CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Information Retrieval and Dominating Sets

Master's thesis in Computer science and engineering

Mattias Ahlstedt

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Information Retrieval and Dominating Sets

Mattias Ahlstedt

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Information Retrieval and Dominating Sets
Mattias Ahlstedt

Supervisor: Peter Damaschke, Department of Computer Science and Engineering
Examiner: Devdatt Dubhashi, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Information Retrieval and Dominating Sets
Mattias Ahlstedt
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The problem studied in this paper is a special case of set membership testing. This problem is defined as storing a single element from a set of m elements, represented as a bit string of optimal length, and answering whether a queried element is the stored one without reading all the stored bits. Damaschke has shown that finding l-cube dominating sets in hypercubes is equivalent to finding schemes for membership testing. The specific schemes that he found only serve as proof-of-concept and the problem of optimal schemes is an open question. Our main contribution is numerical improvements on these schemes. The new schemes have primarily been found using the low-dimensional (grid) relaxation that Damaschke introduced, and a branch-and-bound search approach.

Keywords: cube domination, bit probe model, hypercube set, membership testing.

Acknowledgements

First and foremost I would like to thank my supervisor Peter Damaschke for his invaluable help throughout the entire project. I would also like to thank the reviewers from the writing seminars and my opponents for their feedback.

Mattias Ahlstedt, Gothenburg, June 2019

Contents

List of Figures

List of Tables

1
Introduction

Consider a data structure that supports two operations: storing up to n elements from a given set, represented as binary strings of optimal length, and answering whether a queried element is among the stored elements. This problem is called *set membership testing* and the data structure that solves this problem is commonly denoted as an (m, n, s, t)-scheme where m is the size of the set, n is the maximum number of stored elements, s is the number of bits used to represent an element from the set, and t is the number of bits that are read when answering queries. This is a data structure problem that has been studied extensively [1]–[4]. We will only consider the one-element version of this problem, i.e. $(m, 1, s, t)$-schemes. This is not a limitation per se, since the one-element version can easily be used to implement the general case. One can use multiple instances of the one-element version together with a construction that finds the stored elements in memory and uses the corresponding set membership tester. However, the general case is not necessarily constructed in this manner.

This thesis focuses on trying to improve schemes where $s > t$, i.e. answering queries without reading all the stored bits, and as such we consider the problem in the context of the *bit-probe model*. This is a model of computation where all operations, except memory access, are considered free and the model is thus particularly useful when studying theoretical lower bounds on data usage for data structures. This is since the bit-probe model clearly shows the minimum number of memory accesses necessary to solve a given problem where there is stored data on which we perform a query. There are various recent results on membership testers in the bit-probe model [5]–[8], of which the paper by Damaschke is the primary work on which this thesis builds.

Damaschke [8] presents a generalised version of domination in hypercubes and the equivalence of this problem to saving probe bits in set membership testers. Through experimentation and relaxations of the problem, he has found various specific constructions for saving up to three probe bits. The primary goal of this paper is to improve on the specific schemes from [8] through means of computer aided systematic search. Practically speaking, this means coming up with and implementing algorithmic approaches for finding schemes. Damaschke notes that usual domination in hypercubes is already considered a hard problem and even non-optimal improvements may therefore be of value.

The main reason to study the topic is to attain a better understanding for the combinatorial nature of the problem and the underlying theoretical concepts. The relation between hypercubes and bit strings, as well as domination in graphs, are both common and fundamental concepts in their respective areas. While it may

sound unlikely that saving a few bits is going to have any immediate relevance in practical applications, the increase in big data applications over the last years might result in settings where the saved bits allow for a noticeable increase in performance.

2
Theory

The method for finding $(m, 1, s, t)$-schemes is based on the relation between bit strings of a given length s and the s-dimensional hypercube. Damaschke presents in his paper [8] how a generalisation of domination, called l-cube domination, can be used to find these schemes. More precisely he proves that an $(m, 1, s, t)$-scheme exists if and only if there exists an l-cube dominating set with $l = s - t$ in the s-dimensional hypercube. While not conclusively proven, it is also suspected that there is an upper bound on l, and the current belief is that $l < 4$ [8].
In the following sections we will present the definition of hypercubes, l-cube domination, and related concepts. Furthermore, we will present some of the methods that are used to find cube dominating sets in hypercubes.

2.1 Hypercube graphs

There is more than one way to describe the structure of a hypercube graph, and the interesting one for our application is the relation between binary strings of a fixed length s and the s-dimensional hypercube. Each node of the graph maps to a unique element from the set $\{0, 1\}^s$, and two nodes are adjacent if they differ in exactly one bit. As an example, we present the 3-dimensional hypercube in figure 2.1.
Another name for the hypercube is the *Hamming cube*, due to the relation to the notions of *Hamming weight* and *Hamming distance*. The Hamming weight of a binary string, $hw(s)$, is equal to the number of ones in that string, and the Hamming distance between two nodes is the number of character substitutions that are needed to change one string into the other. This means that the distance between two nodes in the s-dimensional hypercube is equal to the Hamming distance between the binary strings represented by those nodes.

2.2 *l*-cube domination

l-cube domination is a generalised version of domination in graphs. Regular domination is defined as: given an arbitrary graph, a subset D of the nodes of the graph is a dominating set if every node of the graph is either in D or has a neighbour in D. In l-cube domination, instead of having a neighbour in D, each node that is not in D must form an l-dimensional hypercube together with $2^l - 1$ nodes from D.
In figure 2.2 we can see an example of a dominating set in the 3-dimensional hypercube. We note that this is also a 1-cube dominating set, since a pair of nodes

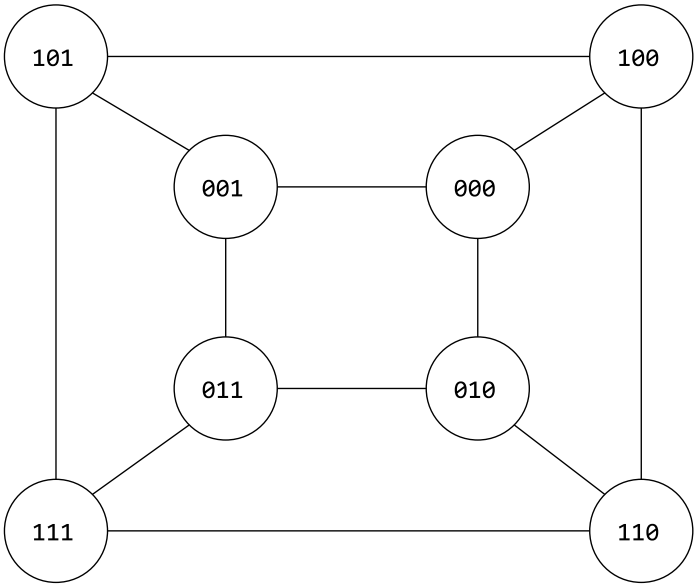

Figure 2.1: The 3-dimensional hypercube with corresponding bit strings.

connected by a single edge is a 1-dimensional hypercube. To illustrate how this extends to higher l we present an example of an 2-cube dominating set in the 3-dimensional hypercube in figure 2.3.

