Bachelor Thesis

Degree Project in Information and Software Systems

Memory Management for Gecode

Author:
Zandra Norman
znorman@kth.se
TRITA-ICT-EX-2012:143

Supervisor and examiner:
Christian Schulte

June 27, 2012

Abstract

This report presents an evaluation of the memory management in the constraint programming platform Gecode.

Constraint programming is developed to find solutions to constraint based problems. Gecode is a tool-kit to help doing so. In the search for solutions a search tree is used. In this search tree every node handles its own heap, which makes the memory management complex.

The evaluation is made in respect of some core variables used in the source code. These variables are used to decide the size of the heap chunk allocations.

The evaluation shows that it is important that the heap chunk allocation size is able to grow to adapt to the needs of every specific program, and shrink to be able to fit every specific node in the search tree used. This decreasing functionality did not work due to a bug, but this was easily repaired.

In the end of the report some recommended values for the core variables and an explanation of how to fix the decrease bug is found. The improvements after these recommendations are also illustrated and explained.

Sammanfattning

Denna rapport presenterar en utvärdering av minneshanteringen i villkorsprogrammeringsplattformen Gecode.

Villkorsprogrammering är utvecklad för att kunna hitta lösningar på villkorsbaserade problem. Gecode är en verktygssamling för att underlätta detta. I sökningen efter lösningar används ett sökträd. I detta sökträd hanterar alla noder sin egen heap, vilket gör minneshanteringen komplex.

Utvärderingen är gjord i respekt av kärnvariabler som används i källkoden. Dessa variabler används för att bestämma storleken för allokeringar från heapen.

Utvärderingen visar att det är viktigt att allokeringsstorleken för heap-delar kan växa för att anpassas för varje specifikt program, och minska för att anpassas efter varje specifik nod i sökträdet som används. Denna förminskande funktionalitet fungerade inte, på grund av en bug som enkelt lagades.

I slutet av rapporten finns rekommenderade värden för kärnvariablerna listade, samt en förklaring för hur förminsknings-buggen kan lagas. Förbättringarna efter dessa rekommendationer är även illustrerade och förklarade.

Contents

1 Introduction

Today's companies grow and spread across the world with the globalization. To keep these companies organized many types of technologies are required.

One important aspect for these growing companies is to optimize their work, to get as profitable as they can. An airline company, for instance, wants to fill and schedule as many flights as possible with as little staff and service costs as possible. These types of optimizations get very complex when the companies grow, so the need for technical aid in this area is increasing. A suitable programming paradigm is available to help with this kind of optimizations. It is called constraint programming and one of its purpose is to find solutions for complex problems that are based on constraints[5].

Constraint programming has a searching technique that requires some extra consideration about the computer memory management. This search should be efficient both in terms of runtime and memory consumption. One of today's best platforms for constraint programming is the generic constraint development environment Gecode[7].

In Gecode a special memory management part is implemented to handle these requirements. This part of Gecode has never been properly evaluated, so that is the goal with this project.

1.1 Background Knowledge

To understand this thesis it is important to have some knowledge about computer memory management, constraint programming and Gecode. These parts will be explained further in Chapter 2 and 3, but a short and somewhat simplified introduction to them all follows below.

1.1.1 Memory Management in Computer Systems

One of the major components in a computer is memory. In an ideal computer the memory would be incredibly fast, big and cheap. Unfortunately that is not possible in the real world where the memory handling is much slower than executing the actual program instructions, more expensive the faster it is, and not infinitely big. Therefore a program is very dependent on its memory management. A poorly optimized program uses more memory, which is expensive, requires more space and will run longer than necessary[2, **?**].

1.1.2 Constraint Programming

Constraint programming is a way to solve problems with constraints. A commonly known constraint based problem is Sudoku. It has the constraints that no two elements in a row or column can have the same value, and the same constraint holds for the nine regions of three times three boxes. Sudoku is explained in more detail in chapter 2.2.

The solving of a constraint based problem contains two main steps: modeling and solving. The first one, modeling, is to formulate the initial problem, including determining the variables, their domains and constraints. The second step, solving, can start when the modeling step is finished.

In the beginning all possible cases, i.e. all possible combinations of values for the different variables, are viewed as potential solutions. Every case stays in this potential solution collection until proven not possible. If a constraint declares that a value for a variable is not possible it is removed from the variable list. This is called propagation. The search is made by creating sub problems to cover every possible case. A sub problem evolves to other sub problems and this creates a search tree. If one node does not lead to a solution it is removed from the collection of potential solutions. When the search has finished the cases which remain in the possible solution collection are the final answers. The result can be either one possible solution, many or none[5].

1.1.3 Gecode

Different tool-kits have been developed to simplify problem solving with constraint programming. One is Gecode, which is mainly developed by Christian Schulte, KTH, Mikael Lagerkvist, KTH, and Guido Tack, Monash University. In 2011 Gecode won all the categories in the MiniZinc Challenge, which aim to compare various constraint solving technology[7].

To solve a constraint programming *model* Gecode uses a search tree which is based on depth-first search. Here it has to be possible to return to previous states if no solution was found in one direction, or if more solutions are needed. Gecode clones, i.e. copies, every state before moving further down one way in the search tree to make that possible. When the program is returning, the part it is returning from is no longer needed. To make it easy to delete the memory a node has consumed when it is not in use anymore, every node in the tree has its own heap[1].

This deletion is important, because the number of nodes in a constraint programming tree can be extremely large. If the old nodes are not deleted it would not be possible to run such a search on today's computers since the memory space on them would not be enough[4].

1.2 Purpose

The memory management in Gecode has never been properly evaluated. The purpose of this project is therefore to perform an evaluation and investigate if it is possible to improve the management. If an improvement is possible also an investigation of how to do it will be made.

1.3 Project Goals

The goals of this project are listed below.

- Evaluate the current core variables for the memory management in Gecode
- Optimize the management in respect of the constant values

1.4 Report Overview

This is a short overview of the different sections of the report and a short explanation of how to read it.

The report contains sections that all are essential to understand the thesis. The two upcoming chapters give some background knowledge, and then the evaluation methodology follows. After this the results and findings are visualized and explained and finally reflections around the thesis results and recommendations for the future memory management.

1.4.1 Background (Chapter 2 and 3)

Chapter two and three contain some background information about memory management, constraint programming, Gecode and the specific memory management in it.

1.4.2 Evaluation Methodology (Chapter 4)

The next part contains information about what was measured during the tests, why and how. It also explains how to read the result presentation and figures.

1.4.3 Test Methods and Results (Chapter 5 and 6)

In these chapters the specific tests and their results are presented, considered and explained. Many graphs are used for visualization to get the results as clear as possible.

1.4.4 Reflections and Recommendations (Chapter 7 and 8)

The final part clarifies what can be learned from this thesis. It gives some concluding comments, reflections and recommendations for the future memory management in Gecode.

1.4.5 How to Read the Report

The report contains terms that might be hard to keep in memory throughout the entire reading, but they are all important to understand. They are highlighted like *this*. Pay some extra attention when they are explained in the text.

Some variable names from the source code of Gecode are also used later in the report. Each of these variables contains one or more underscores, so note that when an underscore turns up in a word this word is a variable name.

Another important thing to remember while reading this report is to look at the illustrating figures. Use them as a base while reading, rather than the other way around.

2 Background

To understand this thesis it is essential to have some knowledge about fundamental computer memory management. It is also important to know some things about constraint programming and finally a little bit about how the constraint programming solver Gecode works. Some of the information below is simplified and focused on explaining the basics rather than introducing all details. It is shaped to make the evaluation understandable.

2.1 Computer Memory Management

A computer system is tied to the hardware in the computer. The memory is one of these hardware components. In today's computers the memory is often slowing programs down, since the memory access speed cannot be compared with how fast instructions on a computer are run. Therefore the program execution time can be depending on their memory management. If it is bad, the program might take more time than necessary or require extra memory space[2].

All programs require different amounts of memory, but they will all need some. Therefore the possibility to allocate memory portions to programs at their requests is important. In order not to lose memory in so called memory leaks it is also important to provide methods to free, i.e. give back, earlier allocated memory portions[2].

This memory allocation is to find a memory chunk that is big enough for the request and then grab, reserve, and give it to the requesting program. This might sound simple, but has some fragmentation problems. A fragmentation problem can occur when a bigger chunk is freed and replaced with a smaller, which creates a hole between the new one and the chunk closest beneath. Figure 1 below shows an overview of what such a problem can look like. In the first situation the memory is very well used but if memory chunk 3 is released and replaced with chunk 7, chunk 2 replaced with 6 and then chunk 1 with 5 the second situation occurs. Here memory chunk 8 is requested but cannot be met since it does not fit in any free space even though there is enough free memory in total[2].

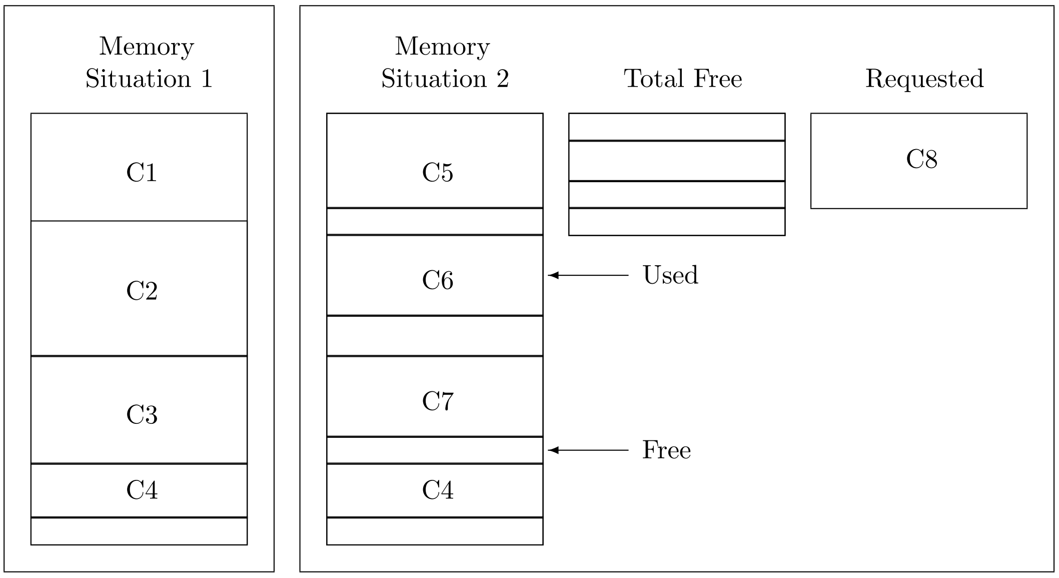

Figure 1: *Memory fragmentation problem - C1 stands for memory chunk 1, C2 for chunk 2 and so on.*

Memory fragmentation can occur both within a program, so called internal fragmentation, and in external situations. The operating system usually considers the external fragmentation. How this is done depends on the operating system, but since it is very important for the performance it is thought through[2].

Many Windows users have probably used the program defrag which makes the files contiguous, as in situation 1 in figure 1, by moving them around. Those who have tried this knows that it takes quite some time[2].

It is impossible to completely avoid memory fragmentation without spending more time on the memory operations than on the actual program running. It is possible, however, to decrease the amount of fragmentation if the memory management is efficiently implemented. How this should be implemented depends on what the program will do, so it will not be discussed further here. The important thing to remember is that it is not a simple process and that a program should avoid fragmentations and wasted time. It is, in other words, important not to do too many and small or too few and big memory allocation requests. These requests have to be well chosen to get an efficient memory handling[2].

