UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LUÍS GUSTAVO MÖLLMANN DOS SANTOS

Analysis of Instance Hardness for the Maximally Diverse Grouping Problem and the Iterated Maxima Search Heuristic

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Marcus Ritt

Porto Alegre
December 2020

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patricia Helena Lucas Pranke
Pró-Reitora de Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Sérgio Luis Cechin
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

LIST OF FIGURES

Figure 1.1 Counter example for polynomial algorithm A1 15
Figure 1.2 Counter example for polynomial algorithm A2 15

Figure 2.1 Solution stabilization runs 20

Figure 3.1 ITB vs. number of nodes 29
Figure 3.2 ITB vs. number of groups 31
Figure 3.3 Effect of the stochastic methodology over ITB 34
Figure 3.4 ITB vs. slack 36
Figure 3.5 ITB vs. group size discrepancy 38
Figure 3.6 Number of possible solutions vs. p 38
Figure 3.7 ITB vs. number of edges in unitary tree 40
Figure 3.8 ITB vs. standard deviation of distances 42
Figure 3.9 ITB vs. diversity of distance values 44

LIST OF TABLES

Table 2.1 Comparison between results found by CPLEX and those found by the stabilization procedure. 22

Table 3.1 Solution stabilization summary for the experiment on number of nodes. 30
Table 3.2 Parameters for the experiment on the number of nodes. 30
Table 3.3 Solution stabilization summary for the experiment on number of groups. 32
Table 3.4 Parameters for the experiment on the number of groups. 32
Table 3.5 Parameters for the experiment on slack. 36
Table 3.6 Solution stabilization summary for the experiment on slack. 36
Table 3.7 Solution stabilization summary for the experiment on difference in group sizes. 37
Table 3.8 Parameters for the experiment on the difference in group sizes. 39
Table 3.9 Parameters for the experiment on number of unitary edges forming a tree. 40
Table 3.10 Solution stabilization summary for the experiment on number of unitary edges forming a tree. 41
Table 3.11 Solution stabilization summary for the experiment on standard deviation of distances. 42
Table 3.12 Parameters for the experiment on standard deviation of distances. 43
Table 3.13 Solution stabilization summary for the experiment on diversity of distance values. 44
Table 3.14 Parameters for the experiment on diversity of distance values. 45

LIST OF ABBREVIATIONS AND ACRONYMS

MDGP	Maximally Diverse Grouping Problem
IMS	Iterated Maxima Search
BKV	Best Known Value
ITB	Iterations to BKV
SAT	Boolean Satisfiability Problem
TSP	Travelling Salesperson Problem

CONTENTS

1 INTRODUCTION......7
1.1 Theoretical Hardness......7
1.2 Empirical Hardness......11
1.3 The Maximally Diverse Grouping Problem......12
1.3.1 NP-completeness......13
1.3.2 Multiplicative Invariance......13
1.3.3 An Easy Subcase......14
1.4 Iterated Maxima Search......17
2 METHODOLOGY......19
2.1 Solution Stabilization......19
2.2 Comparison of Solution Stabilization and Exact Results......20
2.2.1 Exact Solutions......20
2.2.2 Comparison of results with CPLEX......22
2.3 Generating Instances......23
2.3.1 Values for n and m......24
2.3.2 Defining the Distances......24
2.3.2.1 Random Integer Distances......24
2.3.2.2 Normally Distributed Distances......24
2.3.2.3 Distances Limited to a Set of Values......24
2.3.2.4 Unitary Distances Forming a Tree......24
2.3.3 Defining Bounds......25
2.3.3.1 Uniform Distribution......27
2.3.3.2 Geometric Distribution......27
3 EXPERIMENTS......28
3.1 Influence of the Number of Nodes......28
3.2 Influence of the Number of Groups......31
3.3 Influence of the Randomness in the Instance Generation Procedure......32
3.4 Influence of the Slack......35
3.5 Influence of the Difference in Group Sizes......37
3.6 Influence of the Number of Unitary Edges Forming a Tree......39
3.7 Influence of the Standard Deviation of Distances......41
3.8 Influence of the Diversity of Distance Values......43
4 CONCLUSIONS AND PERSPECTIVES FOR THE FUTURE......46
BIBLIOGRAPHY......48

1 INTRODUCTION

The Maximally Diverse Grouping Problem (MDGP) requires grouping a set of elements into disjoint subsets (or groups) in order to maximize the overall diversity between elements of the same group, given some measure of diversity (or distance) between those elements. MDGP can model a variety of problems in real life. It can be used, for example, to construct reviewer groups in the process of reviewing an article. We would like the reviewers to have a diverse background to reduce biases and improve the quality of the review (Chen et al. 2011). It is also used for assigning students to work groups, so that they are immersed in a diverse environment. The problem also has many applications in VLSI design, where we need to group highly connected modules onto the same circuit (Weitz and Lakshminarayanan 1997).

We propose to study the instances of the MDGP by relating their features to how long several runs of a heuristic take to agree on a solution for them. We interpret this time as a measure for the hardness of the instances. We chose in particular the Iterated Maxima Search (or IMS) heuristic. Although we treat IMS mostly as a black box and how it works is not too relevant, we must emphasize that measures of instance hardness are always relative to a particular algorithm and in our case it is IMS.

This work is structured as follows. In Sections 1.1 and 1.2, we present notions of theoretical and empirical hardness of problems and their instances. In Section 1.3, we present a formal definition of MDGP and properties. In Section 1.4, we outline the IMS algorithm. In Section 2.1, we describe the measure for hardness used in this work and our methodology for obtaining it. In Sections 2.2.1 and 2.2, we use an optimizer to analyze our methodology. In Section 2.3, we describe how we generate instances for our experiments. Finally, in Chapter 3, we describe the results of these experiments, which are summarized together with points of improvement for this work in Chapter 4.

1.1 Theoretical Hardness

Turing (1937) showed that there are decision problems which cannot be solved by a computer. A *decision* problem is one that asks a “yes” or “no” question. For example, we can ask, given a program and an input to that program, if the program halts on that input. This is known as the Halting Problem. Turing showed that there is no algorithm that could answer this question for any program and input combination, thus making the

problem *undecidable*. Other problems are known to be undecidable, see for example Berger (1966), S. Paterson (1970), Cassaigne et al. (2014), and for a more comprehensive list, Poonen (2012). Problems that ask a "yes" or "no" question which an algorithm can answer are called *decidable*.

If we focus on decidable problems, we can take the number of steps an algorithm performed to solve their instances as a measure of how hard they are. The theory of computational complexity, formalized by Hartmanis and Stearns (1965), studies the resources necessary to solve computational problems (Sipser et al. 2006). The most important resources studied are the number of steps (time) and the amount of memory (space) an algorithm needs to run on a given input. For this introduction, we will refer mostly to time complexity.

The *time complexity* for an algorithm expresses the largest amount of time (steps) it takes to compute an answer to the problem in terms of the input size. For example, performing a linear search on an array of size n takes roughly n steps at most. Performing a binary search on a sorted array of size n takes approximately $\log_2 n$ steps at most. Breaking a numeric password with n digits can take up to 10^n steps. We usually denote worst case complexity using the *Big-O notation*. Let $f(n)$ be the maximum number of steps an algorithm executes on an input of size n. We say that $f(n)$ is $O(g(n))$ if there exist constants c and n_0 such that $f(n) \leq cg(n)$ when $n \geq n_0$. This is called asymptotic complexity, in light of the fact that we only care for the condition when $n \geq n_0$. We would then say that the linear search complexity is $O(n)$ while binary search has complexity $O(\log n)$ (and, by the definition, also $O(n)$, but we usually consider only the "tightest" known complexity). Going back to our search examples, we can say that deciding if an item is in an unsorted list is a problem that belongs to the class of problems solvable in $O(n)$ steps, while doing the same in a sorted list belongs to the class solvable in $O(\log_2 n)$.