2.3 Finding schemes

Damaschke proves in [8] that the problem of finding an $(m, 1, s, t)$-scheme is equivalent to finding an l-cube dominating set in the s-dimensional hypercube. This means that finding an optimal scheme, i.e. maximising m for a given pair of s and t, is equivalent to finding the smallest possible l-cube dominating set in the s-dimensional hypercube. In addition to the m element strings and the cube dominating set, we also reserve a single bit string to represent the state where no element is stored, and we denote this string *the zero string*.

From the cube dominating set we get a set of *probes*. For our purposes it is not necessary to know how this is done and we therefore refer the interested reader to [8] for the details. The set of probes is of size m, one probe for each element string, which defines which bits need to be read. A probe is an element from the set $\{0, 1, *\}^s$, where the $*$ is the wildcard character. If the i:th character of a probe is a $*$, then the i:th character of the corresponding element string need not be read.

Since the aforementioned zero string does not represent an element, we do not need to find a probe for it. This means we can remove the node that represents the zero string and get a punctured hypercube, which is a hypercube after removal of a single node and all of its incident edges.

Since regular domination is identical to 1-cube domination, we can use existing results for dominating sets in hypercubes to find schemes. These results are summarised in [8] and presented in table 2.1. It may be possible to improve these results

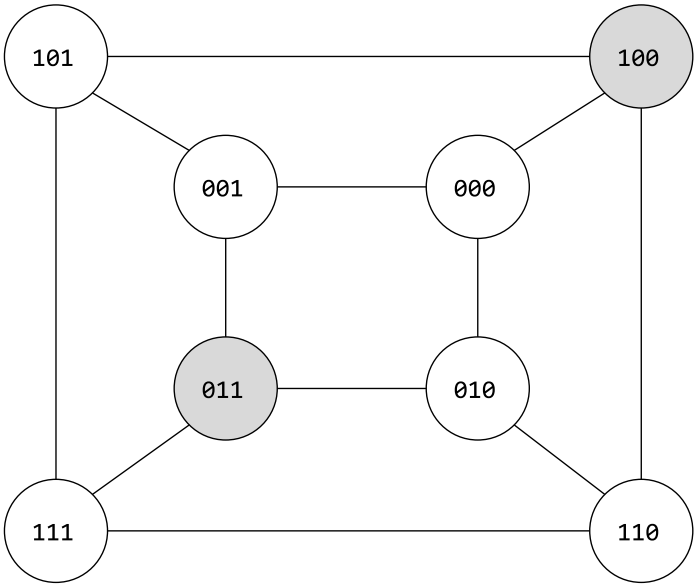

Figure 2.2: The 3-dimensional hypercube with an example of a 1-cube dominating set. The grey nodes belong to the dominating set.

by one, since we only need to find a dominating set in the punctured hypercube.

Dimensions	Domination number	Scheme
3	2	$(5, 1, 3, 2)$
4	4	$(11, 1, 4, 3)$
5	7	$(24, 1, 5, 4)$
6	12	$(51, 1, 6, 5)$
7	16	$(111, 1, 7, 6)$
8	32	$(223, 1, 8, 7)$
9	62	$(449, 1, 9, 8)$

Table 2.1: Known domination numbers for hypercubes and the corresponding schemes these would yield.

2.4 Grid relaxation

The schemes for saving two or three bits that were found in [8] were all constructed from a low-dimensional grid relaxation of the problem. The construction works by mapping the nodes of the hypercube onto a grid. This is done by first splitting the bit string of length s into g segments. The Hamming weight of the i:th segment corresponds to the index along the i:th dimension of the grid. Each point in the grid has a numeric value that is the amount of nodes mapped to that point.
To illustrate this further, we will use the bit string of length five as an example. If we map this to a 1-dimensional grid, each entry will simply be the number of nodes with Hamming weight (hw) equal to the index of that entry. This would give the

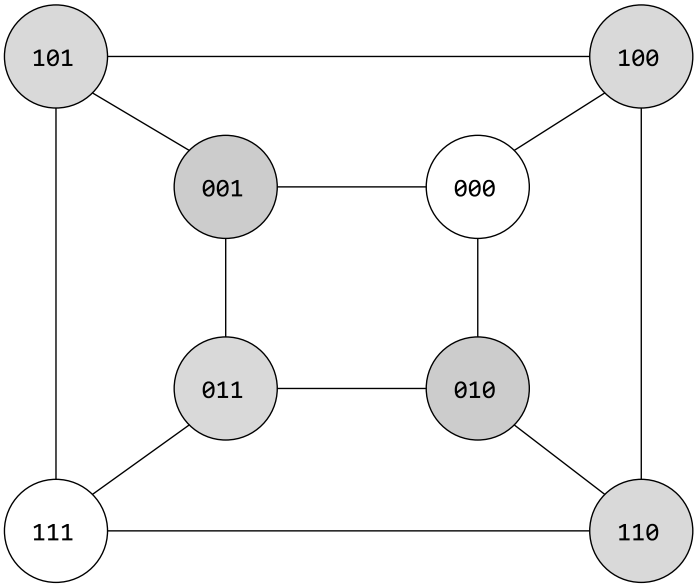

Figure 2.3: A 2-cube dominating set in the 3-dimensional hypercube. The grey nodes belong to the dominating set.

following grid, where all bit strings s where $hw(s) = 0$ map to the first entry, all bit strings s where $hw(s) = 1$ the second entry, etc.