2.1.1 Heap and Stack

In computer memory management both a stack and a heap can be used. Their functionalities are different which is explained below.

A stack grows and shrinks and can be compared to a stack of plates. When a plate is added it is put on the top of the stack and when one is taken it is also taken from the top. It is the same way with a computer memory stack but usually upside down where the bottom is used instead of the top. This also means that the last added thing is the first thing to be removed. The stack is generally used for local variables since they are only needed for a short amount of time[3].

To store the global variables the heap is used. From the heap they can be accessed randomly, at any time. When using the heap the memory first has to be allocated with a key word procedure call. What this word is depends on the programming language used[2].

The biggest difference between the two is their different how to get hold of the memory and their lifetimes which affects how they are used. The stack is suitable when memory is temporarily needed since it is easier to get but cannot be used for long, and the heap is better for more contiguous memory requirements.

Gecode is implemented in the C++ programming language. In C++ the memory stack is managed automatically for local variables, but the heap memory management has to be implemented in the program. This makes it very important to keep track of the heap memory portions allocated from the heap and delete them when they are no longer needed. There is, in other words, no garbage collection like in, for example, Java. If all pointers to a heap chunk are lost before it is deleted, that memory part will also be lost for further usage. It cannot be accessed again without any pointer to it and it cannot be allocated again since it was not freed and therefore is still marked as reserved. If this happens the program has so called memory leaks[8].

In order to keep track of these heap memory portions so called container classes are very useful. They can be compared to containers in real life, like for example a box that contains cereals or the cover and binding around a book. These containers makes it possible to work with the objects they contain, and just like in real life containers in programming helps to handle their content, in this case memory portions. In C++ a container class is designed to hold and organize a number of instances to other classes. Examples of programming containers are lists, queues, arrays and trees[8].

2.2 Constraint Programming

Constraint programming is a special type of programming that is developed to solve problems which depends on different requirements. Such a problem is limited by a number of constraints, and the goal with constraint programming is to find one or more solutions for the problem in which all these constraints are true.

This type of programming is very useful in, for instance, advanced scheduling, other optimizations and electrical engineering[5].

2.2.1 Example Problems

The easiest way to understand the fundamentals of constraint programming is to look at some typical examples of constraint programming problems. This gives a picture of what constraints means in this context and how constraint programming can help to solve this type of problems. Here a Sudoku and an airline scheduling example is presented.

Sudoku

Sudoku is nowadays a commonly known game that is based on logic. It is a Japanese puzzle which during the last couple of years has spread worldwide and now it often turns up in newspapers beside common crosswords and other puzzles. The game is based on a 9x9 grid with sub-grids of size 3x3 and it has three important constraints:

1. Every row must contain all digits between 1 and 9
   (a) i.e. no two blocks in any row can contain the same digit.
2. Every column must contain all digits between 1 and 9
   (a) i.e. no two blocks in any column can contain the same digit.
3. Every sub-grid must contain all digits between 1 and 9
   (a) i.e. no two blocks in any sub-grid can contain the same digit.

Below, in figure 2, a Sudoku example is illustrated. The first grid shows a start board and the second shows how the first value, 5, is placed in the left upper corner with help from the third Sudoku constraints. The third one shows the final solution.

Start Grid

	4	1			9	6		
8	2	3		6				9
9	6	7			5	4		2
					8			6
	5	8				9	1	
4			5					
2		5	1					
6				3		2		
		9	4			8		

First Value

5	4	1			9	6		
8	2	3		6				9
9	6	7			5	4		2
					8			6
	5	8				9	1	
4			5					
2		5	1					
6				3		2		
		9	4			8		

Solution

5	4	1	2	8	9	6	3	7
8	2	3	7	6	4	1	5	9
9	6	7	3	1	5	4	8	2
1	2	3	9	4	8	5	7	6
7	5	8	6	2	3	9	1	4
4	9	6	5	7	1	3	2	8
2	8	5	1	9	6	7	4	3
6	1	4	8	3	7	2	9	5
3	7	9	4	5	2	8	6	1

Figure 2: *Sudoku example*

As visible above, the solution can be created from the start board with help from the earlier explained constraints. The solution can be found by setting the values that can be set and then removing values that are no longer possible for a box, until the box only have one possible value left.

Airline Scheduling

Another example problem is the scheduling for an airline. Every flight must have staff and a working plane and they both have to be in the right place at the right time, have a certain amount of rest between flights and so on. Many factors can affect the scheduling, like for instance the fact that an airline operates on many airports, often in many different countries, someone can get sick, a plane might have to get serviced. Therefore it is important to have a good way to solve all possible problems on the way. Constraint programming is then a good option since one can take care of all the requirements and find a solution[4].

2.2.2 Problem Solving Procedure

To find solutions for a constraint based problem using constraint programming has two main steps, modeling and solution search, have to be made. The first step, modeling, is to formulate the initial problem, including deciding the variables, domains and constraints. It is also central to decide which or how many answers that are required. Some examples of different answer requirements are all, the best or the first.

When the above is decided and declared the solving step can start. The search, in turn, can also be divided into two steps: *propagation* and guessing. These two steps are repeated until the final solutions are found. With propagation, the first of these steps, means that all currently not possible values are removed. In the Sudoku example in figure 2 shows the first value, 5, is placed since it is the only possible value left in its square. That means that 1, 2, 3, 4, 6, 7, 8 and 9 were removed, i.e. propagated, from the possible values for that square since they are already represented in the sub grid. When all propagation that is possible to make has been done some guessing has to be made.

If the first propagation round is finished, i.e. when all the currently not possible values has been removed, without leading to a final solution the guessing will begin. The search is based on a binary search tree where the node in one direction contains a guessed value for one variable and the node in the other direction contains all other possible values for the variable. Figure 3 pictures an example of such a tree. Here a variable x has the possible values 1, 2 and 3. The guess in the first direction, left in this example, might be that x is 1. Hence the other node will cover the possibilities that it can be 2 or 3. The next step for the right node will then be to guess the next value, for example 3, and create two new sub-nodes, in this case the left tests if x can 3 and the right if it can be 2.

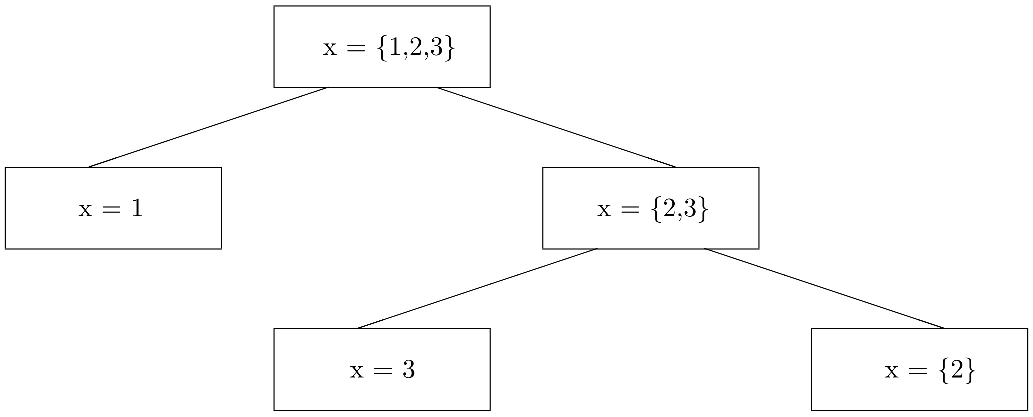

Figure 3: *Example of a constraint programming search tree*

The search is, with other words, made by creating sub problems to cover every possible case. A sub problem evolves to sub-sub problems and so on. When an end, i.e. tree leaf, is reached it can be either a solution to the *model* or show that this way was not a possibility. If it was not, that part of the sub tree can be *propagated* from the set of possible solutions. If it leads to a solution it will be declared as one. When the search has finished it is possible that many solutions to the problem was found.

Different techniques can be used for this search, but the most common are backtracking and branch and bound. Figure 4 shows a small classic backtracking tree. The latter is usually used when the best solution is wanted but is not important for this thesis and will not be discussed further here[5].

2.3 Gecode

Many different toolkits to facilitate constraint programming, such as Gecode, BProlog[9] and JaCoP (Java Constraint Programming Solver)[10], have been developed.

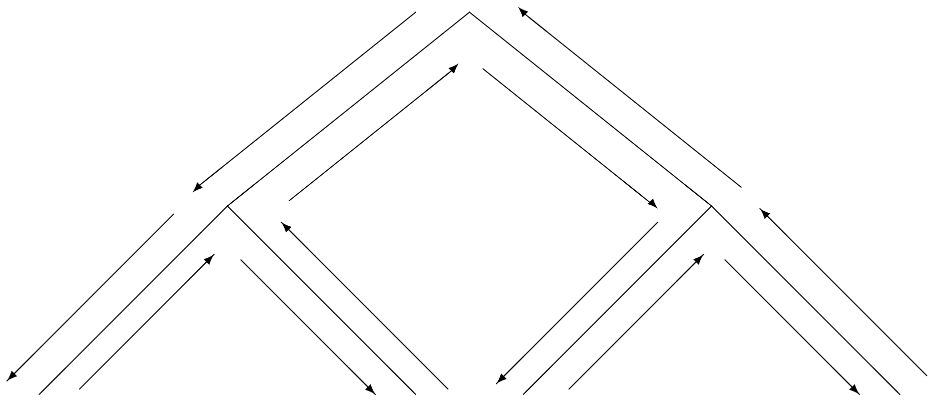

Figure 4: *Backtracking Graph*

Gecode is an open source, MIT licensed, generic constraint development environment. It uses the C++ library and supports backtracking, branch-and-bound and parallel search for solutions of constraint programming problems. It has thousands of users in different areas, for example in education, optimizations and advanced scheduling.

Gecode supports users on Linux (it is included in Debian, Ubuntu and FreeBSD), Windows and Mac[6].

Gecode has been in the front of Constraint programming tool-kits for several years. In the last four years it won all the categories in the MiniZinc Challenge which is based on the memory usage and runtime efficiency for constraint programming solvers[7].

2.3.1 Usage

Gecode is used to solve constraint programming problems in different situations. It is used for research where it can help to do experiments and comparison, for education in constraint programming and also by companies which have integrated the environment in their own products. An example is Quintiq who have integrated Gecode in their advanced scheduling system where they for instance make the truck scheduling for Wal-Mart[11][4].

2.3.2 Search Functionality

Since Gecode is a platform for constraint programming a search tree is used to find solutions to a constraint based problem. In this tree every node is represented by a *space*. A space contains the constraints left to fulfil, the variables and their domains and other important details to be able to find solutions. When one direction is chosen the current space is cloned to be able to return to previous states when needed[1].

The search ways in this tree can be incredibly many, which mean that also the number of spaces can be so. Since every space requires memory the total amount of memory required is often many times more than it is possible to provide for a program. Therefore Gecode only requires memory when it is needed (for the spaces that it is currently searching in) and releases it as soon as it no longer is. This means that a lot of memory is passed between the operating system and Gecode, which takes time. Gecode uses different memory chunk lists to decrease the amount of memory passing but the memory is a finite resource making it important that Gecode is not occupying too much of it[4].