It is important to notice that there are also *optimization problems*, which instead of a "yes" or "no" question as in decision problems, ask for a solution that maximizes (or minimizes) some quantity. For example, given a list of cities and the distances between them, the Travelling Salesman Problem (TSP) asks for the shortest route that visits all of the cities exactly once and returns to the starting point. In this case, we have an optimization problem because we want to minimize the path distance. Most optimization problems have a decision version. For instance, TSP's decision version is as follows: given a real number k, is there a tour with length of at most k?

The notion of complexity is used to categorize problems into classes. We call a

complexity class the set of decision problems that can be solved given the same amount of resources. Although this characterization is limited to decision problems, the framework it provides is very expressive and useful for optimization problems too (Arora and Barak 2009). There are two very important complexity classes: P and NP. *Class P* are problems which can be solved in $O(p(n))$ steps, where $p(n)$ is some polynomial function. *NP class* comprises of decision problems whose positive solutions (with a "yes" answer) have "proofs" (or "certificates") of correctness that can be verified in polynomial time. The proof is any succinct information necessary to check the solution is correct. For a problem that asks "is there a solution to this Sudoku grid?", the proof would be the numbers in the solution and their positions in the grid. Notice that, although problems in NP have solutions verifiable in polynomial time, it might not be possible to *find* those solutions in polynomial time. It is easy to notice that $P \subseteq NP$: if we can find a solution in polynomial time, we can also verify that it is correct in polynomial time. A more "machine-based" definition of NP is that it contains problems that can be solved by a non-deterministic Turing machine in polynomial time (NP stands for "non-deterministic polynomial"). A non-deterministic machine is one that can take more than one computational path at a time. Such a machine could enumerate all possible solutions to a problem and use their certificates to determine the right one. Since it could try all of them in parallel, if each verification takes polynomial time, the answer would be found in polynomial time. Of course, these machines do not exist in reality.

In 1971, Cook showed that any problem in NP could be "reduced" to the SAT problem in polynomial time. Given a boolean formula, the SAT problem asks whether there is an assignment for the variables such that the formula evaluates to true. To *reduce* a problem A to a problem B means we translate instances of problem A into instances of problem B, such that the "yes" or "no" answer to B's instance is the same as that for A's (Karp 1972). This means that, if we find an algorithm for solving problem B, we can use that same algorithm for solving problem A. If we can reduce a problem to SAT, we can use SAT's algorithms to solve that problem. But if we can reduce in polynomial time any problem in NP to SAT, then if SAT is found to be in P, all problems in NP will also be in P. In that sense, Cook proved SAT is "at least as hard" as any other problem in NP. We say it is complete for NP, or NP-complete. An *NP-complete problem* is an NP problem to which all problems in NP can be reduced in polynomial time. Problems to which an NP-complete problem can be reduced are called *NP-hard*. NP-hard problems may or may not be decision problems. To this date, decision problems have been either proven to be

in P or proven to be NP-complete, but not both. One of the biggest open questions in computer science is whether NP-complete problems can be solved in polynomial time (that is, if P = NP).

Any problem can be found to be in P if a previously unknown polynomial algorithm is found for that problem. In 1965, Edmonds presented a new algorithm for the Matching Problem. A matching in a graph $G = (V, E)$ is a subset of E in which no two edges meet at the same vertex. The matching problem asks, given a graph and an integer k, if there is a matching on this graph with k edges or more. A brute force algorithm that enumerates all possible subsets of E to find such matching has running time $O(2^{|E|})$. If this was the only known algorithm for that problem, we would say it is in NP, but not necessarily in P. We can verify a positive solution in polynomial time, but not find one. However, the algorithm proposed by Edmonds runs in polynomial time $O(|E||V|^2)$, which means the Matching Problem is indeed in P. In that same paper, he suggested a distinction between "efficient" and "inefficient" algorithms based on whether their complexity is polynomial or not, respectively, remarking that this difference was often very important when practical application was considered.

The line between "efficient" and "inefficient" algorithms in terms of polynomial time complexity is even more useful when we turn to the empirical fact that when a polynomial algorithm is discovered for a problem, its complexity may be high at first (e.g., n^{100}), but optimizations often follow so that this number is decreased significantly (e.g., to n^4) (Arora and Barak 2009, Cormen et al. 2009, Sipser et al. 2006). Using polynomial time complexity as the division between practical and impractical, we would say problems in P are tractable and those that are NP-complete are intractable in scenarios of practical application. However, the classification of problems into complexity classes is done in terms of the worst case performance of their best algorithms, i.e., based on the problem's hardest possible instance. The analysis is usually done in terms of a simple measure of instance size, but as problems and instances become more complex, the formal hardness analysis becomes much harder.

In practice, one usually deals with certain "typical" instances of a problem. We can take the Vertex Cover Problem as an example. The problem asks, given a graph, if a subset of size k of its n vertices exists such that all edges touch at least one of the vertices in this subset. There is an algorithm that answers the question in $O(kn + (4/3)^k k^2)$ (Balasubramanian, Fellows, and Raman 1998). It is an exponential algorithm, which is expected from an NP-complete problem such as the Vertex Cover Problem. However, if

we are dealing with instances of small values of k, the algorithm runs quickly for any n. Parameterized complexity is a sub-field of complexity theory which divides the input of a problem into size and parameters (Downey and Fellows 1999). In our example, n would be the size and k a parameter. This theory formalizes the notion that when instances are confined to a certain parameter ranges, the problem might be tractable, although the general problem is NP-hard. However, we may not be interested in a simple parameter such as size or we may want an average complexity instead of worst case complexity. As our questions about instance structure become more complex, calculating hardness exactly and formally becomes more difficult. In some cases, we may turn to alternative ways of analyzing hardness.

1.2 Empirical Hardness

Many typical instances of NP-hard problems that arise in practice can be solved rather quickly. For example, polynomial-time algorithms can be used to solve the Graph Coloring Problem with high probability of optimality, even though the problem is NP-hard (Turner 1988, Coja-Oghlan, Krivelevich, and Vilenchik 2010). SAT is another NP-complete problem that is often considered tractable for many propositional formulas – sometimes with millions of variables – derived from industrial applications (Vizel, Weissenbacher, and Malik 2015). The Knapsack Problem is solved so unexpectedly quickly in its domains of application that finding its hard instances is actually a subject of study (Pisinger 2005). In contrast, the Vehicle Routing Problem is known for being hard to solve exactly even for what would be considered a small instance in many applications (Laporte 1992).

Cheeseman, Kanefsky, and Taylor (1991) showed that some measures of structure in the instances correlate with what we can call the *empirical hardness*, a measure of how hard certain instances are based on past algorithm performance. Among other examples, they generated random graphs and used an algorithm to decide whether there was a Hamiltonian cycle in the graph or not. A Hamiltonian cycle is a closed path through all nodes in a graph where no node is visited twice. The problem of determining if a graph contains a Hamiltonian cycle is NP-complete. What Cheeseman et al. found was that the number of steps the algorithm took to compute an answer to each of the random graphs (their measure of empirical hardness) has a sharp change when plotted against the average degree (i.e., how many edges per node the graph has). The graph average degree

of the graph is then called the *order parameter* that governs a hard-easy phase transition of the problem, much like we use the density to tell apart liquid-vapor phases of water in physics. In that same study, they showed the same behavior occurred for other NP-complete problems (each with their own order parameter) such as graph coloring, SAT and TSP.