1	5	10	10	5	1

If we instead consider a 2-dimensional grid and the following split of the bit string: $[b_0, b_1, b_2|b_3, b_4]$; we would get the following grid, where bits 0 through 2 map to the x-axis, and bits 3 and 4 map to the y-axis. This means that, for example, the rightmost 6 in the grid maps to all nodes where $hw(b_0, b_1, b_2) = 2$ and $hw(b_3, b_4) = 1$. We note that this construction means that all entries along the edges will be binomial coefficients, and the remaining entries will be products thereof.

1	3	3	1
2	6	6	2
1	3	3	1

In these grids we mark an entry with $*$ to denote that all nodes that are mapped to that entry are in the dominating set. For saving two bits, the nodes that map to an entry e are dominated if the entry matches one of the following two cases, in any orientation.

$*$	$*$	e

$*$	$*$
$*$	e

Similarly for saving three bits, we instead look for the following cases.

$*$	$*$	$*$	e

$*$	$*$	$*$
$*$	$*$	e

Returning to the previous two example grids, we present the following examples of 2-cube dominating sets in said grids.

*1	*5	10	10	*5	*1

*1	*3	3	1
*2	6	6	*2
1	3	*3	*1

2.5 The doubling lemma

This is a lemma from [8] which is used as a method for finding an l-cube dominating set in the $(d+1)$-dimensional hypercube, given an l-cube dominating set in the d-dimensional hypercube. In general, this does not give an optimal solution in the $(d+1)$-dimensional hypercube.

> "Let D be an l-cube dominating set in the full $(L+1)$-dimensional hypercube Q_{L+1} with $|D| = 2^{L+1} - m$, hence with exactly m vertices outside D. Let f be any positive integer. Then we have: (i) There exists an $(m-1, 1, L+1, L+1-l)$-scheme. If D has some redundant vertex, then there exists an $(m, 1, L+1, L+1-l)$-scheme. (ii) There exists a $(2^f m - 1, 1, L+1+f, L+1+f-l)$-scheme. If D has some redundant vertex, then there exists a $(2^f m, 1, L+1+f, L+1+f-l)$-scheme."

To explain the lemma in a less formal manner, consider the following: In order to arrive at the $(d+1)$-dimensional hypercube, one can take two copies of the d-dimensional hypercube, and add an edge between each node and its clone in the copy. Similarly one can copy an l-cube dominating set in the d-dimensional hypercube and arrive at an l-cube dominating set in the $(d+1)$-dimensional hypercube. Since both halves of the $(d+1)$-dimensional hypercube are l-cube dominated, the full hypercube is also l-cube dominated.

3
Method

The following sections describes the algorithmic approaches that have been used to search for cube dominating sets. The implementations can be found in appendix B.

3.1 Exhaustive search

The first approach taken was that of a simple exhaustive search. We start by generating the power set of the set of nodes, then we test each subset for whether it is an l-cube dominating set or not, and finally output the smallest l-cube dominating set that we found. Obviously this will not be a feasible approach for higher dimensions since the number of subsets are $2^{(2^s)}$, where s is the dimension of the hypercube. The reason for exploring this approach is primarily to find some results that can be used to test more complex approaches, but it also serves the purpose of familiarising oneself with the problem.
For the case $l = 1$, i.e. regular domination, we can narrow the search significantly by using the known domination numbers. Recalling that the result can be improved by at most one for reasons mentioned in section 2.3, we only need to test subsets of size $d-1$, where d is the domination number of the full hypercube. For saving more than one bit, we can take a similar approach by using the results from [8], and test sets of decreasing size until a solution no longer is found.

3.2 Grid relaxation

For the grid relaxation we chose to go with Damaschke's suggestion and adopt a branch and bound approach. The branching is done through two recursive calls, one where the entry is included in the l-cube dominating set, and one where it is not. We followed Damascke's strategy of trying to dominate the central numbers first, by going through the grid points in descending order. The intuition behind this strategy is that since the central points are the largest entries, they are unlikely to be in the solution, and one can eliminate branches with these points earlier. No formal analysis of the running time was done, but experimentation with branching order showed that this approach is noticeably faster than both going through the points in the order they appear in the grid, and smallest entries first.
Most of Damaschke's results are based on 2-dimensional grids, but since he mentions that these are ad hoc results, there is no guarantee that these grids are optimal. We will therefore start by searching for l-cube dominating sets in 2-dimensional grids, and proceed with grids of higher dimensions. There are multiple possible grids for

each hypercube, depending on how one chooses to split the bit string. We examine all possible splits that give a unique grid, i.e. we ignore splits that give the transpose of an already examined grid.
In summary the algorithm performs three steps. Firstly, it calculates whether the sum of the currently selected entries is higher than the currently best known solution, and abandons the branch if it is not. For our initial bounds, we use the known schemes from [8]. Secondly, we test whether the currently selected entries form an l-cube dominating set, and if so we update the bound. Lastly, should the total sum not be too high and no l-cube dominating set be found, we branch and continue the search. Each branch will be followed either until it yields a result or exceeds the current bound, before moving on to the next branch.

4
Results

The following sections presents the results that were found through the approaches described in section 3. Due to the limited time frame of the project, we decided to kill any run that did not terminate within four hours. All of the dominating sets and grids that are mentioned or otherwise implied in this section, can be found in appendix A.

4.1 Exhaustive search

Starting with saving one bit, we first note that the cases $s = 1$ and $s = 2$ are trivial, and that the known domination numbers are already optimal in the punctured hypercube as well. All runs for hypercubes of dimensions three through five terminated successfully in under one second on the system. The resulting schemes are $(5, 1, 3, 2)$, $(11, 1, 4, 3)$ and $(24, 1, 5, 4)$ for the respective instances, which means that we did not improve on the existing schemes.
The run for $s = 6$ did not terminate within the set time constraint and was therefore killed. A quick estimation of the running time shows us that we could probably find a solution for $s = 6$ within a couple of days. We deemed it not worth it considering that even if we solved $s = 6$, the next instance $s = 7$ could take thousands of years in the worst case.
Before proceeding with results for saving more bits, we note that schemes for saving two bits only exist for $s > 4$ and schemes for saving three bits only exist for $s > 6$ [8]. The results from saving one bit tells us that we should be able to find a $(m, 1, 5, 3)$-scheme, and that we will not be able to find a scheme for saving three bits using this approach, since six or more dimensions is deemed infeasible. The run for saving two bits and $s = 5$ gave us a $(19, 1, 5, 3)$-scheme, which is the same result that Damaschke found, showing that the 2-dimensional grid is optimal in this case.

