8
75	25	75	1950	1875	62.981	234	11
75	25	100	2575	1875	61.948	298	14
75	50	5	325	3750	75	195	5
75	50	25	1325	3750	75	297	8
75	50	50	2575	3750	75	424	12
75	50	75	3825	3750	75	551	16
75	50	100	5075	3750	75	678	22
75	75	5	450	5625		415	9
75	75	25	1950	5625		570	14
75	75	50	3825	5625	75	754	20
75	75	75	5700	5625	75	953	34
75	75	100	7575	5625	75	1144	45
100	5	5	125	500	50.747	8	1
100	5	25	225	500	53.341	8	1
100	5	50	350	500	18.446	45	2
100	5	75	475	500	13.56	65	2

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
100	5	100	600	500	12.704	86	3
100	25	5	225	2500		97	3
100	25	25	725	2500		178	6
100	25	50	1350	2500	80.604	280	12
100	25	75	1975	2500	74.345	382	16
100	25	100	2600	2500	65.083	489	19
100	50	5	350	5000	100	339	7
100	50	25	1350	5000		502	12
100	50	50	2600	5000		705	20
100	50	75	3850	5000	100	909	30
100	50	100	5100	5000	100	1112	48
100	75	5	475	7500	100	726	15
100	75	25	1975	7500	100	973	27
100	75	50	3850	7500	100	1281	46
100	75	75	5725	7500	100	1588	64
100	75	100	7600	7500	100	1902	82
100	100	5	600	10000	100	1260	32
100	100	25	2600	10000	100	1582	63
100	100	50	5100	10000	100	2002	76
100	100	75	7600	10000	100	2399	93
100	100	100	10100	10000	100	2806	121
125	5	5	150	625	56.3	12	1
125	5	25	250	625	34.277	36	2
125	5	50	375	625	21.583	65	2
125	5	75	500	625	14.915	95	3
125	5	100	625	625	13.832	125	4
125	25	5	250	3125		148	4
125	25	25	750	3125		269	10
125	25	50	1375	3125	85.586	421	15
125	25	75	2000	3125	76.504	572	21
125	25	100	2625	3125	67.83	723	26
125	50	5	375	6250		522	11
125	50	25	1375	6250		765	18
125	50	50	2625	6250		1067	38
125	50	75	3875	6250	125	1357	64
125	50	100	5125	6250	125	1672	104
125	75	5	500	9375	125	1124	27
125	75	25	2000	9375	125	1481	50
125	75	50	3875	9375	125	1942	73
125	75	75	5750	9375	125	2382	113
125	75	100	7625	9375	125	2831	135
125	100	5	625	12500	125	1951	63
125	100	25	2625	12500	125	2427	90
125	100	50	5125	12500	125	3041	124
125	100	75	7625	12500	125	3645	159
125	100	100	10125	12500	125	4251	198
150	5	5	175	750	58.644	16	1
150	5	25	275	750	30.399	50	2
150	5	50	400	750	24.998	91	3
150	5	75	525	750	15.266	133	4
150	5	100	650	750	14.407	174	5
150	25	5	275	3750		210	5
150	25	25	775	3750	122.329	376	13
150	25	50	1400	3750	95.691	584	21
150	25	75	2025	3750	77.94	791	27
150	25	100	2650	3750	70.553	999	44
150	50	5	400	7500	150	745	16
150	50	25	1400	7500	150	1080	34
150	50	50	2650	7500	150	1493	66
150	50	75	3900	7500	146.894	1909	132
150	50	100	5150	7500	133.849	2323	171
150	75	5	525	11250	150	1609	49

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
150	75	25	2025	11250	150	2106	79
150	75	50	3900	11250	150	2734	116
150	75	75	5775	11250	150	3358	156
150	75	100	7650	11250	150	3983	201
150	100	5	650	15000	150	2790	88
150	100	25	2650	15000	150	3451	128
150	100	50	5150	15000	150	4297	173
150	100	75	7650	15000	150	5120	223
150	100	100	10150	15000	150	5950	282
175	5	5	200	875	68.712	21	1
175	5	25	300	875	29.234	65	2
175	5	50	425	875	18.368	120	3
175	5	75	550	875	15.733	175	5
175	5	100	675	875	12.911	230	6
175	25	5	300	4375		283	7
175	25	25	800	4375	119.465	501	17
175	25	50	1425	4375	103.807	774	26
175	25	75	2050	4375	83.969	1047	42
175	25	100	2675	4375	70.778	1319	59
175	50	5	425	8750	175	1009	24
175	50	25	1425	8750	175	1442	51
175	50	50	2675	8750	172.893	1998	121
175	50	75	3925	8750	156.623	2536	174
175	50	100	5175	8750	130.435	3082	217
175	75	5	550	13125	175	2178	67
175	75	25	2050	13125	175	2827	106
175	75	50	3925	13125	175	3661	148
175	75	75	5800	13125	175	4483	213
175	75	100	7675	13125	175	5305	295
175	100	5	675	17500	175	6857	126
175	100	25	2675	17500	175	4656	171
175	100	50	5175	17500	175	5741	236
175	100	75	7675	17500	175	6865	309
175	100	100	10175	17500	175	7963	389
200	5	5	225	1000	66.103	27	1
200	5	25	325	1000	34.73	83	3
200	5	50	450	1000	20.815	153	4
200	5	75	575	1000	18.37	222	6
200	5	100	700	1000	12.946	292	7
200	25	5	325	5000	200	366	9
200	25	25	825	5000	139.865	647	20
200	25	50	1450	5000	107.072	996	41
200	25	75	2075	5000	70.464	1338	55
200	25	100	2700	5000	71.482	1684	79
200	50	5	450	10000	200	1313	34
200	50	25	1450	10000	200	1864	70
200	50	50	2700	10000	182.81	2570	148
200	50	75	3950	10000	159.921	3270	201
200	50	100	5200	10000	138.262	3968	265
200	75	5	575	15000	200	2832	89
200	75	25	2075	15000	200	3663	136
200	75	50	3950	15000	200	4701	203
200	75	75	5825	15000	200	5771	313
200	75	100	7700	15000	199.568	6808	593
200	100	5	700	20000	200	4934	172
200	100	25	2700	20000	200	6041	235
200	100	50	5200	20000	200	7424	312
200	100	75	7700	20000	200	8808	424
200	100	100	10200	20000		10250	544
225	5	5	250	1125	64.403	34	1
225	5	25	350	1125	29.658	103	3
225	5	50	475	1125	19.116	189	5