Much research has been done on the effect of the structure of the instance and their hardness (e.g., Prosser 1996, Mitchell, Selman, and Levesque 1992, Achlioptas, Naor, and Peres 2005, Smith-Miles, Hemert, and Lim 2010, Kromer, Platos, and Kudelka 2017). To cite an example, Kanda et al. 2011 analyzed how the performance of several algorithms for solving the TSP related to graph features. They study the influence of several parameters, such as the number of vertices, number of edges, average of edge costs, etc.

Characterizing the structure of hard instances can be used, for example, to build algorithm portfolios, a method that combines several algorithms and chooses the one that is most likely to be efficient for the instance at hand based on its features (Gomes and Selman 2001). SATzilla (Xu et al. 2008) is an example of a tool for building such portfolios for the SAT problem. The tool takes as input a set of instances and a set of algorithms. It then constructs a portfolio by optimizing some objective function, like runtime, solution quality, etc. On a new instance, this portfolio is used to choose which of the input algorithms to use. The process of choosing the algorithm uses an empirical hardness model, a computationally inexpensive predictor of the algorithm performance based on the instance features and on the past performance of those algorithms. As a bonus, the study of instance features that relate to hardness for an algorithm also aids us to build better algorithms by making us understand better the problem structure.

1.3 The Maximally Diverse Grouping Problem

The Maximally Diverse Grouping Problem (MDGP) consists of grouping nodes in an edge-weighted graph maximizing the "in-group" weights while respecting some restrictions on group sizes. It is usual to call those weights *distances*, representing a dissimilarity measure (hence the name MDGP). More formally, using $[k]$ to denote the set $\{1, 2, \dots, k\}$, consider the edge-weighted, complete graph $G = (V, E)$, where $V = [n]$ are the nodes (or vertices), $E = \{\{i, j\} \mid i, j \in V, i \neq j\}$ are the edges and $\boldsymbol{D}$ is a symmetric matrix whose entries $d_{ij} \geq 0$ represent the edge weights between nodes i and

j for $i, j \in [n]$, with diagonal entries equal to zero. The problem is to partition V into into m disjoint sets, forming a solution $S = \{V_g \mid g \in [m]\}$, while maximizing

$$\sum_{g \in [m]} \sum_{i,j \in V_g} d_{ij}. \tag{1.1}$$

Additionally, the size of each subset V_g is constrained by lower bounds $\boldsymbol{a} = (a_1, a_2, \ldots, a_m)$ and upper bounds $\boldsymbol{b} = (b_1, b_2, \ldots, b_m)$, such that the solution must have $a_g \leq |V_g| \leq b_g$ for all $g \in [m]$.

1.3.1 NP-completeness

The decision version of MDGP is NP-complete. It asks, given an instance, if there is a solution with a value of at least k. To prove that this problem is NP-complete, we must (1) prove a solution is verifiable in polynomial time and that (2) we can reduce another NP-complete problem to it.

Proof. For the first step, note that, if given a solution for an instance of the MDGP, we can verify it has a value of k or more in polynomial time by simply performing the summation in (1.1) and checking if the number of nodes in each groups is within bounds. For the second part of the proof, consider the problem of m-Dimensional Matching on a graph (mDM). mDM is NP-complete (Feo and Khellaf 1990) and asks the following question. Given an edge-weighted, complete graph $G = (V, E)$, where the number of vertices $|V| = mr$ for some integer $r > 2$, is there a partition of V into m groups of r vertices each, such that the sum of the weights of all edges between vertices in the same group is k or more? We can see that this is a special case of MDGP when $a_g = b_g = r$, for $r > 2$. ■

1.3.2 Multiplicative Invariance

MDGP has the property that, for a given instance, we can multiply all of its distances d_{ij} by some positive constant a without changing the optimal partitioning of nodes. This property will be useful in the analysis at Section 1.3.3. The proof is quite simple, we just notice that any solution S for the original instance will have a value for the new instance that is equal to

$$a \times \sum_{g\in[m]} \sum_{i,j\in V_g} d_{ij}.$$

That is, the order of all solutions when ranked by quality is unchanged, but the absolute value of their quality is multiplied by a.

1.3.3 An Easy Subcase

We found an easy subcase of the MDGP that can be solved in polynomial time. It occurs when all distances are equal. If distances are all zero, any solution is trivial and has value zero. Otherwise, we can use the multiplicative invariant to normalize them such that all $d_{ij} = 1$, for all $i, j \in [n]$. For this simple subcase, we do not need to distinguish between nodes, so we can refer to a solution simply by the number of nodes in each group. Thus, for now, we will refer to the number of nodes in group g as x_g and to all of them them as the vector $\boldsymbol{x}$. Let us consider two simple algorithms for this subcase:

A1 Satisfy the lower bound requirements of all groups by setting $x_g = a_g$, and then assign the remaining $n - \sum_g a_g$ items in order of non-decreasing lower bounds a_g.

A2 Satisfy the lower bound requirements of all groups by setting $x_g = a_g$, and then assign the remaining $n - \sum_g a_g$ items in order of non-decreasing slacks $b_g - a_g$.

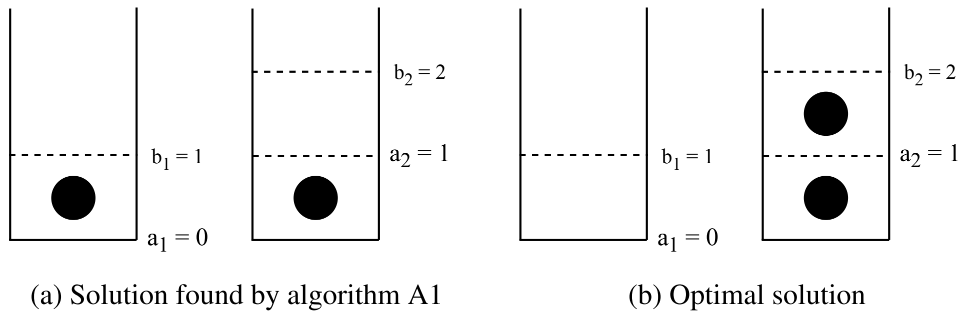
The following two examples show that these algorithms do not always yield the optimal solution. In the case of A1, consider the instance where $m = 2$, $n = 2$, $\boldsymbol{a} = (0, 1)$ and $\boldsymbol{b} = (1, 2)$. Following the algorithm, we arrive at the solution $\boldsymbol{x} = (1, 1)$ (Figure 1.1a), while the optimal solution in this case would be $\boldsymbol{x} = (0, 2)$ (Figure 1.1b). For A2, consider the instance where $m = 2$, $n = 2$, $\boldsymbol{a} = (0, 1)$, but $\boldsymbol{b} = (100, 2)$. Following the algorithm, we arrive at the solution $\boldsymbol{x} = (1, 1)$ (Figure 1.2a), while the optimal solution is actually $\boldsymbol{x} = (0, 2)$ (Figure 1.2b).

We next state optimality conditions and then show how to compute an optimal solution in polynomial time.