The size of the spaces will usually shrink the deeper in the search tree they get, since many variables and constraints gets assigned or eliminated on the way there. The only things that might get bigger are the variables, even though they start with many possible values which is reduced until only one or no possible value is left. The reason for this is the technique used to keep track of the different possible values where it is central to make it easy to remove no longer possible values. Therefore a linked list is used, as in the example in figure 5[4].

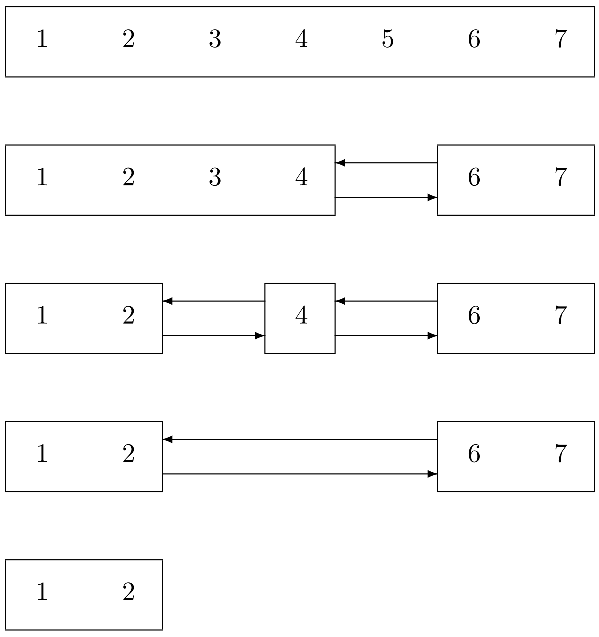

Figure 5: *Variable list illustration in different steps*

As in the example in figure 5 all possible values can be represented in only one node in the beginning. When a value is removed the list might be split, and hence pointers will be needed. Here, value 5 is removed between the first and second step, in step three the 3 is removed, then the 4 and finally the 6 and 7. Since the number of pointers might increase during the run, as between step one and three, the list might actually require more memory when values are removed. It is impossible to know if the memory space needed for the variable lists will grow or shrink in the next step. This is the only part of a space that might actually grow when moving down in the tree[4].

3 Memory Management in Gecode

In constraint programming the search is done using a search tree where the ability to return to old states and continue the search in another direction has a central role. To be able to do so Gecode clones the current state every time a choice is made, and when the search has reached an end it returns using backtracking.

In this chapter some variables will be explained in the text. They will be used by their source code names and they all contain at least one underscore, so when a word with underscores turns up, note that it is a variable.

3.1 Overview

The memory management part of Gecode can be divided into sections with different purposes. These sections will all be explained further in this chapter. Below the different sections are listed:

- Configuration Constants
- *Containers*
  - MemoryChunk
  - HeapChunk
  - FreeList
- Managers
  - SharedMemory
  - MemoryManager

3.2 Configuration Constants

One of the most important parts of the memory management in Gecode is the constants. They are all declared in a configuration class and have impact on all the other parts of the memory handling. These constants will be explained below, since seeing them in their environment makes it easier to understand them.

3.3 Containers

The memory is handled in lists for different applications. This makes it important to have different node objects to keep them apart and be able to put the memory parts in the various lists. A typical list with memory parts is pictured in figure 6.

3.3.1 MemoryChunk and HeapChunk

A MemoryChunk is a node in a list, and has two variables; a pointer to the next MemoryChunk and the size of the chunk. The HeapChunk object extends MemoryChunk, so it has the same variables and also a pointer to the start of the actual memory area it contains. The HeapChunks are used a lot in the memory management, which will be explained later.

3.3.2 FreeList

The FreeList class is actually not a list as the name might indicate. It is rather representing the nodes in the lists of free memory for constraint variables to use. It has, as any node in a linked list, a pointer to the next element in the list.

The free list is used to allocate memory for the variable lists that are used in Gecode. These lists tend to change size after *propagations*, as explained in more detail in chapter 2.3.4, and can both grow and shrink, so it is important to be able to allocate and release memory when they change.

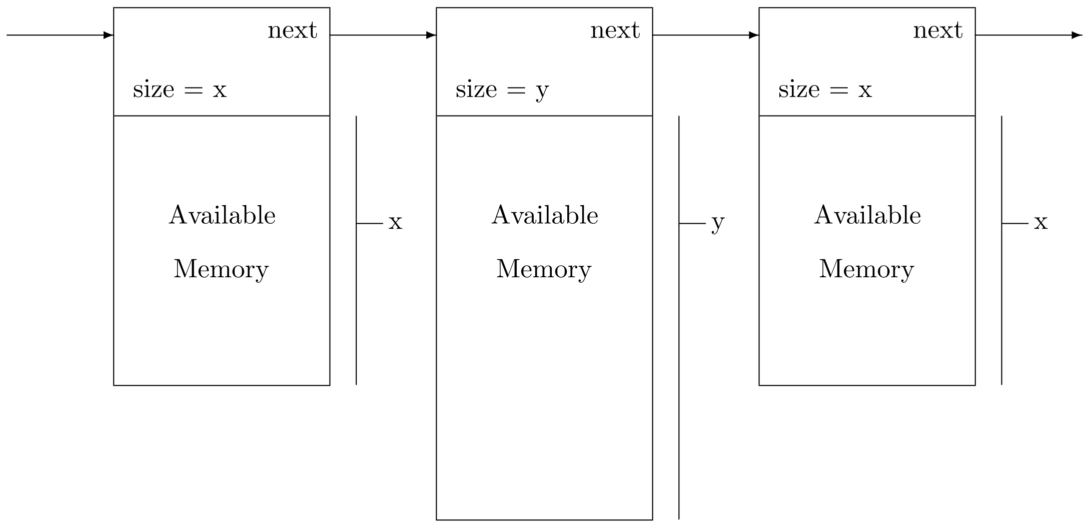

Figure 6: *List objects*

3.4 Managers

The *spaces* handle their own memory as much as possible, and do not share any variables or objects if not necessary. This is done to be able to release all the memory a space has used when the space can be removed. If the memory was shared this release would be harder to do[4]. It also makes parallelization simpler and more efficient. Some memory management is shared between spaces though. The reason for that is the memory *cache* in MacOS which did not cooperate with Gecode's memory usage, so the shared memory part was implemented to work around this problem. This shared part is called SharedMemory but since it will not be evaluated in this project it will not be discussed further here.

3.4.1 SharedMemory

The SharedMemory is a memory object that is shared between several spaces. It is implemented improve parallel execution.

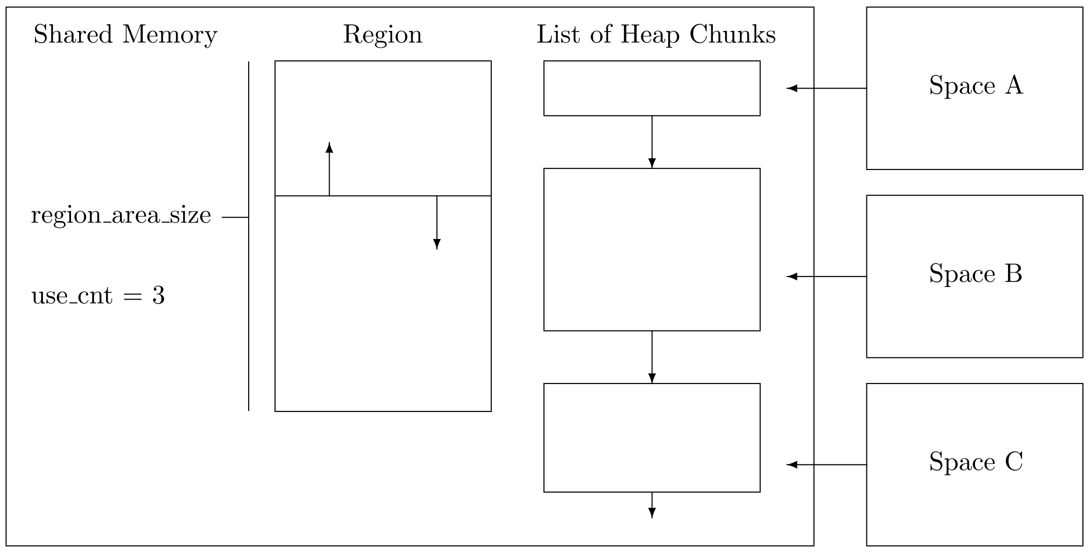

Figure 7: *SharedMemory*

The shared memory has both a region of memory that can be used and a list of HeapChunks. It is illustrated in figure 7. The HeapChunk list is used as Gecode's own HeapChunk *cache* which can keep a number of heap chunks for a while. If this cache is empty when a request for a HeapChunk comes from a *space*, new memory will be allocated from the operating system. Otherwise a HeapChunk from the cache is returned to meet the request. The cache list can only contain n_hc_cache number of elements. This constant, n_hc_cache, is declared in the configuration class mentioned above.

The Region of memory is handled like a stack with the total size region_area_size. The memory allocated here will only be released when that part of the Region and all Region parts requested later have been deleted. If a memory request for the Region cannot be met heap memory will be allocated instead. This heap memory is returned to the operating system as soon as it is deleted, i.e. it does not have to wait for the memory parts requested later to also be deleted.

Since this is a shared memory it must be possible to add and remove users, in this case, spaces. A copy and a release function which helps to keep track of how many instances there is to the class makes this possible. The copy function increases the count of users, release function decreases it. When the number is decreased to zero the destructor is called and the SharedMemory is deleted.

3.4.2 MemoryManager

Every space has its own MemoryManager. This memory manager has a pointer to a SharedMemory where it sends requests for memory when it needs it from the operating system.

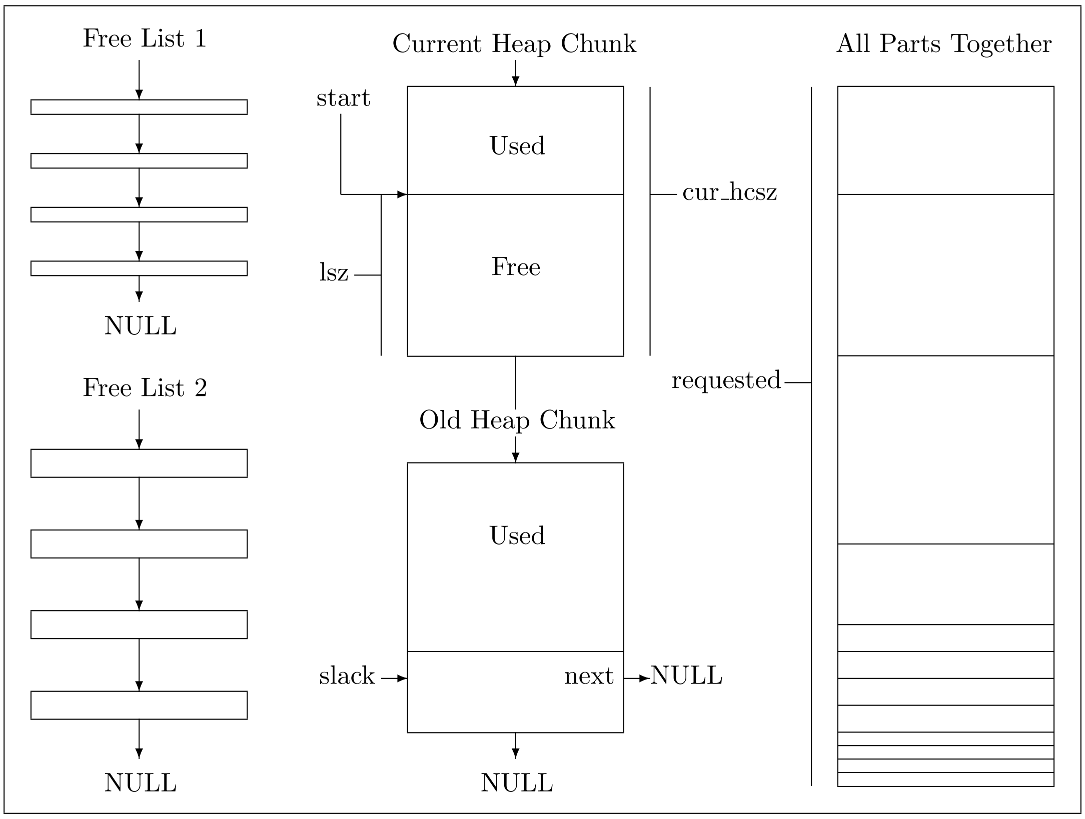

Figure 8: *MemoryManager*

The MemoryManager contains and uses quite some variables and objects. It has a HeapChunk that it can currently take memory from, some free lists for variables, a slack list that contains old HeapChunk parts that are too small to be used in the HeapChunk but big enough to put in the free lists if needed, the used HeapChunks and an integer to declare how much memory the manager currently has control over. All this is illustrated in figure 8 and will be explained further below.