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
225	5	75	600	1125	19.615	275	7
225	5	100	725	1125	16.208	362	9
225	25	5	350	5625	220.354	460	13
225	25	25	850	5625	149.093	808	26
225	25	50	1475	5625	95.076	1242	52
225	25	75	2100	5625	79.229	1675	73
225	25	100	2725	5625	70.372	2108	94
225	50	5	475	11250	225	1652	49
225	50	25	1475	11250	225	2339	101
225	50	50	2725	11250	186.133	3215	185
225	50	75	3975	11250	158.244	4083	246
225	50	100	5225	11250	144.297	4949	306
225	75	5	600	16875	225	3575	115
225	75	25	2100	16875	225	4606	178
225	75	50	3975	16875	225	5895	278
225	75	75	5850	16875	225	7221	501
225	75	100	7725	16875	203.729	8522	756
225	100	5	725	22500	225	6230	230
225	100	25	2725	22500	225	7604	309
225	100	50	5225	22500	225	9321	433
225	100	75	7725	22500	225	11039	586
225	100	100	10225	22500	225	12759	764
250	5	5	275	1250	64.84	41	2
250	5	25	375	1250	25.531	125	4
250	5	50	500	1250	19.818	230	6
250	5	75	625	1250	17.856	334	8
250	5	100	750	1250	15.146	439	11
250	25	5	375	6250	233.053	565	15
250	25	25	875	6250	131.561	983	34
250	25	50	1500	6250	99.534	1514	62
250	25	75	2125	6250	86.605	2041	90
250	25	100	2750	6250	72.037	2568	114
250	50	5	500	12500	250	2033	63
250	50	25	1500	12500	250	2869	124
250	50	50	2750	12500	193.977	3913	218
250	50	75	4000	12500	170.477	4985	285
250	50	100	5250	12500	141.247	6039	349
250	75	5	625	18750	250	4400	222
250	75	25	2125	18750	250	5658	225
250	75	50	4000	18750	250	7225	408
250	75	75	5875	18750	235.289	8793	686
250	75	100	7750	18750	212.513	10414	885
250	100	5	750	25000	250	7678	301
250	100	25	2750	25000	250	9351	397
250	100	50	5250	25000	250	11438	572
250	100	75	7750	25000	250	13527	741
250	100	100	10250	25000	250	15667	1275
275	5	5	300	1375	60.006	49	2
275	5	25	400	1375	29.811	150	4
275	5	50	525	1375	21.67	276	7
275	5	75	650	1375	19.465	402	9
275	5	100	775	1375	14.905	527	12
275	25	5	400	6875	227.067	681	18
275	25	25	900	6875	148.653	1181	43
275	25	50	1525	6875	102.736	1814	74
275	25	75	2150	6875	83.443	2441	100
275	25	100	2775	6875	72.644	3069	131
275	50	5	525	13750	275	2455	77
275	50	25	1525	13750	268.393	3453	194
275	50	50	2775	13750	190.673	4701	256
275	50	75	4025	13750	171.82	5947	344
275	50	100	5275	13750	147.518	7230	432