Proposition 1.3.1 *At optimality, if there are two groups g_1 and g_2 with sizes x_1 and x_2, if $x_1 \geq x_2$, then $x_1 = b_1$ or $x_2 = a_2$.*

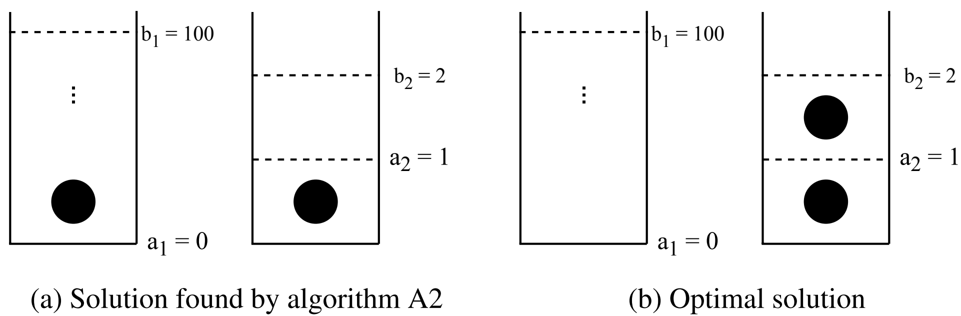
Proof. Suppose our proposition is false and we have an optimal solution where $x_1 < b_1$ and $x_2 > a_2$, but $x_1 \geq x_2$. In that case, we could remove one node from the second group and add it to the first one. Let c be the part of the solution value contributed

Figure 1.1: Counter example for polynomial algorithm A1



(a) Solution found by algorithm A1 (b) Optimal solution

Figure 1.2: Counter example for polynomial algorithm A2



(a) Solution found by algorithm A2 (b) Optimal solution

by the groups not involved in the exchange (i.e., $g \neq 1, 2$), and let $x'_1 = x_1 + 1$ and $x'_2 = x_2 - 1$ in the new solution. The new solution has value

$$\begin{aligned} c + \binom{x'_1}{2} + \binom{x'_2}{2} &= c + \binom{x_1 + 1}{2} + \binom{x_2 - 1}{2} \\ &= c + \binom{x_1}{2} + x_1 + \binom{x_2}{2} - (x_2 - 1) \\ &= c + \binom{x_1}{2} + \binom{x_2}{2} + x_1 - x_2 + 1 \\ &> c + \binom{x_1}{2} + \binom{x_2}{2} \end{aligned}$$

That is, by reassigning one node, we get a better solution, which is in contradiction with the optimality of x_1 and x_2 assumed at the beginning. ■

This tells us that in an optimal solution, if $x_1 \geq x_2$, then either $x_1 = b_1$ or $x_2 = a_2$. That is, if one group has a size larger than another, either the first is at its upper bound or the second at its lower bound. Therefore, an optimal solution must have every group – except one – at either its lower or upper bound. This suggests a dynamic programming approach to compute the optimal solution. Consider $D(k, g)$ the maximum value obtained

in a solution by distributing k nodes to groups $g, \ldots, m$. Because of our proposition, the distribution can leave at most one group with $x_g \neq a_g$ or $x_g \neq b_g$. Let us define now that only the last group is allowed to be in this situation when we are computing $D(k, g)$. We will deal with the consequences of this restriction later. The optimal solution in that case has a value $D(n, 1)$, i.e., the value when we distribute n nodes into groups $1, \ldots, m$. We can make the following conclusions about D:

- The value of $D(k, m)$ is the maximum value we can get when distributing k nodes into only one group, the "last" one (group m). If $a_m \leq k \leq b_m$, then all k nodes fit into the group and $D(k, m) = \binom{k}{2}$. If $k < a_m$ or $k > b_m$, we define $D(k, m) = -\infty$, since the nodes do not fit and the subproblem is invalid.
- The value of $D(k, g)$, when $g < m$, is the maximum value we can get by distributing k nodes into groups $g, \ldots, m$. When $k < a_g$, there are not enough nodes to satisfy the lower bound of group g, so a solution is impossible and we define $D(k, g) = -\infty$ in this case. When $k \geq a_g$, we have enough nodes to satisfy its lower bounds. By our proposition, optimal solutions have all but one group with the number of elements equal to the upper or lower bounds. By our definition of D, only group m is allowed to be in this situation. Therefore, if $k \geq a_g$, we have two options:
  - First, we can assign a_g nodes to group g, in which case the best solution has value $\binom{a_g}{2} + D(k - a_g, g + 1)$. If $k < b_g$, this is our first and only option because of our proposition that says x_g must be either a_g or b_g.
  - Second, we may also have the option to assign b_g nodes to this group if $k \geq b_g$. The best solution in this case has a value of $\binom{b_g}{2} + D(k - b_g, g + 1)$.

  Therefore, we arrive at the following expressions for D:

$$D(k, m) = \begin{cases} \binom{k}{2}, & a_m \leq k \leq b_m, \\ -\infty, & \text{otherwise}, \end{cases} \tag{1.2}$$

  and, for $i < m$,

$$D(k, i) = \begin{cases} -\infty, & k < a_g, \\ \binom{a_g}{2} + D(k - a_g, g + 1), & a_g \leq k < b_g, \\ \max\left\{ \binom{a_g}{2} + D(k - a_g, g + 1), \binom{b_g}{2} + D(k - b_g, g + 1) \right\}, & k \geq b_g. \end{cases} \tag{1.3}$$

We can compute $D(n, 1)$ in polynomial time and space $O(mn)$ via dynamic programming. As we can see from its definition, $D(k, g)$ is relative to a particular ordering of the groups. Only the last group is allowed to have the number of elements not equal to its lower or upper bound. This means we have to repeat this process m times, each time leaving a different group as the last group (of index m) to find the maximum value. This yields an algorithm that runs in $O(m^2 n)$ time and $O(mn)$ space.

1.4 Iterated Maxima Search

Iterated Maxima Search (IMS) is a heuristic search to solve MDGP instances (Lai and Hao 2016). IMS works by improving upon an initial solution doing the following steps:

1. Starting from the current solution, perform a local search and obtain a new current solution.
2. Perturb the current solution to obtain a new current solution.
3. Repeat until a cutoff time is reached.

For the purposes of this research, IMS can be treated as a black box, therefore we only give a brief description of it. For more details, we advise looking into the original paper by Lai and Hao (2016). Algorithm 1 describes the IMS procedure, which consists of two levels of local search and perturbation. The lower level search (Algorithm 2) slightly perturbs the current solution and performs a complete local search in its neighborhood. Meanwhile, the upper level search (Algorithm 1) strongly perturbs the current solution and calls the lower level search.

Algorithm 1: IteratedMaximaSearch

input : I, an instance of the MDGP
α, the depth of Maxima Search
η_s, the magnitude of the strong perturbation
t_{max}, the cutoff time for the algorithm
output: S^*, the best solution found for I
$S \leftarrow$ `InitialSolution(`I`)`
keep the best solution found in S^*
while *runtime* $< t_{max}$ **do**
  $S \leftarrow$ `MaximaSearch(`S, α`)`
  $S \leftarrow$ `StrongPerturbation(`S, η_s`)`
end

Algorithm 2: MaximaSearch

```
input  : α, the depth of Maxima Search
         S, the initial solution
output: S*, the best solution found for I
S ← CompleteLocalSearch(S)
keep the best solution found in S*
while iterations without improvement < α do
    S ← WeakPerturbation(S)
    S ← CompleteLocalSearch(S)
end
```

The algorithm `InitialSolution`(I) returns the best solution found by running `CompleteLocalSearch` on 10 random solutions for I and returning the best one found. The algorithm `CompleteLocalSearch`(S) finds a locally optimal solution by making a complete search over two neighborhoods of S. The first neighborhood, called $N_1(S)$, consists of all solutions obtained by transferring any one node in S to a different subset of V, respecting the group boundaries. This is commonly referred to as a "shift" neighborhood. The second neighborhood, $N_2(S)$, consists of all solutions obtained by exchanging any two nodes of different subsets in S. This is referred to as a "swap" neighborhood.