Active memory manager

The active memory manager keeps track of how much memory that is currently under the *space*'s management, i.e. the total amount of heap memory requested and allocated. This is not necessarily how much that is used for the moment, but how much memory the space has control over, including free list, unused heap chunk parts and slack. The variable used to this is named requested and is visible to the right in figure 8.

When more memory is needed for the space the memory manager checks if there is enough memory left in the currently allocated heap chunk. If it is not a check is made to see if the unused part of the heap chunk is possible to put in any of the free lists or the slack list. Then the heap chunk allocation size is doubled if needed and finally an allocation request is sent to the SharedMemory. The increase check is showed in figure 9. If cur_hcsz, which declares the current heap chunk allocation size, times hcsz_inc_ratio is smaller than the requested variable the current heap chunk allocation size, cur_hcsz, will to be doubled. The new HeapChunk that is returned on the request is put first in the HeapChunk list, before the old one.

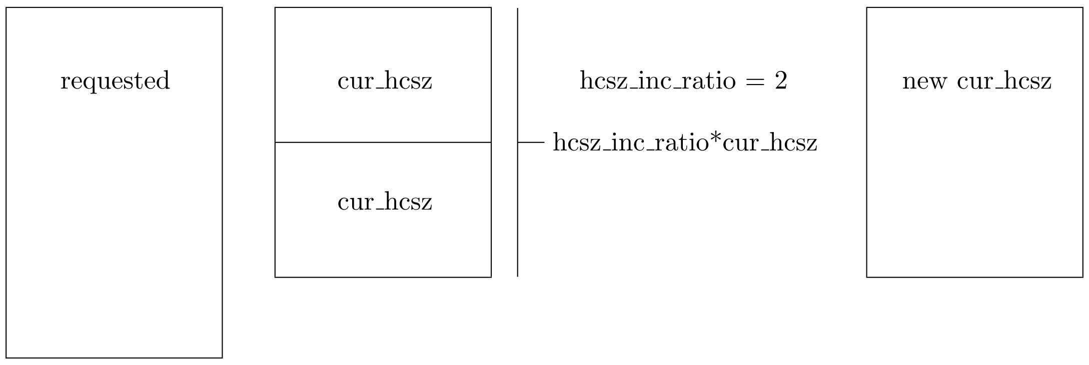

Figure 9: *Illustration of when the heap chunk request size doubles if hcsz_inc_ratio is 2*

The current heap chunk request size, cur_hcsz, can not only grow, but also shrink. This is important since the nodes in the search tree tend to get smaller the deeper in the tree they are located. Therefore a decrease check is made every time a space is cloned. This check works as the increase check but backwards, and uses the constant hcsz_dec_ratio instead of hcsz_inc_ratio. If hcsz_dec_ratio times cur_hcsz is bigger than the total amount of requested the cur_hcsz is halved. The mechanism illustrated in figure 10 below.

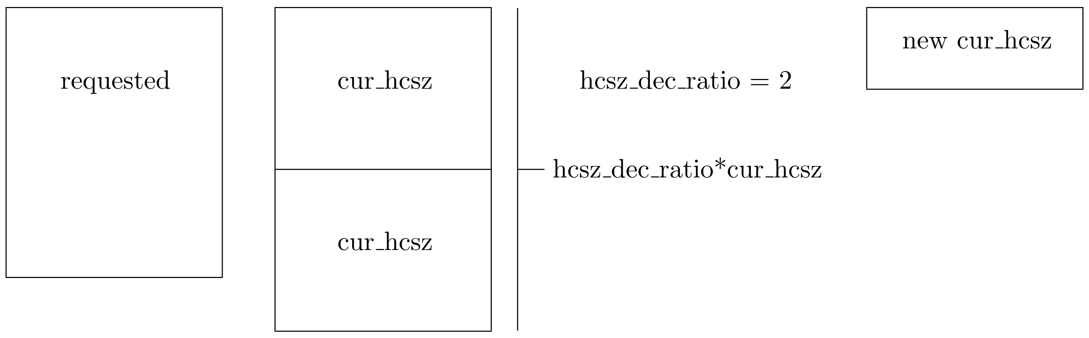

Figure 10: *Illustration of when the heap chunk request size halves if hcsz_dec_ratio is 2*

When the space is not needed anymore all its allocated heap chunks and other memory can be released.

Slack handling - to decrease amount of internal fragmentation

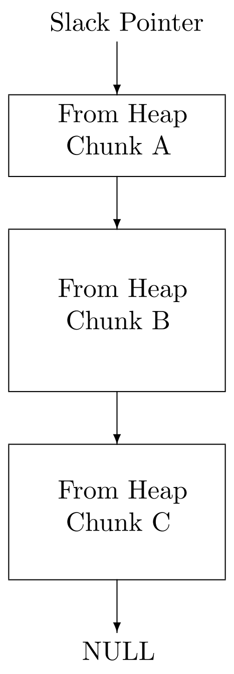

Figure 11: *Slack List example*

If the free part of the current HeapChunk has become too small to be of any use a check, to see if the free part is reusable for another purpose, is done. In a case when the chunk is smaller than the minimum size for elements in the free lists it is just dropped. If the size on the other hand is bigger than the maximum size it is put in the slack list, visualized in figure 11. Otherwise, if it is possible to put the whole chunk in a free list as it is, it is put in the list it fits in.

Why the slack is not put in the free lists right away if the size is not right is partly because it is important to put it in the right list, i.e. the one needing memory next, and partly because it is not always necessary to perform the job since this memory is sometimes not needed.

Free list handling

The memory manager has an array of free lists, where the size of the nodes is increased with one unit size for every index.

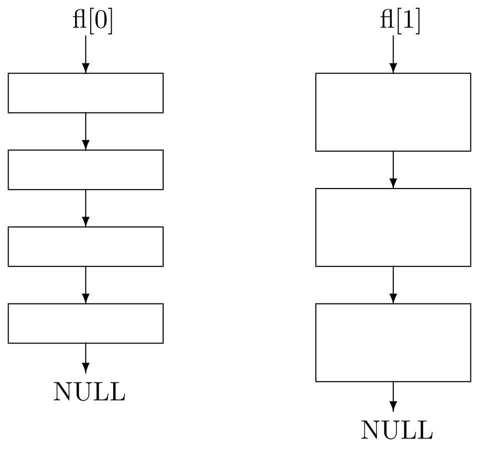

Figure 12: *Free Lists*

Currently this array does only contain pointers to two lists, with different node sizes. The free lists are illustrated in figure 12.

As explained earlier, constraint programming memory requirements are special. These requirements make it inefficient to use any already implemented memory management. One of the big reasons that make Gecode better than their competitors is that they have chosen to implement their own memory management. In this memory management they are cheating some to make the program run faster. The time that would be wasted taking care of these small cheatings is more important to avoid wasting than taking care of the small amount of memory wasted not doing so.

4 Evaluation Procedure

To be able to optimize something it is essential to evaluate it properly. The evaluation procedure of this thesis is explained below.

4.1 Measurements

The first thing to consider for an evaluation is what to measure. Since both the memory usage and runtime are supposed to be as good as possible the peak memory and runtime are two things to look into. It is also important to think through what can have impact on the memory usage and runtime. Therefore some traces will also be done in this evaluation. The traces will cover all allocations made when running the test programs and how the slack is used.

4.1.1 Peak Memory

The peak memory measures the maximum amount of memory that the program requires. It is not interesting how much memory is used in total, since it does not matter if one memory chunk is used and then returned before another chunk is requested.

4.1.2 Runtime

The runtime is one of the most important measures that have to be done. In computer programs the runtime often depends heavily on the memory management since one memory allocation call takes many times longer than to run a usual program instruction.

4.1.3 Number of Allocations

As mentioned above it takes quite some time to run a memory allocation request, so the runtime is dependent on how many allocations will be done during the run. In turn, the number of allocations depends of the allocation size. If the size is too small many allocations will be needed, and if the allocation size is too big unnecessarily much memory will be required to run the program. Therefore it is not only central to measure the total number of allocations, but also how big each of them are to see how this affect the peak memory.

4.1.4 Slack List Usage

The usage of the slack can be a memory hole if too much of it is put there, even if it will not be used. Some slack tracing and tests will be made to see if the slack lists get too big and if it is worth taking care of.

4.2 Test Programs

All tests will be run for seven different programs to get a good overview and approximation of how different kinds of programs will react on the different changes.

4.2.1 Memory Requirements

As for all kinds of computer programs, constraint programming programs requires different amount of memory. It is therefore essential to test programs with various memory requirements to see how they react on the memory management changes. In this project the test programs have been divided into three groups depending on their memory requirements; programs that use little memory, programs with medium memory usage and programs which use a lot of memory. Since the optimization for the programs with larger memory usage is most interesting four of the seven test programs use a lot of memory, the category for little memory has two test programs on the middle category only has one.

A list of the test programs follows below, but since it is not important to know what they do to understand this thesis they will not be explained further here. The critical thing to remember is in which memory category they are. All programs are taken from the example programs that come with Gecode when downloading it.

- Little Memory
  - Golomb Ruler
  - Bin-packing
- Medium Memory
  - Crossword
- A Lot of Memory
  - Bibd
  - Knights
  - Perfect Square
  - Queens

4.3 Isolation

When investigating what impact every constant has on the memory management it is important to start analysing every constant for itself, without impact from the others. Therefore the other constants have to be set to values that do not interfere with the variable currently investigated. Hence not only the currently tested variable value is considered during a test run, but also the values of the surrounding constants.

4.4 Presentation

4.4.1 Graphs

Most of the results will be presented in graphs where the colour of the graph will declare to which memory group it belongs. Here is a list of the different colours:

- Little Memory - red
- Medium Memory - green
- A Lot of Memory - blue

The graphs are normalized around 0, where a positive value is worse than 0 and a negative is better.

4.4.2 Synthesis

Every graph has explaining comments and in chapter 9 and 10 concluding remarks, reflections and recommendations are written.

5 Core Variables - Isolated

Four core variables, hcsz_min, hcsz_max, hcsz_inc_ratio and hcsz_dec_ratio, are used for the memory management in Gecode. They all help deciding and adjusting the size of the heap chunk allocations.