4.2 Grid Relaxation

We were unable to improve the schemes for $s = 5$ and $s = 6$, but came to the same results as Damaschke. In the case of $s = 5$ we found both a 2- and a 3-dimensional grid that result in a $(19, 1, 5, 3)$-scheme, which we know, from our results with exhaustive search, cannot be improved. For $s = 6$ on the other hand, we were unable to find a 2-dimensional grid. We note that Damaschke's $(39, 1, 6, 4)$-scheme does not come from a grid either, but from the doubling lemma. The 3-dimensional grid we found for this scheme reflects the doubling lemma result in that it is the

2-dimensional grid for the $(19, 1, 5, 3)$-scheme repeated twice. We present the 3-dimensional grid as the planes along the z-axis, listed from left to right.

1	3	*3	*1
*2	6	6	*2
*1	*3	3	1

*1	*3	3	1
*2	6	6	*2
1	3	*3	*1

For our next scheme we found our first improvement upon the existing results with a $(81, 1, 7, 5)$-scheme and the following 2-dimensional grid.

*1	*4	6	4	1
*3	12	18	*12	*3
*3	12	18	*12	*3
*1	*4	6	4	1

We were even able to further improve this to a $(82, 1, 7, 5)$-scheme with the following 3-dimensional grid. Note that there might be an even better 3-dimensional grid for the case $s = 7$, since this run was not allowed to finish. The search was left running for longer than four hours, since it had produced intermediate improvements within that time frame, but did not produce any further results.

1	3	3	*1
*2	*6	6	2
*1	*3	3	1

2	6	*6	*2
*4	12	12	*4
*2	6	6	*2

*1	*3	3	1
*2	6	6	*2
1	3	*3	*1

The last two schemes we were able to find within the time constraint are $(169, 1, 8, 6)$ and $(346, 1, 9, 7)$, through the following two 2-dimensional grids.

1	*4	*6	4	*1
4	*16	*24	16	4
6	24	36	24	*6
*4	16	24	*16	*4
*1	*4	6	4	*1

1	*5	*10	*10	5	*1
*4	20	*40	40	20	4
6	30	60	60	*30	*6
*4	*20	40	40	*20	*4
*1	*5	*10	10	*5	*1

We make two noteworthy observations regarding these grids. Firstly, both of these dominating sets contains nodes that are not used to dominate other nodes, e.g. the upper right corner in both grids. For the other schemes we needed to use an element node for the zero string, but in these two schemes we can instead select nodes from the dominating sets since not all of them are used to dominate other nodes. Secondly, these grids are the first non-symmetric solutions that have been found.

5

Discussion

This section serves primarily two purposes. Firstly, we will highlight and discuss some of the interesting aspects about the results that we have found. Secondly, we will bring up areas of future research, both in terms of continuing working with the grid relaxation as well as other approaches.

5.1 The results

The first results of particular interest comes from the initial work with exhaustive search, where we concluded that the $(19, 1, 5, 3)$-scheme is an optimal scheme We were also able to get the same scheme from both the 2- and 3-dimensional grids. This is an interesting result since it might indicate that the grid relaxation is a promising approach.

Another interesting observation is that the grids for the $(169, 1, 8, 6)$-scheme and the $(346, 1, 9, 7)$-scheme are not symmetrical. All prior results, including our result for the smaller instances, come either directly from symmetrical grids, or use the doubling lemma. Due to the naturally symmetric structure of the hypercube, we hoped that we could find symmetric cases when searching for cube dominating sets, since this could have been used to reduce the search space.

Not only are the solutions not symmetrical, but we can barely find discernible patterns at all. In his motivation for the branching order of largest grid points first, Damaschke mentioned that he believes that it is unlikely that the central points are in the cube dominating set, and that we therefore should start by trying to cube dominate these first, as cheaply as possible. This prediction holds for all of the schemes we have found. None of them have included the largest grid points in the cube dominating set, with the exception of the 3-dimensional grid for the $(19, 1, 5, 3)$-scheme that can be found in appendix A, which obviously must contain one of the centre points.

Both the $(169, 1, 8, 6)$-scheme and the $(346, 1, 9, 7)$-scheme also have nodes in their respective cube dominating sets that are not used to dominate other nodes. We wonder if this is a results of the coarseness caused by the grid relaxation, i.e. it is cheaper to just include small entries in the cube dominating sets than to include the larger adjacent larger entries necessary in order to cube dominate them, or if this is the case for optimal cube dominating sets as well.

5.2 Future work

The first natural extension of our work is to move on to grids in dimensions greater than three. We have already seen improvements in the case $s = 7,\ l = 2$, when increasing the dimensions of the grid, but this is far from enough to draw any conclusions. The branch and bound approach is still believed to be a good one, and we suggest further investigation into both better bounds as well as improving the branching orders.
It would also be interesting to work on optimal solutions directly in the hypercubes, and not only to improve the schemes, but in order to get further insight into how good the grid relaxation really is. Although we already have concluded that 2-dimensional relaxations are not optimal for $s > 6$ with the $(82, 1, 7, 5)$-scheme, it may still be the case that low-dimensional grids is a good approach.
Damaschke suggested a dynamic programming approach for finding optimal schemes, where the nodes are ordered into layers based on their Hamming weight, see figure 5.1. The algorithm would then solve this for the first layer(s), and extend the solution layer by layer. However, since the size of the layers grows rapidly we need some additional means of reducing the search space. The intuitive idea would be to use the structure of the hypercube and try to find symmetric cases, for example by adapting heuristics for graph isomorphism. While not conclusive, our results from the larger instances did not give symmetric solutions, which might indicate that this approach may not be as good as was first believed.