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
275	75	5	650	20625	275	5319	188
275	75	25	2150	20625	275	6817	275
275	75	50	4025	20625	273.919	8690	613
275	75	75	5900	20625	245.152	10557	848
275	75	100	7775	20625	217.738	12431	1042
275	100	5	775	27500		bounded	
275	100	25	2775	27500		bounded	
275	100	50	5275	27500		bounded	
275	100	75	7775	27500		bounded	
275	100	100	10275	27500		bounded	
300	5	5	325	1500	62.154	58	2
300	5	25	425	1500	32.094	177	5
300	5	50	550	1500	21.732	325	8
300	5	75	675	1500	18.427	473	11
300	5	100	800	1500	13.166	620	15
300	25	5	425	7500	253.256	809	23
300	25	25	925	7500	146.042	1396	61
300	25	50	1550	7500	107.3	2141	103
300	25	75	2175	7500	84.59	2882	148
300	25	100	2800	7500	74.597	3615	192
300	50	5	550	15000	300	2914	119
300	50	25	1550	15000	256.164	4091	248
300	50	50	2800	15000	207.202	5558	351
300	50	75	4050	15000	164.892	7028	468
300	50	100	5300	15000	151.716	8499	593
300	75	5	675	22500	300	6320	319
300	75	25	2175	22500	300	8083	374
300	75	50	4050	22500	284.6	10291	734
300	75	75	5925	22500	248.481	12493	1100
300	75	100	7800	22500	219.536	14694	1416
300	100	5	800	30000		bounded	
300	100	25	2800	30000		bounded	
300	100	50	5300	30000		bounded	
300	100	75	7800	30000		bounded	
300	100	100	10300	30000		bounded	
325	5	5	350	1625	81.644	68	2
325	5	25	450	1625	33.106	205	5
325	5	50	575	1625	19.529	377	9
325	5	75	700	1625	17.337	549	13
325	5	100	825	1625	15.263	721	17
325	25	5	450	8125	262.577	946	29
325	25	25	950	8125	146.069	1628	64
325	25	50	1575	8125	99.21	2482	102
325	25	75	2200	8125	85.036	3355	145
325	25	100	2825	8125	78.536	4215	182
325	50	5	575	16250	325	3415	136
325	50	25	1575	16250	277.542	4782	274
325	50	50	2825	16250	200.622	6489	393
325	50	75	4075	16250	175.504	8197	507
325	50	100	5325	16250	151.32	9908	580
325	75	5	700	24375	325	7409	323
325	75	25	2200	24375	325	9461	529
325	75	50	4075	24375	298.8	12022	954
325	75	75	5950	24375	240.709	14582	1238
325	75	100	7825	24375		bounded	
325	100	5	825	32500		bounded	
325	100	25	2825	32500		bounded	
325	100	50	5325	32500		bounded	
325	100	75	7825	32500		bounded	
325	100	100	10325	32500		bounded	
350	5	5	375	1750	72.733	78	2
350	5	25	475	1750	26.055	236	6

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
350	5	50	600	1750	23.48	434	10
350	5	75	725	1750	18.653	631	14
350	5	100	850	1750	15.49	829	19
350	25	5	475	8750	268.054	1095	34
350	25	25	975	8750	133.844	1880	72
350	25	50	1600	8750	111.428	2862	121
350	25	75	2225	8750	82.633	3845	161
350	25	100	2850	8750	77.393	4854	215
350	50	5	600	17500	350	3955	137
350	50	25	1600	17500	277.01	5528	285
350	50	50	2850	17500	200.569	7492	398
350	50	75	4100	17500	170.506	9457	545
350	50	100	5350	17500	149.496	11418	685
350	75	5	725	26250		bounded	
350	75	25	2225	26250		bounded	
350	75	50	4100	26250		bounded	
350	75	75	5975	26250		bounded	
350	75	100	7850	26250		bounded	
350	100	5	850	35000		bounded	
350	100	25	2850	35000		bounded	
350	100	50	5350	35000		bounded	
350	100	75	7850	35000		bounded	
350	100	100	10350	35000		bounded	
375	5	5	400	1875	72.753	88.9	3
375	5	25	500	1875	38.354	269.2	7
375	5	50	625	1875	21.849	494.6	12
375	5	75	750	1875	21.524	720	17
375	5	100	875	1875	15.334	945.4	103
375	25	5	500	9375	282.218	1254.4	82
375	25	25	1000	9375	166.368	2149.376	168
375	25	50	1625	9375	106.861	3268.608	252
375	25	75	2250	9375	87.165	4388.864	357
375	25	100	2875	9375	74.644	5509.12	448
375	50	5	625	18750	375	4534.272	301
375	50	25	1625	18750	295.517	6327.296	398
375	50	50	2875	18750	208.073	8566.784	461
375	50	75	4125	18750	177.186	10803.2	620
375	50	100	5375	18750	151.818	13045.76	767
375	75	5	750	28125		bounded	
375	75	25	2250	28125		bounded	
375	75	50	4125	28125		bounded	
375	75	75	6000	28125		bounded	
375	75	100	7875	28125		bounded	
375	100	5	875	37500		bounded	
375	100	25	2875	37500		bounded	
375	100	50	5375	37500		bounded	
375	100	75	7875	37500		bounded	
375	100	100	10375	37500		bounded	

A.2 ILP

The following table shows the raw results of every executed LP relaxation test configuration. For infeasible test configurations, the optimal value shows GLPK's suboptimal and expected optimal solution in the following form: suboptimal <= optimal.