The two perturbation methods also make use of $N_1(S)$ and $N_2(S)$. The algorithm `StrongPerturbation`(S) starts from a solution S and chooses a random solution in $N_1(S) \cup N_2(S)$ to be the new solution. It repeats this for η_s steps, where η_s is called the strength of the strong perturbation. Lai and Hao (2016) empirically arrive at the values for η_s, setting it to n/m if $n < 400$ and to $1.5 \times n/m$ otherwise. The `WeakPerturbation`(S) algorithm also starts from a solution S and repeats the following steps η_w times:

1. Select at random a solution in $N_1(S) \cup N_2(S)$ to be the new current solution S.
2. Pick the best solution S' of $|V|$ random solutions in $N_1(S) \cup N_2(S)$. If S' has greater value than S, S' becomes the new current solution.

The value η_w is called the strength of the weak perturbation and is set to n by Lai and Hao (2016). In this work, we use the same values of η_s and η_w as the original authors.

2 METHODOLOGY

We use the number of iterations IMS takes to find a solution for an instance of the MDGP as a measure for how hard that instance is for the algorithm. We use the following methodology in order to find features of the instance that influence this number.

In Section 2.1, we describe the process we use to measure hardness, called "solution stabilization". In Section 2.2, we analyze the stabilization process by comparing it's results with the results of an optimizer for instances that it can solve exactly in feasible time. In Section 2.3, we describe a method to generate our own instances in order to have more control over their features. With that, we can keep parameters fixed and only vary the ones we are interested in seeing the influence over hardness. Finally, in Chapter 3, we present the experimental results regarding the impact of instance features.

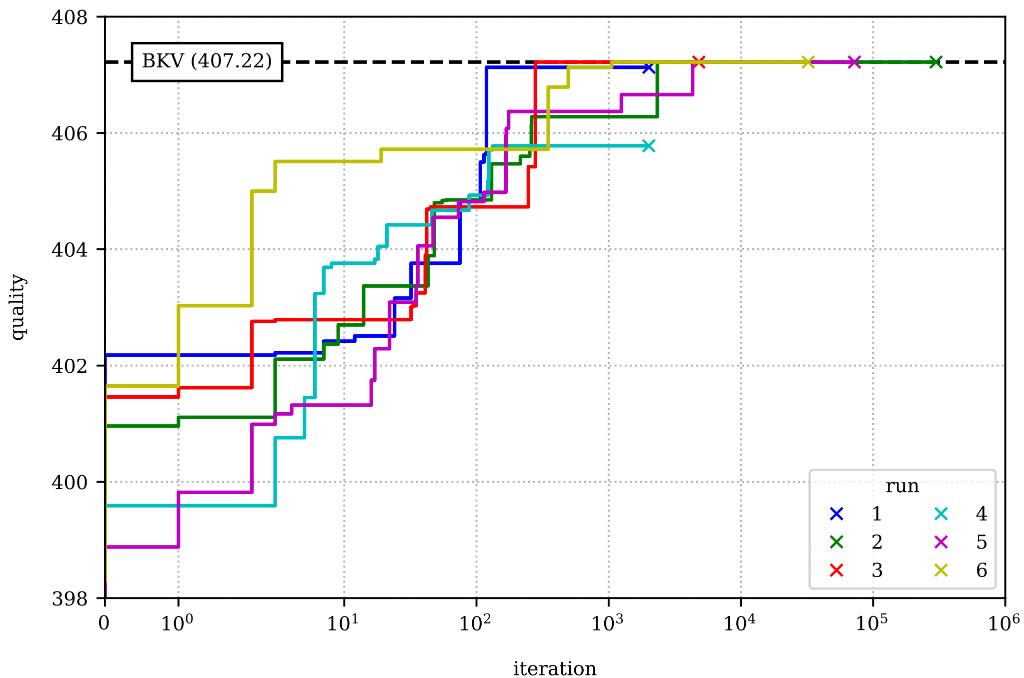
2.1 Solution Stabilization

To gauge how hard it is for IMS to find a solution, we use a procedure we call *solution stabilization*. In each run of solution stabilization, we take an instance of the MDGP and let IMS run as long as it is making progress. We stipulate that the algorithm stops making progress when the last iteration that found a new, better solution is 15 times smaller than the current iteration. We use this number of iterations as a measure of hardness. We use iterations instead of absolute time because it is independent of the computing environment.

Figure 2.1 shows 6 runs of IMS for the same instance with $n = 100$ and $m = 8$. The figure shows the value for the best solution as a function of the number iterations of the main loop of Algorithm 1. Note the x-axis has a logarithmic scale. We call the highest value found by any run the *best known value* (BKV). For most instances in this work, a great majority of the runs will converge to the BKV. We can also see the curves have different lengths, as indicated by the X marks. This is a result of our stopping criteria, which makes all runs have their plateaus 15 times longer than the total length of the curve on the x-axis. For those runs that reached the BKV, we call the total amount of iterations taken to reach that value *iterations to BKV* (ITB). Those that did not reach the BKV are analyzed separately in cases where their number is significant.

From several runs of solution stabilization, we can extract statistical measures of the ITB. We take the median as a measure of how hard it is for IMS to find a good solution

Figure 2.1: 6 runs of the stabilization procedure for an instance of size $n = 100$ and $m = 8$.



for that instance. It is clear that not reaching the BKV means the solution found is sub-optimal. However, the BKV is not necessarily optimal. In the next sections, we will compare the solution stabilization method with exact solutions for small instances of the MDGP to gain confidence that the BKV represents a good solution.

2.2 Comparison of Solution Stabilization and Exact Results

For small enough instances, we can find optimal solutions for MDGP using a integer programming solver. In order to gain some confidence on the results produced by the solution stabilization procedure, we compare its output with the exact results found by the CPLEX solver.