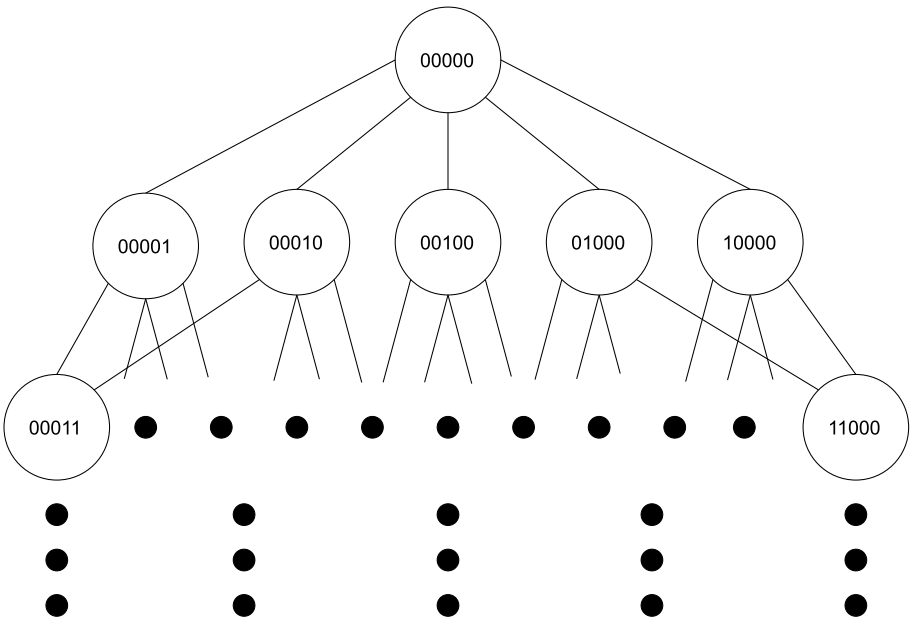

Figure 5.1: The 5-dimensional hypercube, structured into layers where all nodes of layer i has Hamming weight i.

6
Conclusion

We have shown that many of the previously known schemes can indeed be improved. Through a branch-and-bound approach in conjunction with the grid relaxation from [8], we have found several 2- and 3-dimensional grids that lead to improvements in specific schemes. The new schemes and the improvements are summarised in table 6.1.
We have also shown that the optimal grid is not necessarily symmetric, which in turn indicates that using the symmetric structure of the hypercube may not work as well as we had hoped when searching for cube dominating sets. In addition, we have experimented with the branch-and-bound search approach and confirmed Damaschke's idea that largest-grid-point-first seems to work well for finding grids.

Old scheme	New scheme	Improvement ($m_{new} - m_{old}$)
(5,1,3,2)	(5,1,3,2)	0
(11,1,4,3)	(11,1,4,3)	0
(24,1,5,4)	(24,1,5,4)	0
(19,1,5,3)	(19,1,5,3)	0
(39,1,6,4)	(39,1,6,4)	0
(79,1,7,5)	(82,1,7,5)	3
(159,1,8,6)	(169,1,8,6)	10
(319,1,9,7)	(346,1,9,7)	27

Table 6.1: Summary of the new schemes and the improvements on the previously best known results. The first section are schemes for saving one bit and the second for saving two bits.

Bibliography

[1] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh, "Are bitvectors optimal?", *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1723–1744, 2002.

[2] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman, "Exact and approximate membership testers", in *Proceedings of the tenth annual ACM symposium on Theory of computing*, ACM, 1978, pp. 59–65.

[3] R. Pagh, "On the cell probe complexity of membership and perfect hashing", in *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, ACM, 2001, pp. 425–432.

[4] R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets", *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 4, p. 43, 2007.

[5] M. Garg and J. Radhakrishnan, "Set membership with a few bit probes", in *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 2015, pp. 776–784.

[6] ——, "Set membership with non-adaptive bit probes", in *LIPIcs-Leibniz International Proceedings in Informatics*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, vol. 66, 2017.

[7] M. Lewenstein, J. I. Munro, P. K. Nicholson, and V. Raman, "Improved explicit data structures in the bitprobe model", in *European Symposium on Algorithms*, Springer, 2014, pp. 630–641.

[8] P. Damaschke, "Saving probe bits by cube domination", in *Graph-Theoretic Concepts in Computer Science*, A. Brandstädt, E. Köhler, and K. Meer, Eds., 44th International Workshop on Graph-Theoretic Concepts in Computer Science WG 2018, LNCS 11159 (Springer), Cham: Springer International Publishing, 2018, pp. 139–151, ISBN: 978-3-030-00256-5.

A

Dominating sets

A.1 Exhaustive search

The following sets of bit strings are the results of our exhaustive search for dominating sets in the punctured hypercubes. For ease of implementation we chose to use the string of all ones as *the zero string*.

A.1.1 Saving one bit

A.1.1.1 (5,1,3,2)

$\{010, 101\}$

A.1.1.2 (11,1,4,3)

$\{0000, 0110, 1011, 1101\}$

A.1.1.3 (24,1,5,4)

$\{00000, 00001, 00010, 01111, 10111, 11011, 11100\}$

A.1.2 Saving two bits

A.1.2.1 (19,1,5,3)

$\{00010, 00100, 00110, 00111, 01001, 01110, 10001, 10110, 11000, 11001, 11011, 11101\}$

A.2 Grid relaxation

A.2.1 Two-dimensional grids

A.2.1.1 (19,1,5,3)

1	3	*3	*1
*2	6	6	*2
*1	*3	3	1

A.2.1.2 (81,1,7,5)

*1	*4	6	4	1
*3	12	18	*12	*3
*3	12	18	*12	*3
*1	*4	6	4	1

A.2.1.3 (169,1,8,6)

1	*4	*6	4	*1
4	*16	*24	16	4
6	24	36	24	*6
*4	16	24	*16	*4
*1	*4	6	4	*1

A.2.1.4 (346,1,9,7)

1	*5	*10	*10	5	*1
*4	20	*40	40	20	4
6	30	60	60	*30	*6
*4	*20	40	40	*20	*4
*1	*5	*10	10	*5	*1

A.2.2 Three-dimensional grids

The planes along the z-axis (depth) are presented from left to right.

A.2.2.1 (19,1,5,3)

1	3	*3	*1
*1	3	3	*1

*1	3	3	*1
*1	*3	3	1

A.2.2.2 (39,1,6,4)

1	3	*3	*1
*2	6	6	*2
*1	*3	3	1

*1	*3	3	1
*2	6	6	*2
1	3	*3	*1

A.2.2.3 (82,1,7,5)

1	3	3	*1
*2	*6	6	2
*1	*3	3	1

2	6	*6	*2
*4	12	12	*4
*2	6	6	*2

*1	*3	3	1
*2	6	6	*2
1	3	*3	*1

B

Implementation

The main file is the entry point to the program. Its purpose is to parse command line arguments.