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
25	5	1	30	125	25	0.589	0.744
25	5	5	50	125		0.995	0.894
25	5	25	150	125		2.93	3.368
25	5	50	275	125		5.252	1.435
25	5	75	400	125		7.573	1.776
25	5	100	525	125		9.764	1.028
25	25	1	50	625	25	6.157	0.885
25	25	5	150	625		8.144	1.172
25	25	25	650	625		17.8	1.977
25	25	50	1275	625		29.31	2.274
25	25	75	1900	625		41.27	2.783
25	25	100	2525	625		52.71	23.95
50	5	1	55	250	33	1.533	0.946
50	5	5	75	250		11.32	1103.8
50	5	25	175	250	22		bounded
50	5	50	300	250	10	18.34	342.2
50	5	75	425	250	10	35.55	1290.3
50	5	100	550	250	6	27	1.951
50	25	1	75	1250	50	21.56	1.376
50	25	5	175	1250		26.86	2.143
50	25	25	675	1250		53.23	4.819
50	25	50	1300	1250	45		bounded
50	25	75	1925	1250	40 <= 50		bounded
50	25	100	2550	1250	32 <= 45		bounded
50	50	1	100	2500	50	79.66	2.804
50	50	5	300	2500		90.44	5.238
50	50	25	1300	2500		141.8	11.76
50	50	50	2550	2500		206.9	16
50	50	75	3800	2500		271.6	17.63
50	50	100	5050	2500		332.1	39.53
75	5	1	80	375	47	2.925	1.909
75	5	5	100	375	37	5.56	10.97
75	5	25	200	375	14	23.37	548.8
75	5	50	325	375	11	32.05	347.8
75	5	75	450	375	9	46.22	471.2
75	5	100	575	375	6	61.56	305.8
75	25	1	100	1875	75	46.33	1.832
75	25	5	200	1875	75	56.43	3.627
75	25	25	700	1875	71		bounded
75	25	50	1325	1875	51		bounded
75	25	75	1950	1875	37 <= 53		bounded
75	25	100	2575	1875	35 <= 54		bounded
75	50	1	125	3750	75	174.5	5.433
75	50	5	325	3750	75	194.5	10.18
75	50	25	1325	3750	75	295.5	27.5
75	50	50	2575	3750	75	418.7	719.3
75	50	75	3825	3750			bounded
75	50	100	5075	3750	63		bounded
75	75	1	150	5625	75	384.6	8.387
75	75	5	450	5625	75	414.8	23.11
75	75	25	1950	5625	75	566.3	38.4
75	75	50	3825	5625	75	753.3	59
75	75	75	5700	5625	75	927.3	78.77
75	75	100	7575	5625	75	1111.04	97.83
100	5	1	105	500	61	4.73	1.707
100	5	5	125	500			bounded
100	5	25	225	500	15 <= 17		bounded
100	5	50	350	500	14	69.1	4341.6
100	5	75	475	500	7	87.93	1425.5
100	5	100	600	500	7	91.81	512.6
100	25	1	125	2500	100	80.52	5.468

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
100	25	5	225	2500		96.88	6.781
100	25	25	725	2500			bounded
100	25	50	1350	2500	54 <= 77		bounded
100	25	75	1975	2500	47 <= 70		bounded
100	25	100	2600	2500	37 <= 58		bounded
100	50	1	150	5000	100	306.3	22.81
100	50	5	350	5000	100	338.2	15.43
100	50	25	1350	5000		501.2	40.11
100	50	50	2600	5000			bounded
100	50	75	3850	5000			bounded
100	50	100	5100	5000	66 <= 100		bounded
100	75	1	175	7500	100	677.1	19.36
100	75	5	475	7500	100	725.3	27.64
100	75	25	1975	7500	100	964.2	87.89
100	75	50	3850	7500	100	1260.544	168.3
100	75	75	5725	7500	94		bounded
100	75	100	7600	7500	89		bounded
100	100	1	200	10000	100	1192.96	33.8
100	100	5	600	10000	100	1257.472	42.99
100	100	25	2600	10000	100	1575.936	120.7
100	100	50	5100	10000	100	1971.2	229.7
100	100	75	7600	10000	100	2369.536	324.5
100	100	100	10100	10000	100	2767.872	342.4
125	5	1	130	625	63	6.954	2.034
125	5	5	150	625			bounded
125	5	25	250	625			bounded
125	5	50	375	625	16 <= 18		bounded
125	5	75	500	625	9	172.4	11426.4
125	5	100	625	625	8	144.2	1738.2
125	25	1	150	3125	125	124	9.462
125	25	5	250	3125		148	20.62
125	25	25	750	3125			bounded
125	25	50	1375	3125	57		bounded
125	25	75	2000	3125	48 <= 71		bounded
125	25	100	2625	3125	39 <= 59		bounded
125	50	1	175	6250	125	474.3	18.28
125	50	5	375	6250	125	521.8	27.24
125	50	25	1375	6250	122		bounded
125	50	50	2625	6250	105 <= 125		bounded
125	50	75	3875	6250	81 <= 125		bounded
125	50	100	5125	6250	70 <= 122		bounded
125	75	1	200	9375	125	1051.648	36.4
125	75	5	500	9375	125	1122.304	42.59
125	75	25	2000	9375	125	1474.56	193.9
125	75	50	3875	9375	122		bounded
125	75	75	5750	9375	107		bounded
125	75	100	7625	9375			bounded
125	100	1	225	12500	125	1854.464	62.2
125	100	5	625	12500	125	1949.696	105.5
125	100	25	2625	12500	125	2418.688	221.9
125	100	50	5125	12500	125	3003.392	545.7
125	100	75	7625	12500	124		bounded
125	100	100	10125	12500	115		bounded
125	125	125	15750	15625		6522.88	657.5
150	5	1	155	750	83	9.806	5.941
150	5	5	175	750			bounded
150	5	25	275	750	26 <= 28		bounded
150	5	50	400	750	20 <= 23		bounded
150	5	75	525	750	9 <= 10		bounded
150	5	100	650	750	9	181.1	2214.4
150	25	1	175	3750	150	176.9	11
150	25	5	275	3750		210.1	41.59