2.2.1 Exact Solutions

We can model the MDGP as a quadratic integer programming problem as follows.

$$\begin{aligned}
\text{maximize} \sum_{g\in[m]} \sum_{i\in[n-1]} \sum_{j\in[i+1,n]} d_{ij} x_{ig} x_{jg} & & (2.1)\\
\text{subject to} \sum_{g\in[m]} x_{ig} = 1, & \quad i \in [n], & (2.2)\\
a_g \le \sum_{i\in[n]} x_{ig} \le b_g, & \quad g \in [m], & (2.3)\\
x_{ig} \in \{0,1\}, & \quad i \in [n], g \in [m]. & (2.4)
\end{aligned}$$

Here x_{ig} is a binary variable – as per Constraint (2.4) – that is 1 if node i belongs to group g and 0 otherwise. Function (2.1) is the quality of the solution, which we want to maximize. Constraint (2.2) ensures each node is in only one group. Constraint (2.3) makes sure upper and lower bounds are respected. In this formulation, the problem has nm variables and $n+m$ constraints. To run this on a integer programming solver, we need to eliminate the quadratic objective function in (2.1). We can reformulate the problem by introducing a new variable $y_{ijg} = x_{ig}x_{jg}$ and rewrite the objective function as

$$\sum_{g\in[m]} \sum_{i\in[n-1]} \sum_{j\in[i+1,n]} d_{ij} y_{ijg}.$$

In our model, it suffices to add the constraints that $y_{ijg} \le x_{ig}$ and $y_{ijg} \le x_{jg}$ to represent the relation that $y_{ijg} = x_{ig}x_{jg}$, since if either x_{ig} or x_{jg} are 0, y_{ijg} is also 0. Also, if x_{ig} and x_{jg} are both 1, the objective to maximize makes sure y_{ijg} is 1. This yields the constraints

$$\begin{aligned}
\sum_{g\in[m]} x_{ig} = 1, & \quad i \in [n],\\
a_g \le \sum_{i\in[n]} x_{ig} \le b_g, & \quad g \in [m],\\
y_{ijg} \le x_{ig}, & \quad i \in [n], j \in [n], g \in [m],\\
y_{ijg} \le x_{jg}, & \quad i \in [n], j \in [n], g \in [m],\\
y_{ijg} \in \{0,1\}, & \quad i \in [n], j \in [n], g \in [m],\\
x_{ig} \in \{0,1\}, & \quad i \in [n], g \in [m].
\end{aligned}$$

In this formulation, there are $n^2m + nm$ variables and $2n^2m + m + n$ constraints.

This is the formulation used in the next section when we compare exact, optimal results and those of the stabilization procedure detailed in Section 2.1.

2.2.2 Comparison of results with CPLEX

We use CPLEX 12.9 – a software for solving integer programming problems – to find exact solutions and gaps, and then compare them to the output of our solution stabilization process. Since finding the exact optimal solution is costly, we are only able to run CPLEX for relatively small instances. The comparison of the results for both methods is shown in Table 2.1.

Table 2.1: Comparison between results found by CPLEX and those found by the stabilization procedure.

m	n	stabilization	CPLEX	difference (%)	CPLEX gap (%)	time (s)
2	5	**2.67**	**2.67**	0.00	0.00	0.01
2	10	**13.15**	**13.15**	0.00	0.00	0.07
2	15	**30.12**	**30.12**	0.00	0.00	0.14
2	20	**53.01**	**53.01**	0.00	0.00	0.98
2	25	**85.28**	**85.28**	0.00	0.00	10.65
2	30	**119.90**	**119.90**	0.00	0.00	80.59
2	35	**160.35**	**160.35**	0.00	0.00	1684.7
4	5	**1.00**	**1.00**	0.00	0.00	0.01
4	10	**6.83**	**6.83**	0.00	0.00	0.37
4	15	**15.37**	**15.37**	0.00	0.00	17.95
4	20	**28.81**	**28.81**	0.00	0.00	454.42
4	25	**46.87**	45.49	2.94	64.93	mem. lim.
4	30	**67.41**	64.04	5.00	103.31	mem. lim.
4	35	**90.64**	85.18	6.02	129.68	mem. lim.
5	5	**0.00**	**0.00**	0.00	0.00	0.0
5	10	**4.44**	**4.44**	0.00	0.00	0.7
5	15	**11.69**	**11.69**	0.00	0.00	11.4
5	20	**23.18**	**23.18**	0.00	0.00	1569.38
5	25	**36.94**	34.64	6.23	94.06	mem. lim.
5	30	**54.44**	50.81	6.67	130.68	mem. lim.
5	35	**74.48**	70.26	5.67	158.33	mem. lim.
8	10	**1.96**	**1.96**	0.00	0.00	0.1
8	15	**6.27**	**6.27**	0.00	0.00	7.47
8	20	**14.14**	14.03	0.78	117.53	mem. lim.
8	25	**22.58**	21.50	4.78	189.74	mem. lim.
8	30	**35.14**	31.81	9.48	279.28	mem. lim.
8	35	**48.04**	40.64	15.40	376.99	mem. lim.

Each row of the table represents one instance of the MDGP. It shows the number of groups m and number of nodes n for each of them. The next columns show the BKV found by the stabilization procedure, the quality of the solution found by CPLEX, and the relative difference between them. For each of the instances in the table, all 15 runs of solution stabilization converged to the BKV. Since CPLEX may stop before it can prove optimality of the solution, we also display the gap between the best value and the upper bound for the optimal solution it could find. The gap is shown as a percentage of that best value. For example, for $m = 4$ and $n = 25$, the best solution found by CPLEX had a value of 46.87, and the upper bound found for the optimal solution was 64.93% greater than that. The last column in the table shows the time it took for CPLEX to find such results. The solution stabilization procedure ran in less than a second for all these small instances.

We can see from the table that the difference between the results from the stabilization and CPLEX are zero for all values where the CPLEX gap is zero. That is, for those instances, the BKV is the optimal value. We assume that this general behavior continues for larger instances, meaning the stabilization result is reasonably close to the optimal solution if many runs converge to the BKV.

2.3 Generating Instances

In order to have control over instance features, we generate the instances through the following steps:

1. Choose values for n and m.
2. Define the distance matrix $\boldsymbol{D}$.
3. Define lower and upper bounds $\boldsymbol{a}$ and $\boldsymbol{b}$.

We would like to be in full control of how bounds are distributed among groups, and also how the distances are generated. Sections 2.3.1, 2.3.2 and 2.3.3, respectively, go over each of these three steps in more detail.

2.3.1 Values for n and m

We must choose a value for the number of nodes n and number of groups m our instance will have. We can pick any positive values. The instances in this work usually have n a few times larger m, such that there are at least a few nodes per group, which is often the case in practical applications.

2.3.2 Defining the Distances

To define the distance matrix $\boldsymbol{D}$ of our instance, we only need to make sure is that $d_{ij} \geq 0$ for $i, j \in [n]$. We define the algorithm used to generate those distances as a parameter for our process of generating instances. We call this parameter the *distance generation algorithm*. We now review the algorithms we use in this work.

2.3.2.1 Random Integer Distances

Algorithm `GenerateRandomIntegerDistances`(v_{min}, v_{max}) simply generates distances by assigning a uniform random integer in the range $[v_{min}, v_{max}]$ to each edge weight.

2.3.2.2 Normally Distributed Distances

Algorithm `GenerateRandomIntegerDistances`(μ, σ) is similar to the previous one, but samples from a normal distribution instead of a uniform distribution with mean μ and standard deviation σ.

2.3.2.3 Distances Limited to a Set of Values

Algorithm `GenerateDistancesLimitedToValues`(c) limits the number of possible values that distances can have to only c different values, and assigns those values randomly to all edges in the same ratio (or as close as possible) to all edges.

2.3.2.4 Unitary Distances Forming a Tree

When two nodes have distance 0 between them, it does not matter for the final solution if they are in the same group or not, but it does when the distance between them

is positive. To analyze how this connection between nodes plays into the hardness of the instance, Algorithm 3 generates instances that have edges of weights 0 or 1. It has the number of unitary edges N_u as a parameter and generates the distances by performing the following steps. It repeatedly connects a random node that already has unitary edges with another random node that has not, thus forming a tree if we take into account only unitary edges. It stops once the number of unitary edges is equal to N. Notice that for simplicity, we implicitly assume that d_{ji} is updated whenever d_{ij} is. We hope to see the effect of this "dependency" between them as the number of unitary edges increases.