src/main.cpp

```
#include "es.hpp"
#include "gr2d.hpp"
#include "gr3d.hpp"

int main (int argc, char** argv) {
    if (argc != 4) {
        cout << "Usage: 'run alg d l'" << endl;
        return 1;
    }

    string algorithm = argv[1];
    int d = stoi(argv[2], nullptr, 10);
    int l = stoi(argv[3], nullptr, 10);

    if (algorithm == "es") {
        ES es(d, l);
        es.run();
    } else if (algorithm == "gr" && l == 2) {
        GR2D gr(d);
        gr.run();
    } else if (algorithm == "gr" && l == 3) {
        GR3D gr(d);
        gr.run();
    }

    return 0;
}
```

The "gp" files define a data structure for grid points, where the numerical value is bundled with the coordinates. They support sorting based on the value, and they are used for defining the branching order.

src/gp.hpp

```
#pragma once

#include <vector>

using namespace std;

class GP {
    public:
        int value;
        vector<int> coords;

        GP(int value, vector<int> coords);
        bool operator < (GP gp);
        bool operator > (GP gp);
};
```

src/gp.cpp

```
#include "gp.hpp"

GP::GP(int value, vector<int> coords) {
    this->value = value;
    vector<int> temp;
    for (int c : coords) {
        temp.push_back(c);
    }
    this->coords = temp;
}

bool GP::operator < (GP gp) {
    return value < gp.value;
}

bool GP::operator > (GP gp) {
    return value > gp.value;
}
```

B.1 Exhaustive search

For ease of implementation we use the string of all ones as the zero string.

src/es.hpp

```
#pragma once

#include <vector>
#include <map>
#include <string>
#include <sstream>
#include <iostream>
#include <boost/dynamic_bitset.hpp>

using namespace std;

class ES {
    private:
        int dimensions;
        int l;
        int nodes;
        int subset_size;
        vector<vector<int>> hypercube;
        vector<map<int,int>> bounds;

        bool search(vector<int> subset, int node);
        bool check(vector<int> subset);
        bool l1(vector<int> subset);
        bool l2(vector<int> subset);
        vector<int> common_elements(vector<int> a,
            vector<int> b);
        void print_result(vector<int> subset);

    public:
        ES(int d, int l);
        void run();
};
```

src/es.cpp

```
#include "es.hpp"

void ES::print_result(vector<int> subset) {
    cout << "Found subset of size: " << subset.
       size() << endl;
    for (int node : subset) {
        cout << node << " ";
```

```
 7      }
 8      cout << endl;
 9  }
10
11  vector<int> ES::common_elements(vector<int> a,
       vector<int> b) {
12      vector<int> temp;
13      for (int x : a) {
14          for (int y : b) {
15              if (x == y) temp.push_back(x);
16          }
17      }
18      return temp;
19  }
20
21  bool ES::l2(vector<int> subset) {
22      vector<bool> dominated(hypercube.size());
23      for (int node : subset) {
24          dominated[node] = true;
25      }
26
27      for (int i = 0; i < (int) hypercube.size(); i
           ++) {
28          if (dominated[i]) continue;
29          vector<int> temp = common_elements(
               hypercube[i], subset);
30
31          for (int n1 : temp) {
32              for (int n2 : temp) {
33                  if (n1 == n2) continue;
34                  vector<int> temp2, temp3;
35                  temp2 = common_elements(hypercube[
                       n1], hypercube[n2]);
36                  temp3 = common_elements(temp2,
                       subset);
37                  for (int c : temp3) {
38                      if (c != i) {
39                          dominated[i] = true;
40                          break;
41                      }
42                  }
43              }
44          }
45      }
46
47      for (bool node : dominated) {
```

```
48          if (!node) return false;
49      }
50
51      return true;
52  }
53
54  bool ES::l1(vector<int> subset) {
55      vector<bool> dominated(hypercube.size());
56      for (int node : subset) {
57          dominated[node] = true;
58          for (int adjacent : hypercube[node]) {
59              dominated[adjacent] = true;
60          }
61      }
62
63      for (bool node : dominated) {
64          if (!node) return false;
65      }
66
67      return true;
68  }
69
70  bool ES::check(vector<int> subset) {
71      switch (l) {
72          case 1: return l1(subset);
73          case 2: return l2(subset);
74          default: return false;
75      }
76  }
77
78  bool ES::search(vector<int> subset, int node) {
79      if (node > (int) hypercube.size()-1) return
           false;
80
81      if ((int) subset.size() == subset_size) {
82          bool flag = check(subset);
83          if (flag) print_result(subset);
84          return flag;
85      }
86
87      vector<int> copy(subset);
88      copy.push_back(node);
89      return search(copy, node + 1) || search(subset
           , node + 1);
90  }
91
```

```
92  ES::ES(int d, int l) {
93      this->dimensions = d;
94      this->l = l;
95      this->nodes = (1 << d)-1;
96      this->bounds = {
97          {{3, 2}, {4, 4}, {5, 7}},
98          {{5, 12}}
99      };
100     this->subset_size = bounds[l-1][d];
101
102     hypercube.resize(nodes);
103     for (int n = 0; n < nodes; n++) {
104         for (int d = 0; d < dimensions; d++) {
105             int tmp = n ^ (1 << d);
106             if (tmp == nodes) continue;
107             hypercube[n].push_back(tmp);
108         }
109     }
110 }
111
112 void ES::run() {
113     vector<int> tmp;
114     while (subset_size > 0 && search(tmp, 0)) {
115         tmp.clear();
116         subset_size--;
117     }
118 }
```

B.2 2-dimensional grids

src/gr2d.hpp

```
 1 #pragma once
 2
 3 #include <iostream>
 4 #include <vector>
 5 #include <map>
 6 #include <algorithm>
 7
 8 #include "gp.hpp"
 9
10 using namespace std;
11
12 class GR2D {
13     private:
14         int dimensions;
15         map<int,int> bounds;
16         vector<vector<vector<int>>> patterns;
17         vector<vector<vector<int>>> grids;
18
19         int fac(int x);
20         int nck(int n, int k);
21         void init_grids();
22         void print_grid(int grid, vector<vector<
               bool>> star);
23         bool check(int grid, vector<vector<bool>>
               star, vector<GP> gps);
24         void search(vector<vector<bool>> star,
               vector<GP> gps, int gp, int grid);
25
26     public:
27         GR2D(int dimensions);
28         void run();
29 };
```