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
150	25	25	775	3750	93 <= 121		bounded
150	25	50	1400	3750	65 <= 93		bounded
150	25	75	2025	3750	49 <= 73		bounded
150	25	100	2650	3750	42 <= 65		bounded
150	50	1	200	7500	150	679.3	22.4
150	50	5	400	7500	150	745	44.78
150	50	25	1400	7500	140 <= 150		bounded
150	50	50	2650	7500	111 <= 150		bounded
150	50	75	3900	7500	88 <= 143		bounded
150	50	100	5150	7500	69 <= 126		bounded
150	75	1	225	11250	150	1507.328	57.38
150	75	5	525	11250	150	1607.68	76.58
150	75	25	2025	11250	150	2094.08	356.7
150	75	50	3900	11250	134		bounded
150	75	75	5775	11250	115 <= 150		bounded
150	75	100	7650	11250	102 <= 150		bounded
150	100	1	250	15000	150	2662.4	99.58
150	100	5	650	15000	150	2793.472	149.4
150	100	25	2650	15000	150	3443.712	465.8
150	100	50	5150	15000		4250.624	2135.1
150	100	75	7650	15000	137 <= 150		bounded
150	100	100	10150	15000	125 <= 150		bounded
150	150	100	15150	22500		10772.48	1538.9
150	150	125	18900	22500			bounded
150	150	150	22650	22500			bounded
175	5	1	180	875	91	12.8	4.126
175	5	5	200	875	65 <= 68		bounded
175	5	25	300	875	25 <= 28		bounded
175	5	50	425	875	13 <= 14		bounded
175	5	75	550	875	10 <= 11		bounded
175	5	100	675	875	6 <= 7		bounded
175	25	1	200	4375	175	239.1	14.27
175	25	5	300	4375			bounded
175	25	25	800	4375	90 <= 118		bounded
175	25	50	1425	4375	72 <= 102		bounded
175	25	75	2050	4375	53 <= 80		bounded
175	25	100	2675	4375	40 <= 66		bounded
175	50	1	225	8750	175	920.9	40.56
175	50	5	425	8750	175	1007.616	72.96
175	50	25	1425	8750	156 <= 175		bounded
175	50	50	2675	8750	111 <= 170		bounded
175	50	75	3925	8750	90 <= 152		bounded
175	50	100	5175	8750	66 <= 123		bounded
175	75	1	250	13125	175	2045.952	105.3
175	75	5	550	13125	175	2176	124.3
175	75	25	2050	13125	175	2818.048	501.9
175	75	50	3925	13125	151 <= 175		bounded
175	75	75	5800	13125	126 <= 175		bounded
175	75	100	7675	13125	99 <= 175		bounded
175	100	1	275	17500	175	3614.72	170.5
175	100	5	675	17500	175	3787.776	271.8
175	100	25	2675	17500	175	4644.864	763.8
175	100	50	5175	17500	167 <= 175		bounded
175	100	75	7675	17500	147 <= 175		bounded
175	100	100	10175	17500	136 <= 175		bounded
175	125	125	15800	21875			bounded
200	5	1	205	1000	109	16.95	14.52
200	5	5	225	1000	62 <= 65		bounded
200	5	25	325	1000	30 <= 33		bounded
200	5	50	450	1000	15 <= 18		bounded
200	5	75	575	1000	13 <= 14		bounded
200	5	100	700	1000	8	287.1	53.32