Algorithm 3: GenerateDistancesWithUnitaryTree

input : N_u, the number of unitary edges
output: $\boldsymbol{D}$, the matrix of distances
$d_{ij} \leftarrow 0, \forall i, j$
$K \leftarrow \{1\}$
while $|K| < N_u + 1$ **do**
  $i \leftarrow$ a random node in K
  $j \leftarrow$ a random node not in K
  $d_{ij} \leftarrow 1$
  add j to K
end

2.3.3 Defining Bounds

We must define the lower and upper bounds $\boldsymbol{a}$ and $\boldsymbol{b}$ of our instance. In order to do that, we partition n into m capacities $\boldsymbol{s} = (s_1, s_2, \ldots, s_m)$, called *space* of each group, such that the total space is $\sum_g s_g = n$. We then define a *slack up* as $\boldsymbol{u} = (u_1, u_2, \ldots, u_m)$ and a *slack down* as $\boldsymbol{l} = (l_1, l_2, \ldots, l_m)$ such that $\boldsymbol{a} = \boldsymbol{s} - \boldsymbol{l}$ and $\boldsymbol{b} = \boldsymbol{s} + \boldsymbol{u}$. The values in $\boldsymbol{a}$ and $\boldsymbol{b}$ have the following restrictions:

$$\sum_{g \in [m]} a_g \leq n, \tag{2.5}$$

$$\sum_{g \in [m]} b_g \geq n, \tag{2.6}$$

$$a_g \leq b_g, \qquad \forall g \in [m], \tag{2.7}$$

$$a_g \geq 0, \qquad \forall g \in [m]. \tag{2.8}$$

Constraint (2.5) ensures there are enough nodes to fill the groups. We satisfy that

constraint by making sure the slack down is non-negative ($l_g \geq 0$):

$$\sum_{g \in [m]} a_g = \sum_{g \in [m]} s_g - l_g \leq \sum_{g \in [m]} s_g = n$$

Constraint (2.6) guarantees there is enough overall space for the nodes. We satisfy those constraints by making sure the slack up also is non-negative ($u_g \geq 0$):

$$\sum_{g \in [m]} b_g = \sum_{g \in [m]} s_g + u_g \geq \sum_{g \in [m]} s_g = n$$

Constraint (2.7) ensures the lower bounds are smaller than the upper bounds, which is ensured by our conditions that $l_g \geq 0$ and $u_g \geq 0$:

$$a_g = s_g - l_g \leq s_g + u_g = b_g$$

Finally, Constraint (2.8) can be satisfied by making sure $l_g \leq s_g$:

$$a_g = s_g - l_g \geq 0$$

We generate the upper and lower bounds of our instances indirectly by distributing a total amount of space n into $\boldsymbol{s}$. How we distribute this quantity is described by an algorithm that we will call the *space distribution algorithm*, which defines how we distribute the total space n between the m groups.

After that step, we must choose a value for the total slack up $U = \sum_g u_g$ and the total slack down $L = \sum_g l_g$ and distribute these two quantities among the m groups. The distribution of slack up into u_g and slack down into l_g is described by two algorithms: *slack up distribution algorithm* and *slack down distribution algorithm*. Similar to the space distribution algorithm, those algorithms define how we distribute the total slack up or down into each group.

Notice that an algorithm that distributes a total space of n into m groups can also be used to distribute the total slack up U and down L into those m groups. We call the algorithms used to distribute a generic quantity Q into m groups a *distribution algorithms*. Their output are the amounts $\boldsymbol{q} = (q_1, q_2, \ldots, q_m)$ out of Q assigned to each group. We will now go over the two algorithms used in our experiments. In their description, we will call the quantity to be distributed (space or slack) Q.

2.3.3.1 Uniform Distribution

Algorithm `DistributeUniformly`(Q) is a distribution algorithm that assigns Q as evenly as possible to each of the m groups, such that $|q_g - q_h| \leq 1$ for any two groups $g, h \in [m]$.

2.3.3.2 Geometric Distribution

Algorithm `DistributeGeometrically`(Q) distributes Q such that q_g is proportional to $p(1-p)^{g-1}$ for each group $g \in [m]$, where $0 < p \leq 1$ is a parameter that controls how even the distribution is. For example, suppose we have $Q = 100$ and $m = 5$. If $p = 0.05$, then $\boldsymbol{q} = (22, 21, 20, 19, 18)$, a very even distribution. However if $p = 0.9$, then $\boldsymbol{q} = (90, 9, 1, 0, 0)$, a very uneven distribution.

3 EXPERIMENTS

In order to test our hypothesis about what features of the instance are important to predict its hardness for IMS, we conducted a series of experiments. All experiments follow the same structure. We generate instances using the method described in Section 2.3. We keep all but one parameter fixed, the parameter we are interested to see the effect on hardness. We then run solution stabilization – described in Section 2.1 – a total of 15 times for each instance to obtain a BKV and each run's ITB, our metric for hardness. The only exception is the next experiment, for which we have data of 30 solution stabilization runs per instance left from tests to see if ITB statistics would improve, however it was not worth the cost for all experiments. We then compare runs of different instances to see if the feature has any effect on ITB. Since we use stochastic methods to generate the instances, we replicate the experiment for a few sets of instances with different values for m or n to see if the effects are consistent.

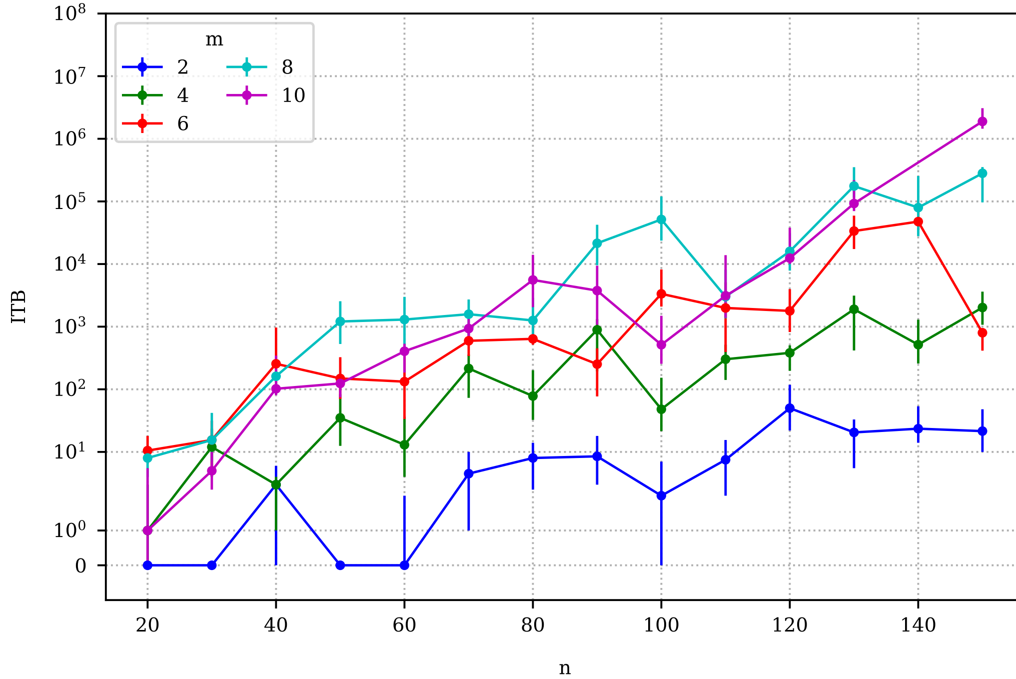
3.1 Influence of the Number of Nodes

To illustrate the process followed in all experiments, let us take the simplest one as an example. It tests the hypothesis that a larger number of nodes makes it harder for IMS to find a good solution. We generate 70 instances split into 5 different sets with different values of m, keeping all parameters fixed, but varying n between instances with same m. The results are presented in Figure 3.1. The plot shows the median ITB of 30 runs of solution stabilization for a single instance, and error bars indicate the first and third quartiles. Notice the y-axis is in what is called a "symmetric logarithmic" scale, which allows us to plot zeroes by taking a linear scale in a small range around zero (Matplotlib 2020).