5.1 Procedure

All values should be chosen wisely, both for the currently tested variable and the surrounding ones. The isolated tests will be explained here.

5.1.1 Heap Chunk Allocation Size

The two variables hcsz_min and hcsz_max are used to decide the range for the heap chunk allocation size. Here this allocation size will be fixed, to see how the programs react on different sizes.

If the hcsz_inc_ratio is set to a large number the hcsz_min will declare the only possible heap chunk size. The reason is that hcsz_min gives the start value for the chunk size and if the hcsz_inc_ratio is large the chunk size will never get bigger. Since hcsz_min gives the minimum allowed size it will not get smaller either. Therefore hcsz_inc_ratio will be set to the maximum int value for these tests. Different hcsz_min values between 1 KB and 128 KB are tested. The hcsz_max will be set to the same as hcsz_min and hcsz_dec ratio is set to 0 even though their values do not really matter here.

5.1.2 Increase Ratio

To see how this variable hcsz_min should be set to a very small value and hcsz_max to a number large enough to show the effect from the increase ratio constant. hcsz_min is here set to 1 KB and hcsz_max to 256 KB. hcsz_dec_ratio will be set to zero so the size never is halved. Values between 1 and 32 are to be tested.

5.1.3 Decrease Ratio

The hcsz_min will stay the same as above, and the hcsz_inc_ratio will be set to a very large number. One change in the code, to see how the decreasing has impact on the run, is to change the initialization of the cur_hcsz from hcsz_min to hcsz_max. Otherwise the decreasing would not be visible since the increasing effect is removed. The value on hcsz_max is set to 128 KB. hcsz_dec_ratio will take be tested for values between 1 and 32.

5.2 Results

The results are illustrated graphically in figure 13 to 18, where red represents the programs with small memory requirements, green the ones in the middle and blue stands for the programs that uses a lot of memory.

5.2.1 Heap chunk allocation size

Runtime:

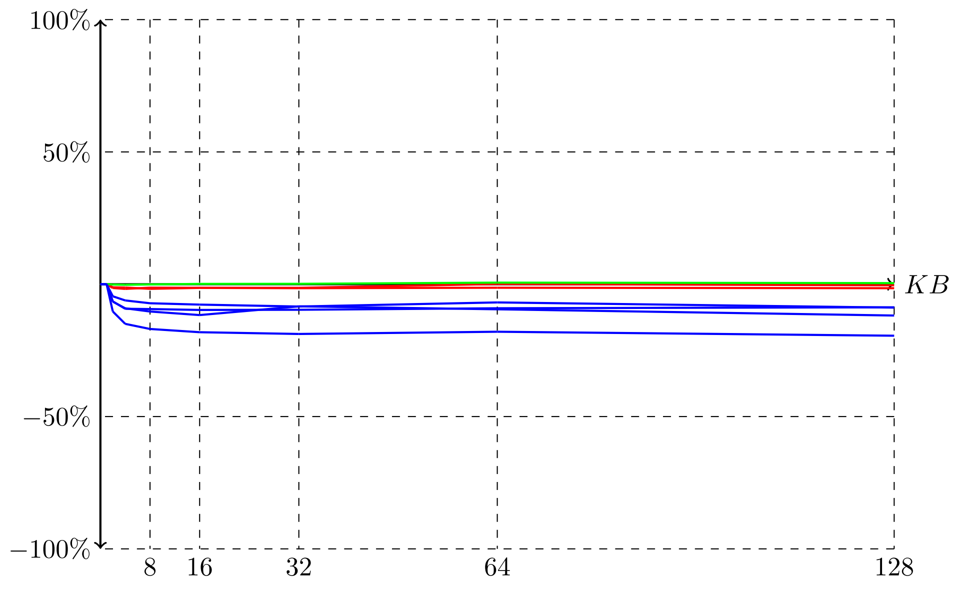

Figure 13: *Runtime for different heap chunk allocation sizes*

Peak Memory:

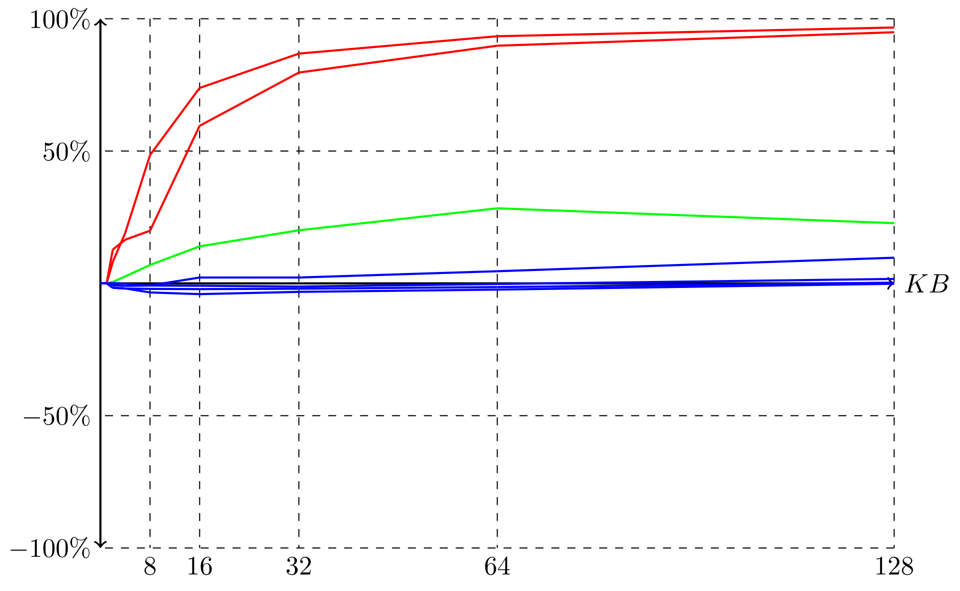

Figure 14: *Peak memory for different heap chunk allocation sizes*

Little Memory

For the programs requiring little memory figure 14 shows that the memory chunks cannot get too big, since they would waste a lot of memory if the allocated chunks are bigger than the needed. The number of allocations does not increase after the size four, which in this case means that basically one allocation is needed per node, and also that this one allocation is a lot bigger than needed for the node. On the run-time results in figure 13 it is visible that the number of allocations is more important than the size of them.

Medium memory

The result from the test runs on the medium memory usage program shows a similar result, but bigger parts of the memory chunks are used. This means that not as much memory is wasted but that it is still essential not to have too big memory chunks. The number of allocations decreases very fast in the beginning, but the runtime is not as affected of this as expected.

A lot of memory

When a program requires a lot of memory the memory allocations should be big. Figure 13 shows that time can be saved since the number of allocations decreases a lot when the heap chunk size is increased. The peak memory graph in figure 14 also shows that all the memory in the allocations is used which also has impact on the runtime results.

Together

The results are very different depending on the amount of memory required for the programs. Hence the heap chunk allocation size has to be dynamic.

5.2.2 Increase Ratio

Runtime:

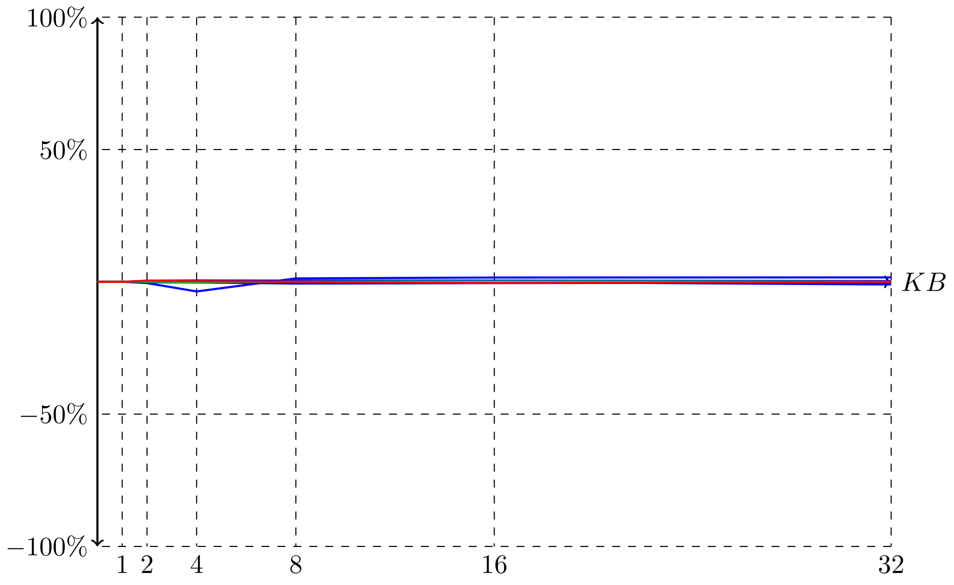

Figure 15: *Runtime for different increase ratio values*

Peak Memory:

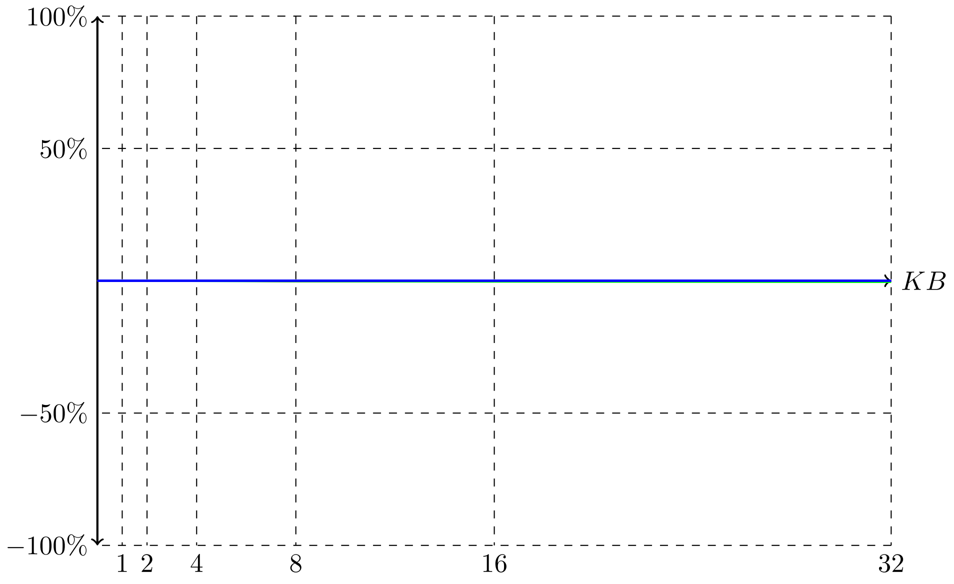

Figure 16: *Peak memory for different increase ratio values*

Little Memory
Neither the runtime, in figure 15, nor the peak memory, in figure 16, shows any significant changes for the programs with small memory requirements. When looking at the trace results it gets clear that the increase goes so fast that the maximum allocation size is reached before any large effect from the small initializing value is visible.

Medium memory

For the program requiring a medium amount of memory the results are similar to those seen above. The increase is very fast and therefore these tests do not show any clear changes.

A lot of memory

Some of the programs using a lot of memory reacted a little bit different on the changes, as shown in figure 15. When looking into it the reason for these reactions were not that the increase affected the memory management, but that the *propagations* were affected. When ignoring the propagation effect the same fashion as above can be seen.