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
200	25	1	225	5000	200	310.5	15.46
200	25	5	325	5000	195 <= 200		bounded
200	25	25	825	5000	106 <= 139		bounded
200	25	50	1450	5000	76 <= 105		bounded
200	25	75	2075	5000	38 <= 64		bounded
200	25	100	2700	5000	42 <= 66		bounded
200	50	1	250	10000	200	1199.104	60.63
200	50	5	450	10000	200	1309.696	76.83
200	50	25	1450	10000	171 <= 200		bounded
200	50	50	2700	10000	118 <= 180		bounded
200	50	75	3950	10000	92 <= 156		bounded
200	50	100	5200	10000	74 <= 130		bounded
200	75	1	275	15000	200	2666.496	157.5
200	75	5	575	15000	200	2831.36	173.3
200	75	25	2075	15000	189 <= 200		bounded
200	75	50	3950	15000	163 <= 200		bounded
200	75	75	5825	15000	127 <= 200		bounded
200	75	100	7700	15000	98 <= 190		bounded
200	100	1	300	20000	200	4712.448	266.1
200	100	5	700	20000	200	4932.608	267.1
200	100	25	2700	20000	200	6030.336	1072
200	100	50	5200	20000	185 <= 200		bounded
200	100	75	7700	20000	157 <= 200		bounded
200	100	100	10200	20000			bounded
225	5	1	230	1125	96	20.34	5.95
225	5	5	250	1125	62 <= 64		bounded
225	5	25	350	1125	25 <= 28		bounded
225	5	50	475	1125	130 <= 150		bounded
225	5	75	600	1125	140 <= 160		bounded
225	5	100	725	1125	10 <= 11		bounded
225	25	1	250	5625	225	391.6	31.44
225	25	5	350	5625	200 <= 220		bounded
225	25	25	850	5625	115 <= 148		bounded
225	25	50	1475	5625	64 <= 93		bounded
225	25	75	2100	5625	50 <= 75		bounded
225	25	100	2725	5625	38 <= 65		bounded
225	50	1	275	11250	225	1514.496	73.56
225	50	5	475	11250	225	1651.712	149
225	50	25	1475	11250	193 <= 225		bounded
225	50	50	2725	11250	120 <= 184		bounded
225	50	75	3975	11250	93 <= 154		bounded
225	50	100	5225	11250	80 <= 137		bounded
225	75	1	300	16875	225	3368.96	182
225	75	5	600	16875	225	3574.784	250
225	75	25	2100	16875	215 <= 225		bounded
225	75	50	3975	16875	170 <= 225		bounded
225	75	75	5850	16875	129 <= 222		bounded
225	75	100	7725	16875	103 <= 194		bounded
225	100	1	325	22500	225	5955.584	308.1
225	100	5	725	22500	225	6228.992	396.6
225	100	25	2725	22500	225	7590.912	2963.5
225	100	50	5225	22500	197 <= 225		bounded
225	100	75	7725	22500	160 <= 225		bounded
225	100	100	10225	22500	138 <= 225		bounded
250	5	1	255	1250	128	25.09	15.33
250	5	5	275	1250	61 <= 64		bounded
250	5	25	375	1250	21 <= 23		bounded
250	5	50	500	1250	16	257.2	9450
250	5	75	625	1250	11 <= 14		bounded
250	5	100	750	1250	10	458.9	7354.8
250	25	1	275	6250	250	481.9	41.54
250	25	5	375	6250	208 <= 233		bounded

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
250	25	25	875	6250	98 <= 130		bounded
250	25	50	1500	6250	67 <= 97		bounded
250	25	75	2125	6250	53 <= 83		bounded
250	25	100	2750	6250	39 <= 66		bounded
250	50	1	300	12500	250	1865.728	113.9
250	50	5	500	12500	250	2033.664	155.8
250	50	25	1500	12500	204 <= 250		bounded
250	50	50	2750	12500	125 <= 192		bounded
250	50	75	4000	12500	99 <= 167		bounded
250	50	100	5250	12500	73 <= 134		bounded
250	75	1	325	18750	250	4153.344	260.7
250	75	5	625	18750	250	4403.2	339.2
250	75	25	2125	18750	227 <= 250		bounded
250	75	50	4000	18750	164 <= 250		bounded
250	75	75	5875	18750	133 <= 231		bounded
250	75	100	7750	18750	112 <= 203		bounded
250	100	1	350	25000	250	7344.128	371.4
250	100	5	750	25000	250	7676.928	534.5
250	100	25	2750	25000	243 <= 250		bounded
250	100	50	5250	25000	204 <= 250		bounded
250	100	75	7750	25000	181 <= 250		bounded
250	100	100	10250	25000	140 <= 250		bounded
275	5	100	775	1375	9	660.2	25927.2
275	25	1	300	6875	261 <= 262		bounded
275	25	5	400	6875	199 <= 226		bounded
275	25	25	900	6875	113 <= 148		bounded
275	25	50	1525	6875	69 <= 101		bounded
275	25	75	2150	6875	49 <= 81		bounded
275	25	100	2775	6875	39 <= 68		bounded
275	50	1	325	13750	275	2254.848	139
275	50	5	525	13750	275	2454.528	225.3
275	50	25	1525	13750	200 <= 267		bounded
275	50	50	2775	13750	120 <= 189		bounded
275	50	75	4025	13750	99 <= 168		bounded
275	50	100	5275	13750	77 <= 142		bounded
275	75	1	350	20625	275	5020.672	327
275	75	5	650	20625	275	5318.656	415.9
275	75	25	2150	20625	245 <= 275		bounded
275	75	50	4025	20625	171 <= 272		bounded
275	75	75	5900	20625	135 <= 241		bounded
275	75	100	7775	20625	113 <= 209		bounded
275	100	1	375	27500		bounded	
275	100	5	775	27500		bounded	
275	100	25	2775	27500		bounded	
275	100	50	5275	27500		bounded	
275	100	75	7775	27500		bounded	
275	100	100	10275	27500		bounded	
300	5	1	305	1500	113	35.17	15.51
300	5	5	325	1500	60 <= 61		bounded
300	5	25	425	1500	27 <= 31		bounded
300	5	50	550	1500	17 <= 19		bounded
300	5	75	675	1500	13 <= 15		bounded
300	5	100	800	1500	8	612.1	488.5
300	25	1	325	7500	262 <= 263		bounded
300	25	5	425	7500	224 <= 253		bounded
300	25	25	925	7500	112 <= 145		bounded
300	25	50	1550	7500	72 <= 106		bounded
300	25	75	2175	7500	52 <= 81		bounded
300	25	100	2800	7500	42 <= 70		bounded
300	50	1	350	15000	300	2679.808	172.7
300	50	5	550	15000	299 <= 300		bounded
300	50	25	1550	15000	182 <= 255		bounded