src/gr2d.cpp

```
 1 #include "gr2d.hpp"
 2
 3 // private
 4
 5 int GR2D::fac(int x) {
 6     if (x < 1) return 1;
 7     return x * fac(x-1);
 8 }
```

```
 9
10 int GR2D::nck(int n, int k) {
11     return fac(n)/(fac(k)*fac(n-k));
12 }
13
14 void GR2D::init_grids() {
15     for (int s = 1; s <= (dimensions/2); s++) {
16         vector<vector<int>> grid;
17         vector<int> x_bin;
18         vector<int> y_bin;
19
20         for (int x = 0; x <= s; x++) {
21             x_bin.push_back(nck(s, x));
22         }
23
24         for (int y = 0; y <= dimensions - s; y++)
               {
25             y_bin.push_back(nck(dimensions - s, y)
                   );
26         }
27
28         for (int x : x_bin) {
29             vector<int> temp;
30             for (int y : y_bin) {
31                 temp.push_back(x*y);
32             }
33             grid.push_back(temp);
34         }
35         grids.push_back(grid);
36     }
37 }
38
39 void GR2D::print_grid(int grid, vector<vector<bool
      >> star) {
40     for (unsigned int i = 0; i < grids[grid].size
          (); i++) {
41         for (unsigned int j = 0; j < grids[grid
              ][0].size(); j++) {
42             cout << (star[i][j] ? "*" : " ") <<
                   grids[grid][i][j] << "\t";
43         }
44         cout << endl;
45     }
46     cout << endl;
47 }
48
```

```
49 bool GR2D::check(int grid, vector<vector<bool>>
      star, vector<GP> gps) {
50     for (GP gp : gps) {
51         bool dominated = false;
52         if (star[gp.coords[0]][gp.coords[1]]) {
53             dominated = true;
54         } else {
55             for (vector<vector<int>> pattern :
                  patterns) {
56                 bool match = true;
57                 for (vector<int> coords : pattern)
                       {
58                     int x = coords[0] + gp.coords
                          [0];
59                     int y = coords[1] + gp.coords
                          [1];
60                     if (x < 0 ||
61                         y < 0 ||
62                         x >= (int) grids[grid].
                              size() ||
63                         y >= (int) grids[grid][0].
                              size() ||
64                         !star[x][y])
65                     {
66                         match = false;
67                         break;
68                     }
69                 }
70                 if (match) {
71                     dominated = true;
72                     break;
73                 }
74             }
75         }
76         if (!dominated) return false;
77     }
78     return true;
79 }
80
81 void GR2D::search(vector<vector<bool>> star,
      vector<GP> gps, int gp, int grid) {
82     if (gp >= (int) gps.size()) return;
83
84     int sum = 0;
85     for (unsigned int x = 0; x < star.size(); x++)
           {
```

```
86          for (unsigned int y = 0; y < star[0].size
               (); y++) {
87              sum += (star[x][y] ? grids[grid][x][y]
                    : 0);
88          }
89      }
90      if (sum >= bounds[dimensions]) return;
91
92      if (check(grid, star, gps)) {
93          cout << "Size of set: " << sum << endl;
94          print_grid(grid, star);
95          bounds[dimensions] = sum;
96          return;
97      }
98
99      vector<vector<bool>> copy;
100     for (vector<bool> row : star) {
101         vector<bool> temp(row);
102         copy.push_back(temp);
103     }
104     copy[gps[gp].coords[0]][gps[gp].coords[1]] =
           true;
105     gp++;
106
107     search(copy, gps, gp, grid);
108     search(star, gps, gp, grid);
109 }
110
111 // public
112 GR2D::GR2D(int dimensions) {
113     this->dimensions = dimensions;
114     this->bounds = {
115         {5,12}, {6,24}, {7,48}, {8,96},
116         {9,192}, {10,300}, {11,583}
117     };
118     this->patterns = {
119         {{1,0}, {0,1}, {1,1}},
120         {{1,0}, {0,-1}, {1,-1}},
121         {{-1,0}, {0,1}, {-1,1}},
122         {{-1,0}, {0,-1}, {-1,-1}},
123         {{1,0}, {2,0}},
124         {{0,1}, {0,2}},
125         {{-1,0}, {-2,0}},
126         {{0,-1}, {0,-2}}
127     };
128 }
```

```
129
130 void GR2D::run() {
131     init_grids();
132
133     for (int i = 0; i < (int) grids.size(); i++) {
134         vector<vector<bool>> star;
135         for (vector<int> row : grids[i]) {
136             vector<bool> temp(row.size(), false);
137             star.push_back(temp);
138         }
139
140         vector<GP> gps;
141         for (int x = 0; x < (int) grids[i].size();
                x++) {
142             for (int y = 0; y < (int) grids[i][x].
                   size(); y++) {
143                 GP gp(grids[i][x][y], {x,y});
144                 gps.push_back(gp);
145             }
146         }
147         sort(gps.begin(), gps.end());
148         reverse(gps.begin(), gps.end());
149
150         search(star, gps, 0, i);
151     }
152 }
```

B.3 3-dimensional grids

src/gr3d.hpp

```
#pragma once

#include <iostream>
#include <vector>
#include <map>
#include <algorithm>

#include "gp.hpp"

using namespace std;

class GR3D {
    private:
        int dimensions;
        map<int,int> bounds;
        vector<vector<vector<int>>> patterns;
        vector<vector<vector<vector<int>>>> grids;

        int fac(int x);
        int nck(int n, int k);
        void init_grids();
        void print_grid(int grid, vector<vector<
           vector<bool>>> star);
        bool check(int grid, vector<vector<vector<
           bool>>> star, vector<GP> gps);
        void search(vector<vector<vector<bool>>>
           star, vector<GP> gps, int gp, int grid);

    public:
        GR3D(int dimensions);
        void run();
};
```