Together

In all tests the allocation size reach the maximum allowed allocation size fast. Hence programs behaviours do not differ so much depending on the increase ratio value. Without any possibility to decrease the allocation size the increase is therefore redundant, since it is basically the same thing as setting the heap chunk size to max from start.

5.2.3 Decrease Ratio

The first tracing showed that the allocation size never got smaller than half of the starting size, which in these tests is the maximum allowed size. When analysing this reaction a bug was found. This bug was fixed and the tests were rerun. Here the results after this bug fix are showed.

Runtime:

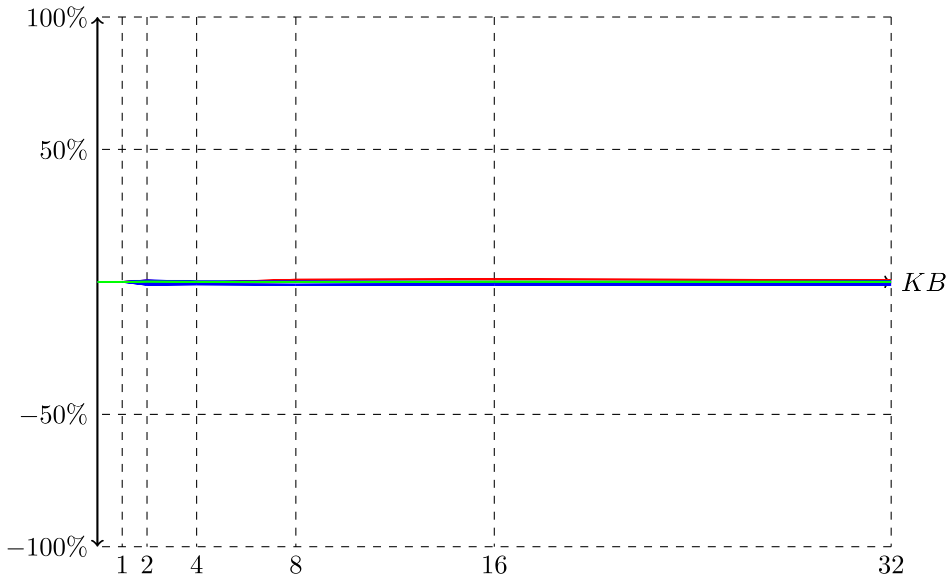

Figure 17: *Runtime for different decrease ratio values*

Peak Memory:

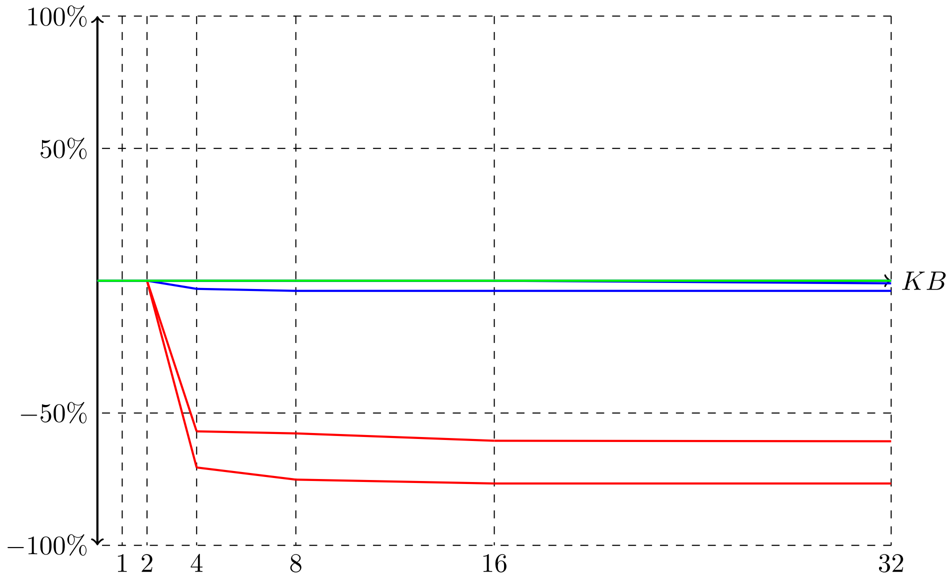

Figure 18: *Peak memory for different decrease ratio values*

Little Memory

In these tests and traces it is evident that the decrease of the allocation size is harder to do than the increase. The big starting value affects the continued allocations to the extent that it overrides some of the effect from the decrease ratio.

In figure 17 it is visible that the decrease is very useful after the bug fix. Memory can be saved if it is possible to adapt to smaller memory requirements by decreasing the allocation size.

The number of allocations stays the same and so does the runtime, which shows that the size of the allocations does not have a big impact on the runtime.

Medium memory
The runtime, showed in figure 18, is not changing so much depending on the decrease ratio. This gets interesting when also looking at the peak memory, in figure 17, and number of allocations. The program does not change the peak memory usage but the number of allocations is increased. The reason that the number of allocations is increased is that the allocation size stabilizes on a smaller value faster for a small decrease ratio. This is interesting since it shows that the runtime is not as dependent on the number of allocations as expected.

A lot of memory
For the programs that use a lot of memory, either the runtime or peak memory changes for the different decrease ratio values. This is because no significant decrease is needed for these programs. The number of allocations changes for some of the programs which shows that the runtime is not affected very much of this.

Together
The runtime is not very dependent on either the number of allocations or the size of them, which is interesting for the further tests. The results also show that it is easier to increase the heap chunk allocation size than to decrease it, but that the decrease is still important.

5.3 Result Synthesis

After these tests two things are in need of further testing, to find efficient ways to implement them. These tests are explained in later chapters.

First; the programs react differently on different heap chunk allocation sizes. If the program requires little memory a small heap chunk size is desirable, but if it requires a lot of memory it has to be big. From this the conclusion that a dynamic heap chunk size is needed can be drawn.

Second; the decrease and increase ratio are too dependent of each other to work alone. They also show that the increase goes faster than the decrease so probably a small starting value for the allocation size is desirable.

6 Merging the Core Variables

Since the core variables are dependent on each other, it is of course also important to test them together to see how they react on the different values.

6.1 Procedure

6.1.1 Merging Decrease and Increase Ratio

Some of the information below is simplified and focused on explaining the basics rather than introducing all details. It is shaped to make the evaluation understandable. The decrease and increase ratio are dependent of each other. To be able to test the effect of them it is therefore important to test them together without too much impact from the starting, minimum or maximum heap chunk size.

From the test results presented in chapter 5 the decrease ratio values for this test run could be decided to go from 1 to 8 and the increase to go from 1 to 16.

When these tests are run the minimum and maximum cannot have too big impact on the tests. Therefore minimum will be set to 0.5 KB and the maximum to 256 KB.

6.1.2 Total Range

When a good decrease ratio and increase ratio is found the maximum and minimum variables can be tested more properly. The values tested are listed below. Different combinations of the values will be run. The minimum allowed heap chunk allocation size is tested for values between 0.5 KB and 8 KB and the maximum for 8 KB to 256 KB.

6.2 Results

The graphical illustrations below, in figure 19 to 24, follows the same colour encoding as in the previous chapter. Red represents the programs with small memory usage, green in between and blue programs with large memory requirements.

6.2.1 Decrease and Increase Ratio

Runtime:

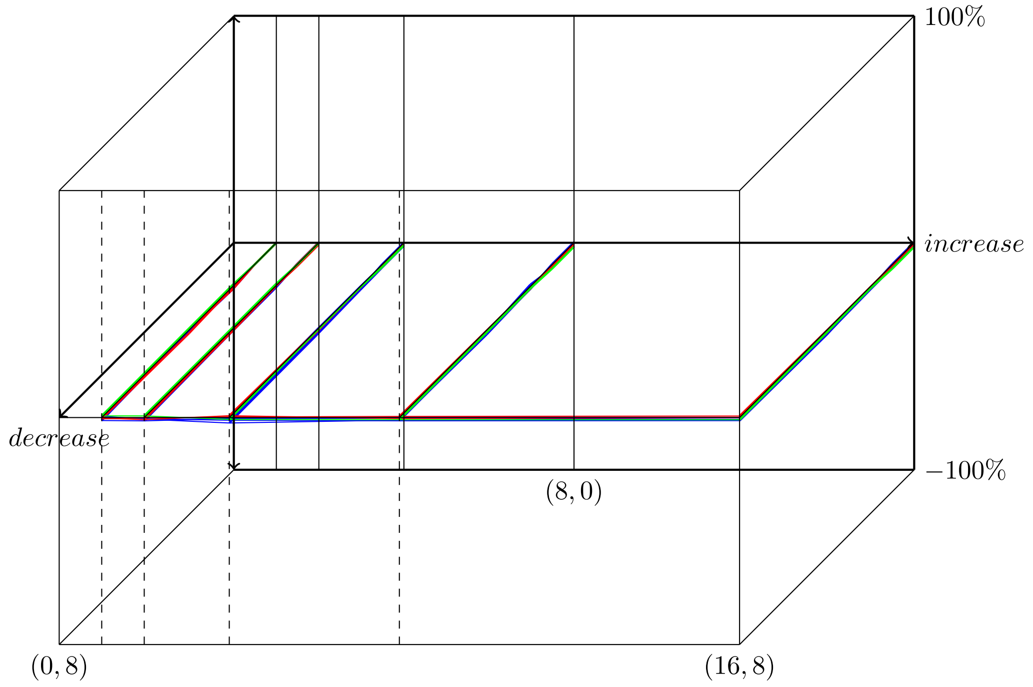

Figure 19: *Runtime for different decrease and increase ratio values*

Peak Memory:

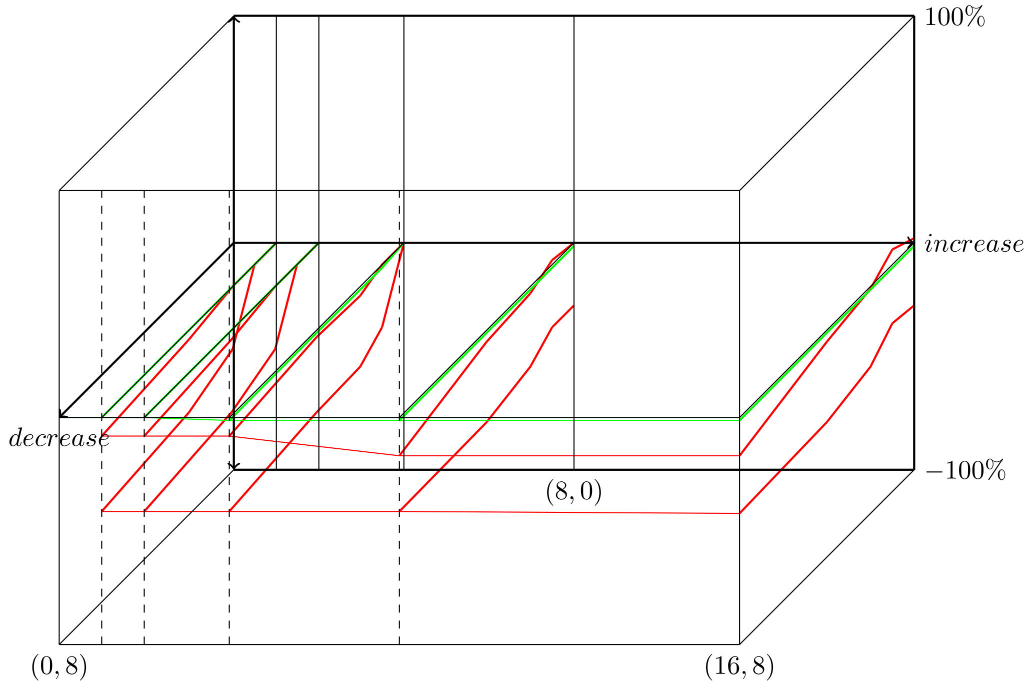

Figure 20: *Peak memory for different decrease and increase ratio values - - Small and medium memory usage*