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
300	50	50	2800	15000	133 <= 206		bounded
300	50	75	4050	15000	92 <= 161		bounded
300	50	100	5300	15000	80 <= 145		bounded
300	75	1	375	22500	300	5968.896	332.1
300	75	5	675	22500	300	6319.104	466.4
300	75	25	2175	22500	251 <= 300		bounded
300	75	50	4050	22500	181 <= 282		bounded
300	75	75	5925	22500	141 <= 244		bounded
300	75	100	7800	22500	110 <= 212		bounded
300	100	1	400	30000	300	bounded	
300	100	5	800	30000	300	bounded	
300	100	25	2800	30000	286 <= 300	bounded	
300	100	50	5300	30000		bounded	
300	100	75	7800	30000		bounded	
300	100	100	10300	30000		bounded	
325	5	1	330	1625	157	41.33	48.06
325	5	5	350	1625	79 <= 81		bounded
325	5	25	450	1625	28 <= 32		bounded
325	5	50	575	1625	14 <= 16		bounded
325	5	75	700	1625	12 <= 13		bounded
325	5	100	825	1625	10 <= 11		bounded
325	25	1	350	8125	299	816.2	104.4
325	25	5	450	8125	237 <= 262		bounded
325	25	25	950	8125	109 <= 145		bounded
325	25	50	1575	8125	67 <= 97		bounded
325	25	75	2200	8125	50 <= 82		bounded
325	25	100	2825	8125	45 <= 76		bounded
325	50	1	375	16250	325	3141.632	214.5
325	50	5	575	16250	324 <= 325		bounded
325	50	25	1575	16250	203 <= 277		bounded
325	50	50	2825	16250	128 <= 198		bounded
325	50	75	4075	16250	100 <= 172		bounded
325	50	100	5325	16250	83 <= 147		bounded
325	75	1	400	24375	325	6999.04	531.8
325	75	5	700	24375	325	7407.616	646.3
325	75	25	2200	24375	0 <= 325		bounded
325	75	50	4075	24375	193 <= 297		bounded
325	75	75	5950	24375	136 <= 236		bounded
325	75	100	7825	24375		bounded	
325	100	1	425	32500		bounded	
325	100	5	825	32500	325	bounded	
325	100	25	2825	32500		bounded	
325	100	50	5325	32500		bounded	
325	100	75	7825	32500		bounded	
325	100	100	10325	32500		bounded	
350	5	1	355	1750	101	46.38	6.638
350	5	5	375	1750	68 <= 72		bounded
350	5	25	475	1750	21 <= 24		bounded
350	5	50	600	1750	19 <= 21		bounded
350	5	75	725	1750	13 <= 15		bounded
350	5	100	850	1750	8 <= 11		bounded
350	25	1	375	8750	311	946.9	8190
350	25	5	475	8750	241 <= 267		bounded
350	25	25	975	8750	100 <= 133		bounded
350	25	50	1600	8750	75 <= 110		bounded
350	25	75	2225	8750	50 <= 80		bounded
350	25	100	2850	8750	44 <= 74		bounded
350	50	1	400	17500	350	3640.32	291.2
350	50	5	600	17500	342 <= 350		bounded
350	50	25	1600	17500	204 <= 276		bounded
350	50	50	2850	17500	128 <= 198		bounded
350	50	75	4100	17500	98 <= 166		bounded