src/gr3d.cpp

```
#include "gr3d.hpp"

// private

int GR3D::fac(int x) {
    if (x < 1) return 1;
    return x * fac(x-1);
}
```

```
 9
10 int GR3D::nck(int n, int k) {
11     return fac(n)/(fac(k)*fac(n-k));
12 }
13
14 void GR3D::init_grids() {
15     for (int s1 = 1; s1 <= (dimensions/3); s1++) {
16         for(int s2 = s1+1; s2 <= (2*dimensions/3);
               s2++) {
17             vector<vector<vector<int>>> grid;
18             vector<int> x_bin;
19             vector<int> y_bin;
20             vector<int> z_bin;
21
22             for (int x = 0; x <= s1; x++) {
23                 x_bin.push_back(nck(s1, x));
24             }
25
26             for (int y = 0; y <= s2 - s1; y++) {
27                 y_bin.push_back(nck(s2 - s1, y));
28             }
29
30             for (int z = 0; z <= dimensions - s2;
                  z++) {
31                 z_bin.push_back(nck(dimensions -
                      s2, z));
32             }
33
34             for (int x : x_bin) {
35                 vector<vector<int>> temp1;
36                 for (int y : y_bin) {
37                     vector<int> temp2;
38                     for (int z : z_bin) {
39                         temp2.push_back(x*y*z);
40                     }
41                     temp1.push_back(temp2);
42                 }
43                 grid.push_back(temp1);
44             }
45             grids.push_back(grid);
46         }
47     }
48 }
49
50 void GR3D::print_grid(int grid, vector<vector<
      vector<bool>>> star) {
```

```
51      cout <<
           "-----------------------------------------------"
            << endl;
52      for (unsigned int i = 0; i < grids[grid].size
           (); i++) {
53          for (unsigned int j = 0; j < grids[grid
               ][0].size(); j++) {
54              for (unsigned int k = 0; k< grids[grid
                   ][0][0].size(); k++) {
55                  if (star[i][j][k]) {
56                      cout << "*" << grids[grid][i][
                           j][k] << "\t";
57                  } else {
58                      cout << " " << grids[grid][i][
                           j][k] << "\t";
59                  }
60              }
61              cout << endl;
62          }
63          cout <<
               "-----------------------------------------------"
                << endl;
64      }
65  }
66
67  bool GR3D::check(int grid, vector<vector<vector<
       bool>>> star, vector<GP> gps) {
68      for (GP gp : gps) {
69          bool dominated = false;
70          if (star[gp.coords[0]][gp.coords[1]][gp.
               coords[2]]) {
71              dominated = true;
72          } else {
73              for (vector<vector<int>> pattern :
                   patterns) {
74                  bool match = true;
75                  for (vector<int> coords : pattern)
                        {
76                      int x = coords[0] + gp.coords
                           [0];
77                      int y = coords[1] + gp.coords
                           [1];
78                      int z = coords[2] + gp.coords
                           [2];
79                      if (x < 0 ||
80                          y < 0 ||
```

```
81                          z < 0 ||
82                          x >= (int) grids[grid].
                               size() ||
83                          y >= (int) grids[grid][0].
                               size() ||
84                          z >= (int) grids[grid
                               ][0][0].size() ||
85                          !star[x][y][z])
86                      {
87                          match = false;
88                          break;
89                      }
90                  }
91                  if (match) {
92                      dominated = true;
93                      break;
94                  }
95              }
96          }
97          if (!dominated) return false;
98      }
99      return true;
100 }
101
102 void GR3D::search(vector<vector<vector<bool>>>
       star, vector<GP> gps, int gp, int grid) {
103     if (gp >= (int) gps.size()) return;
104
105     int sum = 0;
106     for (unsigned int i = 0; i < star.size(); i++)
            {
107         for (unsigned int j = 0; j < star[i].size
               (); j++) {
108             for (unsigned int k = 0; k < star[i][j
                   ].size(); k++) {
109                 sum += (star[i][j][k] ? grids[grid
                       ][i][j][k] : 0);
110             }
111         }
112     }
113     if (sum >= bounds[dimensions]) return;
114
115     if (check(grid, star, gps)) {
116         cout << "Size of set: " << sum << endl;
117         print_grid(grid, star);
118         bounds[dimensions] = sum;
```

```
119             return;
120         }
121
122         vector<vector<vector<bool>>> copy;
123         for (vector<vector<bool>> plane : star) {
124             vector<vector<bool>> temp1;
125             for (vector<bool> row : plane) {
126                 vector<bool> temp2(row);
127                 temp1.push_back(temp2);
128             }
129             copy.push_back(temp1);
130         }
131         copy[gps[gp].coords[0]][gps[gp].coords[1]][gps
              [gp].coords[2]] = true;
132         gp++;
133
134         search(copy, gps, gp, grid);
135         search(star, gps, gp, grid);
136     }
137
138     // public
139     GR3D::GR3D(int dimensions) {
140         this->dimensions = dimensions;
141         this->bounds = {
142             {5,12}, {6,24}, {7,48}, {8,96},
143             {9,192}, {10,300}, {11,583}
144         };
145         this->patterns = {
146             {{1,0,0},{2,0,0}},
147             {{-1,0,0},{-2,0,0}},
148             {{0,1,0},{0,2,0}},
149             {{0,-1,0},{0,-2,0}},
150             {{0,0,1},{0,0,2}},
151             {{0,0,-1},{0,0,-2}},
152             {{1,0,0},{0,1,0},{1,1,0}},
153             {{1,0,0},{0,-1,0},{1,-1,0}},
154             {{1,0,0},{0,0,1},{1,0,1}},
155             {{1,0,0},{0,0,-1},{1,0,-1}},
156             {{-1,0,0},{0,1,0},{-1,1,0}},
157             {{-1,0,0},{0,-1,0},{-1,-1,0}},
158             {{-1,0,0},{0,0,1},{-1,0,1}},
159             {{-1,0,0},{0,0,-1},{-1,0,-1}},
160             {{0,1,0},{0,0,1},{0,1,1}},
161             {{0,1,0},{0,0,-1},{0,1,-1}},
162             {{0,-1,0},{0,0,1},{0,-1,1}},
163             {{0,-1,0},{0,0,-1},{0,-1,-1}}
```

```
164     };
165 }
166
167 void GR3D::run() {
168     init_grids();
169
170     for (int i = 0; i < (int) grids.size(); i++) {
171         vector<vector<vector<bool>>> star;
172         for (vector<vector<int>> plane : grids[i])
                {
173             vector<vector<bool>> temp1;
174             for (vector<int> row : plane) {
175                 vector<bool> temp2(row.size(),
                       false);
176                 temp1.push_back(temp2);
177             }
178             star.push_back(temp1);
179         }
180
181         vector<GP> gps;
182         for (int x = 0; x < (int) grids[i].size();
                x++) {
183             for (int y = 0; y < (int) grids[i][x].
                   size(); y++) {
184                 for (int z = 0; z < (int) grids[i
                       ][x][y].size(); z++) {
185                     GP gp(grids[i][x][y][z], {x,y,
                           z});
186                     gps.push_back(gp);
187                 }
188             }
189         }
190         sort(gps.begin(), gps.end());
191         reverse(gps.begin(), gps.end());
192
193         search(star, gps, 0, i);
194     }
195 }
```