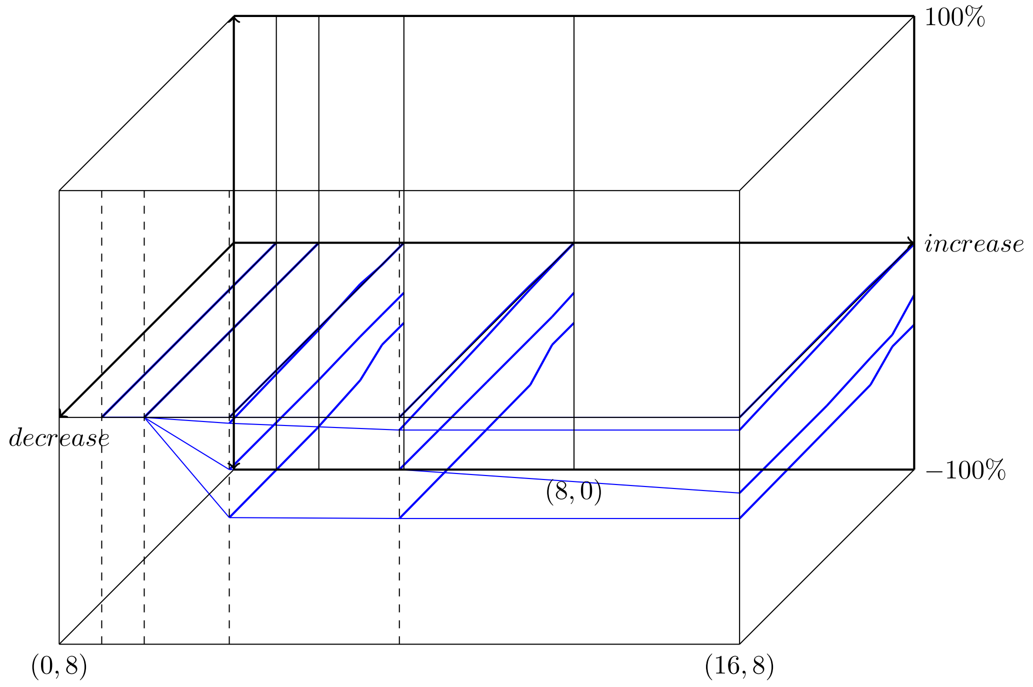

Figure 21: *Peak memory for different decrease and increase ratio values - Large memory usage*

Little Memory

When the value of the decrease ratio is larger, which means that decreasing is made earlier, less memory is used than if the value is small. This is showed in figure 20. The increase ratio does not have so much impact on the memory usage though. Some tracing shows that the reason for this is that the increase is not needed a lot for these programs.

Medium memory
For the program using a middle amount of memory the test result changes are small. The reason for this is that the search tree node sizes for this program do not change much during the search, and therefore either the decrease or increase is so important when an efficient allocation size is found.

A lot of memory
The tests from the programs requiring a lot of memory show similar results but in different orders of magnitude. For some of them a lot of memory can be saved, as visible in figure 21, if both the decrease and increase ratio is a bit larger. That means that the decreasing goes faster and the increasing slower than for small values.

Together
The runtime, in figure 19, does not show any significant changes at all, so the focus for these test results is on the peak memory usage. Memory is saved if the decrease goes faster and the increase goes slower, so large values on both the decrease and increase ratio is to prefer.

6.2.2 Total Range

Runtime:

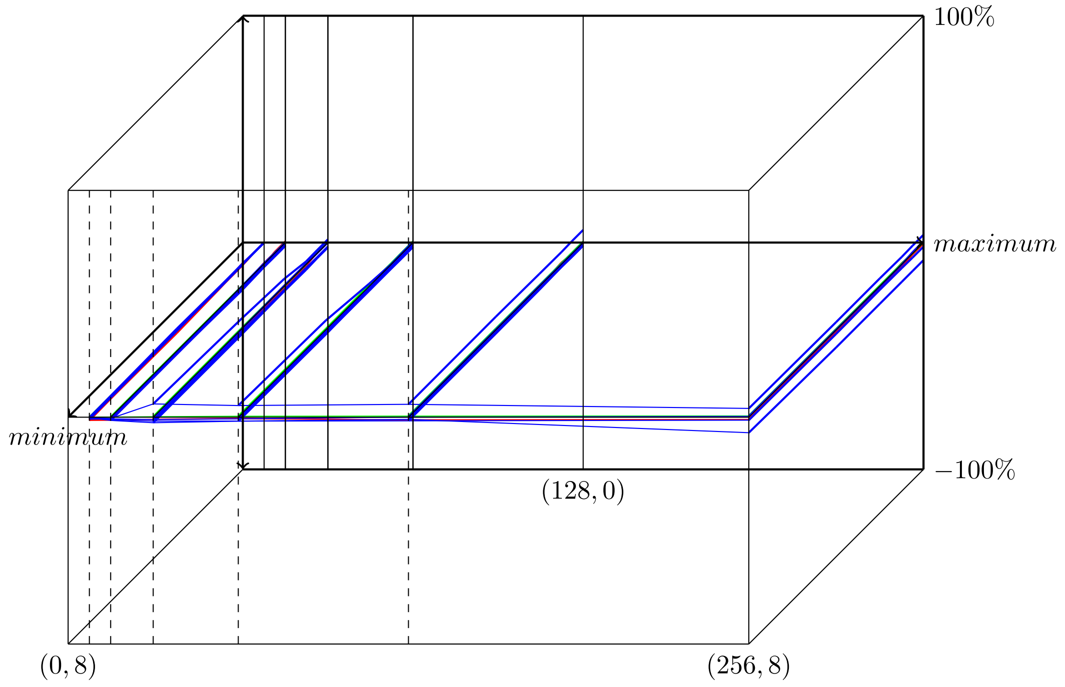

Figure 22: *Runtime for different minimum and maximum allocation size values*

Peak Memory:

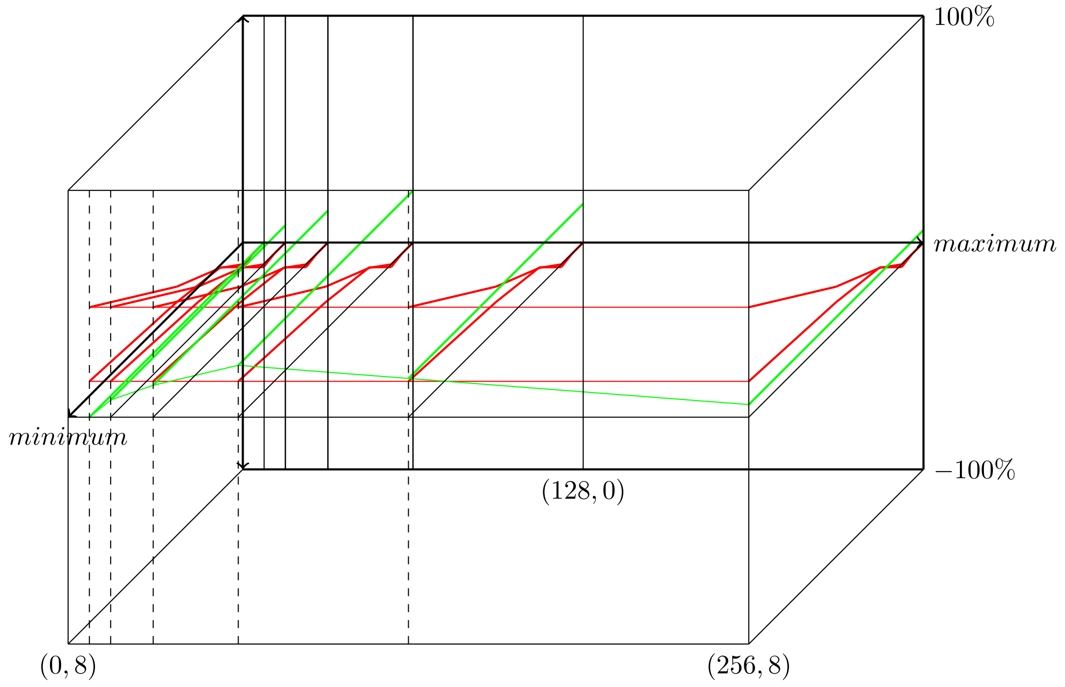

Figure 23: *Peak memory for different minimum and maximum allocation size values - Small and medium memory usage*

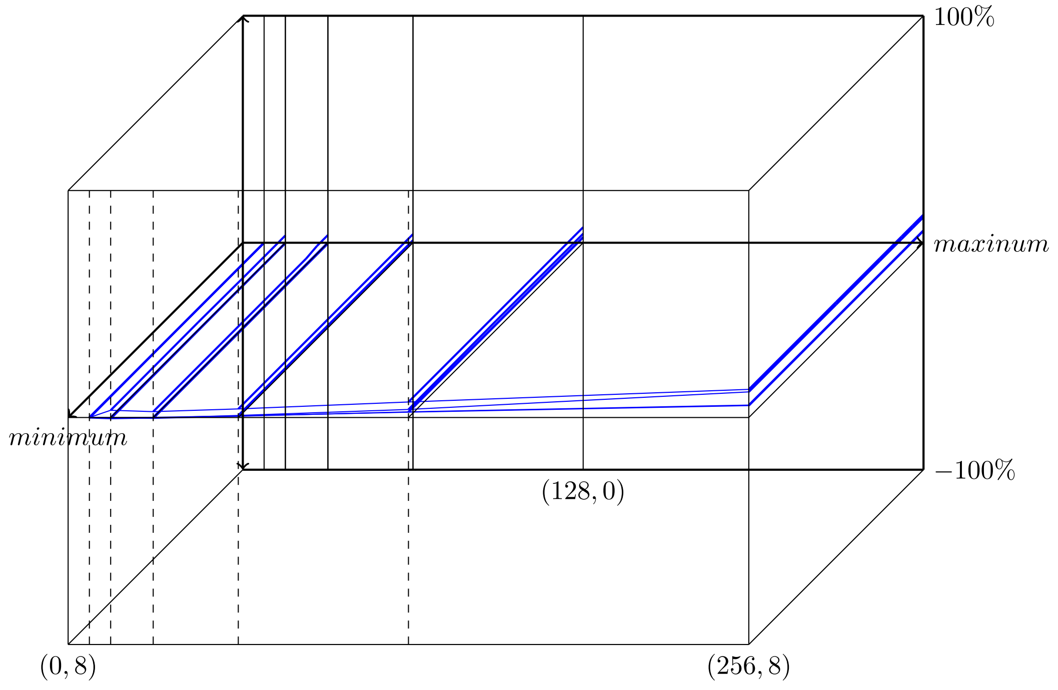

Figure 24: *Peak memory for different decrease and increase ratio values - Large memory usage*

Little Memory
The test programs requiring little memory gets a better memory usage result, shown in figure 23, when the minimum size is small. This is because the programs do not need the amount of memory that a larger minimum size forces them to allocate. Extra memory will then be requested and wasted. The maximum size does not matter though, since it is never reached.