n	m	k	# constraints	# dec. vars	opt.val.	MBs	Exec. time
350	50	100	5350	17500	75 <= 142		bounded
350	75	1	425	26250	350	bounded	
350	75	5	725	26250		bounded	
350	75	25	2225	26250		bounded	
350	75	50	4100	26250		bounded	
350	75	75	5975	26250		bounded	
350	75	100	7850	26250		bounded	
350	100	1	450	35000	350	bounded	
350	100	5	850	35000		bounded	
350	100	25	2850	35000		bounded	
350	100	50	5350	35000		bounded	
350	100	75	7850	35000		bounded	
350	100	100	10350	35000		bounded	
375	5	1	380	1875	116	53.04	11.11
375	5	5	400	1875			bounded
375	5	25	500	1875	32 <= 37		bounded
375	5	50	625	1875	17 <= 18		bounded
375	5	75	750	1875	15 <= 19		bounded
375	5	100	875	1875	90 <= 110		bounded
375	25	1	400	9375	299 <= 300		bounded
375	25	5	500	9375	253 <= 282		bounded
375	25	25	1000	9375	129 <= 165		bounded
375	25	50	1625	9375	70 <= 105		bounded
375	25	75	2250	9375	52 <= 84		bounded
375	25	100	2875	9375	41 <= 71		bounded
375	50	1	425	18750	375	4175.872	290.1
375	50	5	625	18750	362 <= 375		bounded
375	50	25	1625	18750	219 <= 295		bounded
375	50	50	2875	18750	134 <= 206		bounded
375	50	75	4125	18750	103 <= 175		bounded
375	50	100	5375	18750	80 <= 146		bounded
375	75	1	450	28125	375	bounded	
375	75	5	750	28125		bounded	
375	75	25	2250	28125		bounded	
375	75	50	4125	28125		bounded	
375	75	75	6000	28125		bounded	
375	75	100	7875	28125		bounded	
375	100	1	475	37500		bounded	
375	100	5	875	37500		bounded	
375	100	25	2875	37500		bounded	
375	100	50	5375	37500		bounded	
375	100	75	7875	37500		bounded	
375	100	100	10375	37500		bounded	

Glossary

Term	Definition
Branch and Bound	An algorithm to solve ILPs by partitioning the feasible region
Branch and Cut	An algorithm to solve ILPs by mixing Branch and Bound and Cutting Planes
cut	Cuts in the cutting plane algorithm are Gomory-Chvâtal cuts
Cutting Planes	An algorithm to solve ILPs by cutting away fractional parts of the feasible region
Feasible region	A n-dimensional space of real numbers that satisfy every constraint
Geometry	Visual representation of the feasible region
Infeasible solution	Infeasible problem is a problem where no solution could be found for the given constraints
Integer feasible region	All integer points in the feasible region, a subset
Model	A linear programming model that contains a objective function and constraints
Node	A processing unit in form of a computer, usually a server or a cloud server
Optimal solution	No better solution can be found in order to satisfy the LP constraints
Task	Problem that is required to be solved computational on a node
Timestamp	A timestamp is a positive integeral number representing a point in time. Unix time, or POSIX time, is an example of a timestamp. Unix time started on 1 January 1970. Main difference is that unix time starts with a zero.
Unbounded solution	The feasible region is not bounded by a constraint in any direction
Variable	Processing a task requires certain capacity of a node, available and required capacity can be defined as one or more variables. Hard disk usage, memory usage, and processing power are examples of variables

Acronyms

Term	Definition
cpu	Central Processing Unit
CSP	Constraint Satisfaction Problem
ILP	Integer Linear Program / Problem
LP	Linear Program / Problem
s.t.	subject to

Bibliography

[1] E.M.L. Beale and J.J.H. Forrest. Global optimization using special ordered sets. *Mathematical Programming 10*, pages 52--69, 1976.

[2] Vaŝek Chvátal. *Linear Programming*. Series of books in the mathematical sciences. W. H. Freeman and Company, 1983.

[3] Keith Gilbertson. Process Accounting. http://www.linuxjournal.com/article/6144, December 2002.

[4] Tim Jones (IBM). Scheduling in hadoop. http://www.ibm.com/developerworks/library/os-hadoop-scheduling/.

[5] Janies V. Koch. A linear programming model of resource allocation in a university. *DECISION SCIENCES, Vol. 4, Illinois State University*, 1973.

[6] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly 2*, 1955.

[7] Dongning Liang, Pei-Jung Ho, and Bao Liu. Scheduling in distributed systems. *Department of Computer Science and Engineering, University of California, San Diego*.

[8] Andrew Makhorin. Glpk (gnu linear programming kit). http://www.gnu.org/software/glpk/.

[9] Sayanta Mallick and Gaétan Hains. A Resource prediction model for virtualization servers. *Laboratory for Algorithmics, Complexity and Logic (LACL)*, December 2011.

[10] John E. Mitchell. Branch-and-Cut algorithms for combinatorial optimization problems. *Handbook of Applied Optimization, Oxford University Press*, 2000.

[11] Sriram Sankaranarayanan and Shalom D. Ruben. Coursera MOOC, Linear and Integer Programming. https://www.coursera.org/course/linearprogramming.

[12] The Apache Software Foundation. Apache hadoop. http://hadoop.apache.org/.

[13] Ubuntu / Canonical. Releasenotes Ubuntu 14.04. https://wiki.ubuntu.com/TrustyTahr/ReleaseNotes.

[14] Valgrind. Massif: a heap profiler. http://valgrind.org/docs/manual/ms-manual.html.

[15] Pascal Van Hentenryck. Coursera MOOC, Discrete Optimization. https://www.coursera.org/course/optimization.

[16] Robert J. Vanderbei. *Linear Programming, Foundations and Extensions, 3th*. Springer, 2008.