Medium memory
The reaction on the program using a medium amount of memory is a bit confusing. It uses more memory for the middle values of maximum allocation size, without showing any behaviour changes. This is visible in figure 23. The tracing do not show any clear reason for these results either.

A lot of memory
The maximum heap chunk size has a clear impact on these programs, since it is reached early for them all. An interesting thing here is that the runtime, showed in figure 22, is not affected in a bad way even if the total number of allocations is a bit larger for a smaller maximum than a for larger one. This means that the maximum limit do not need to be large to fit these programs, but should rather be a bit smaller to save memory, which gets visible when looking at the peak memory results in figure 24.

Together
Running these tests shows that the minimum size should be small to fit the programs requiring little memory, and the maximum size does not have to be very large since the runtime is not badly affected if it is a little bit smaller. The maximum should not be too small though, but has to give the allocation size some opportunity to grow when needed.

6.3 Result Synthesis

To make the increase and decrease functionalities to work well together both of them should be quite large.

The minimum size should be small to meet the requirements for the programs with a small memory usage and the maximum size should not be too big since memory can be saved with a smaller value without affecting the runtime in a bad way.

From the test results in chapter 5 it shows that a fast increase to adapt to the right size for the specific program was to prefer. One possibility to make this adaption even faster is to create a new variable to use for initiating the starting allocation size. After the traces of the tests in this chapter it got clear that this would not make any significant difference. The increase is already so fast that no time can be saved by adding this.

7 Interpretation and Reflection

The results from the tests that have been run during this project will be discussed in this chapter.

7.1 Core Variables

From the core variable test results the following recommendations can be made.

7.1.1 Runtime vs Memory usage

Already in the first test runs it was shown that the memory management did not have as big impact on the runtime as expected. Usually the memory allocations take so much time that the runtime of a program is very dependent on its memory management, but that is not the case in Gecode. Therefore it is more important to optimize Gecode's memory management in perspective of the peak memory requirements rather than on runtime.

7.1.2 Dynamic Allocation Size

The first results also show that the allocation size has to be dynamic. The different programs react too differently on the different heap chunk sizes to be able to find a size that would be good for all.

7.1.3 Decrease Bug Fix

The decrease functionality did not work as planned. Right after the decrease was made an increase to the original size followed. Due to this bug the planned decreasing only had the effect of slowing the increase down, so no actual decrease was made. By using a Boolean (true/false) variable already sent to the function causing this increase this following increase could be stopped without affecting the rest of the increases. When this fix was made quite some memory could be saved.

7.1.4 Small Start - Rapid Adjustment

The decreasing and increasing traces indicate that the decrease is slower than the increase. The reason for that is the fact that the decrease only is done when creating a new node and the increase can be done many times for each node. It is also harder to recover from memory lost for too big allocations than it is to recover from some extra allocation calls caused by a small initializing size. Therefore it is important that the start value of the heap allocation size is not too big, but rather small but able to increase very fast. If a program requires a lot of memory the size will be adjusted fast enough to overcome the time lost in the start, but it does not work the other way around.

7.2 Merging the Core Variables

The test results from chapter 6 give an insight in how the core variables react on each other's changes. The relationship between the decrease and increase ratio, and what size of the total range that works best were investigated.

7.2.1 Decrease and Increase Ratio

A decrease of the allocation size can only be made when a new node in the search tree is created. Increasing on the other hand can be done for every heap chunk allocation request, and hence more than once per node if needed. Therefore the decrease should be made earlier to have the right impact and the increase can be made a bit slower since it is easier to do.

The decrease gets faster the larger the decrease ratio value gets, and the increase gets slower the larger the increase ratio gets. Therefore both the decrease and increase ratio should be large.

7.2.2 Total Range

When the programs with small memory usage requirement were run it got clear that the minimum allocation size has to be small. If it gets too big these programs would allocate more memory than they need.

One of the more interesting facts showed from the results of changing the maximum allocation size was that the runtime did not change significantly even if it was smaller and some extra allocations were needed. Therefore memory can be saved with a smaller maximum size without affecting the runtime in a bad way.

7.2.3 New Initializing Size

The tracing in chapter 6 show that an earlier considered new initializing value is not needed, since the increase is so fast that it would not do any significant different. It would just add some complexity without giving any optimizing results.

7.3 Project Review

In this project some choices have been made for different reasons. These choices might have affected the test results and are important to consider.

7.3.1 Number of Test Programs

In the beginning of the project eight test programs were used, but one of them turned out to react by changing its behaviour for the different tests. Hence the test results showed from this program did not give a fair view of the memory management. Therefore it was removed from the test program set.

With this program removed from the test program set only seven programs were used making this evaluation. The small number of programs gives a clear and non-complex results to analyse, but cannot guarantee that all programs will react in the same way.

7.3.2 Operating Sytems

All tests in the project have been run on a Linux (Ubuntu) computer. It is important to note that other operating systems might react differently on the changes and hence give other results.

8 Recommendations

Some simple changes in the source code of Gecode can be made to optimize its memory management. First the core variables should be set to other values and second the decrease bug should be fixed. After these changes the memory management is improved in respect of both peak memory usage and runtime.

8.1 Core variables

The evaluation showed that by changing the values on the core variables the memory management can be improved. After the evaluation the following values can be recommended for the core variables:

```
hcsz_min = 1024
hcsz_max = 32*1024
hcsz_dec_ratio = 8
hcsz_inc_ratio = 8
```

8.2 Decrease Bug Fix

As explained in section 5.2.3 and 7.1.3, the decrease functionality had a bug. This can simply be fixed by adding a constraint in the if statement deciding if to increase or not. If the Boolean variable first, which always is true after a decrease, is used in this if statement the increase can be avoided after a decrease. So instead of:

```
if (((requested > MemoryConfig::hcsz_inc_ratio*cur_hcsz) ||
          (sz > cur_hcsz)) &&
         (cur_hcsz < MemoryConfig::hcsz_max))
```

this can be used:

```
if (((requested > MemoryConfig::hcsz_inc_ratio*cur_hcsz) ||
          (sz > cur_hcsz)) &&
         (cur_hcsz < MemoryConfig::hcsz_max) && !first)
```

8.3 Run tests on other OS

In this project the tests were only run on Linux (Ubuntu). Since other operating systems might react differently on the recommended changes tests should also be run on Mac and Windows when the changes are implemented, to make sure that the memory management is not affected in a bad way for them.

8.4 Improvements

When these recommended changes have been implemented the memory management has been improved for all test programs used in the project, as visible in figure 25. The red line shows the runtime and peak memory usage before the changes and the blue bars show the results after the changes.

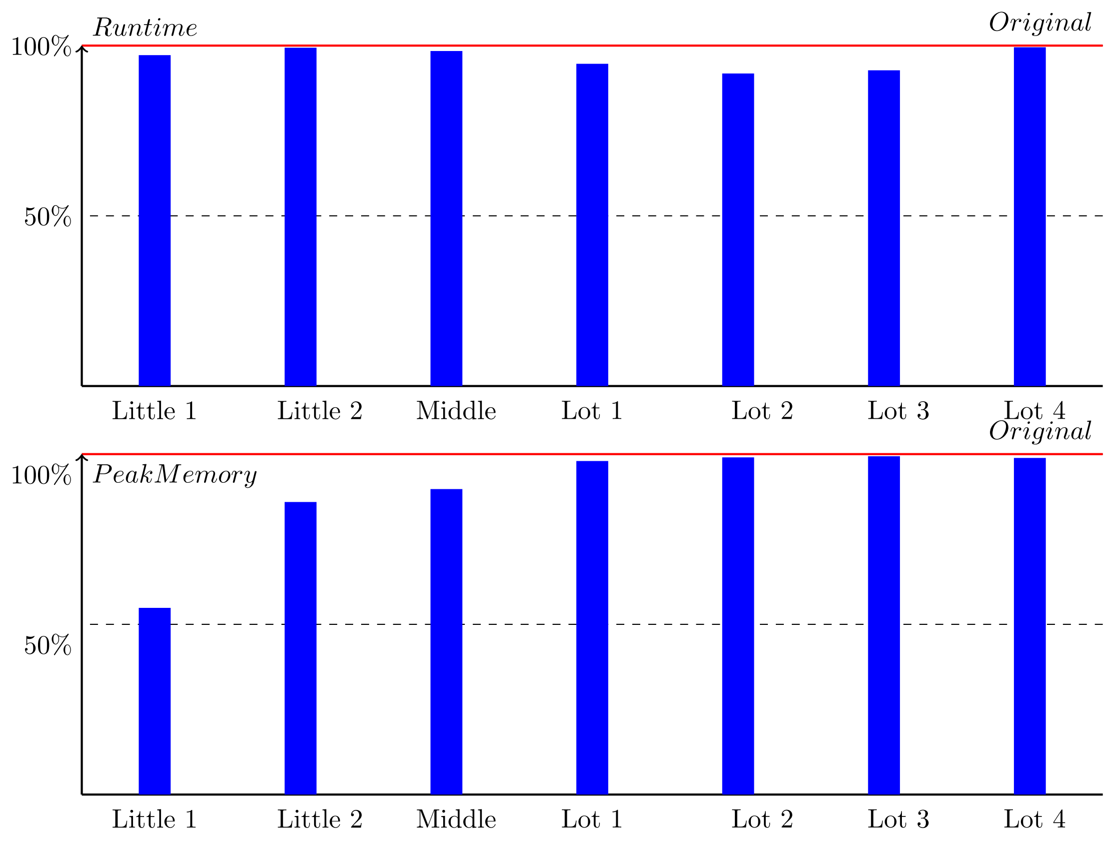

Figure 25: *Improvements - The red lines show the results before the optimizations and the blue after they were implemented*

Figure 25 shows that even in the worst test program case some improvements have been made for all the test programs.

As figure 25 also shows, the programs requiring a small amount of memory have improved their peak memory usage more than the once requiring a lot. This might seem like a bad result, but actually shows that the adaption for every specific program has gotten better. The programs requiring a lot of memory has also been improved both when looking at the runtime and peak memory usage. Since they have a large memory usage quite some memory has been saved for them as well, even if it is not as visible when measuring the difference from the start.

References

[1] Schulte, C. Tack, G. Lagerkvist, M. (2012a). *Modeling and Programming with Gecode.* Corresponds to Gecode 3.7.3.

[2] Tanenbaum, A.S. (2009). *Modern Operating Systems (Third Edition).* United States of America.

[3] Brorsson, M. (1999). *Datorsystem Program- och maskinvara.* Lund: Studentlitteratur.

[4] Christian Schulte. (2012b). *Main developer Gecode.* Stockholm: KTH.

[5] *ID2204 Constraint Programming.* (2012). Stockholm: KTH Cource.

[6] *Gecode website.* (2005) [Last change; 20 Mar 2012]. www.gecode.org

[7] *MiniZinc website.* (2012) [Retrieved; 22 May 2012]. http://www.g12.csse.unimelb.edu.au/minizinc/

[8] *The C++ Resources Network.* (2011) [Retrieved; 22 May 2012]. http://www.cplusplus.com

[9] *BProlog website.* (1994) [Last change; 23 Apr 2012]. http://probp.com/

[10] *JaCoP website.* (2012) [Last change; 16 Apr 2012]. http://jacop.osolpro.com/

[11] *Email conversation with Quintiq.* (2012) [Last mail: 28 Mar 2012]