As we can see, as n increases, there is a tendency for the ITB to increase too. This can serve as a sanity check for our methodology, since this behavior is expected. However, we can see it is not always the case that the ITB increases with an increment in n. For example, when $m = 6$ and $n = 150$, we recorded lower ITBs than for $n = 140$. This may be due to the stochastic nature of our instance generation process, as we will see in Section 3.3.

As mentioned in Section 2.1, not necessarily all runs of solution stabilization will converge to the BKV. Therefore, when presenting the experiments, we only show the

Figure 3.1: ITB vs. n. Each point on the plot represents 15 runs of solution stabilization, median ITB defines the point's center, while the first and third quartiles define the error bars.



ITB of solution stabilization runs that reached the BKV. Those runs that did not reach the BKV are described in a summary, as shown in Table 3.1 for this experiment. The first column of the table identifies the set of instances. The second column shows how many of the solution stabilization runs for those instances reached the BKV. The third and fourth columns show the average $\overline{\Delta}$ and standard deviation σ_{Δ} of the percent distance between the BKV and the quality of the solution found by those runs that did not reach the BKV. In this specific case, for $m = 2$, only one solution stabilization run did not converge (99.76% did), hence σ_{Δ} is not shown.

To generate the instances, we use the parameters shown in Table 3.2. Values that do not apply are marked with a bar (–), such as arguments for algorithms that take none. The slack is arbitrarily set to 0. We've also chosen the simplest distribution algorithm `DistributeUniformly` – described in Section 2.3.3.1 – for distributing space among groups. The different values of n were chosen by some trial and error to be in a range where ITB varies a fair amount (around 6 orders of magnitude in this case). Replicating the experiment with different values of m lets us see if behaviors are consistent for instances with a different number of groups. In all experiments we chose m and n in a range that produces a significant amount of iterations for IMS to avoid making all instances easy. However, since experiments take many runs of IMS, m and n must not be

too large, so that we can compute all those runs in feasible time. We must also take into account the trial and error nature of choosing ranges for parameters in order so see any significant pattern of change in ITB. For similar reasons, solution stabilization was run 15 times for each instance in other experiments, instead of 30 we used in this one.

Table 3.1: Summary of solution stabilization runs that did not reach the BKV for the experiment on number of nodes.

m	Reached BKV (%)	$\overline{\Delta}$ (%)	σ_Δ (%)
2	99.76	0.01	–
4	95.00	0.02	0.01
6	88.81	0.08	0.06
8	81.67	0.10	0.09
10	85.13	0.14	0.12

Table 3.2: Parameters for the experiment on the number of nodes.

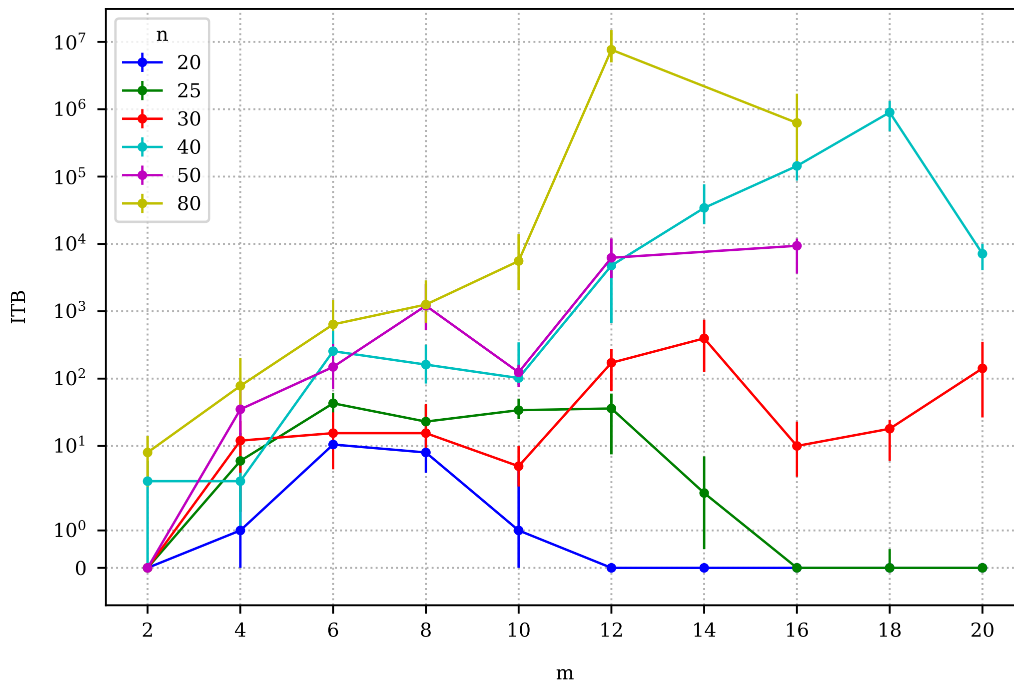
Parameter	Value
n	$10 + 10 \times [14]$
m	$2 \times [5]$
Total slack up (U)	0
Total slack down (L)	0
Space distribution algorithm	`DistributeUniformly`
Space distribution parameters	–
U distribution algorithm	`DistributeUniformly`
U distribution parameters	–
L distribution algorithm	`DistributeUniformly`
L distribution parameters	–
Distance generation algorithm	`GenerateRandomIntegerDistances`
Distance generation parameters	$v_{min} = 0$, $v_{max} = 10$
Random seed	0

3.2 Influence of the Number of Groups

Just like the number of nodes, we want to see the influence of the number of groups on ITB. Figure 3.2 shows the median ITB as a function of m for a series of instances with different values of n. We can see that there is a tendency for ITB to increase as the number of groups becomes larger. There is also a significant amount of variation in the behavior for different values of n. For example, for the set of instances with $n = 30$, ITB is in the range of hundreds of iterations at $m = 14$, but only tens at $m = 16$. This variation is present for the other values of m larger than 30. We study this behavior more closely in Section 3.3.

For the smaller values $n = 20$ and $n = 25$, ITB decreases in the higher range of values of m until it reaches practically zero iterations. This is likely due to the fact that, since we have few nodes per group, and a total slack of zero, instances become over-constrained and the problem becomes easier, as observed in other optimization problems (Cheeseman, Kanefsky, and Taylor 1991)

Figure 3.2: Runs that converged to the BKV.



Finally, Table 3.3 shows a summary of the solution stabilization runs for this experiment. Most of the runs converge to the BKV. For $n = 80$, a more significant portion of the runs didn't converge to the BKV, which can be attributed to the size of the problem. The parameters used for this experiment are shown in Table 3.4.

Table 3.3: Summary of solution stabilization runs that did not reach the BKV for the experiment on the number of groups.

n	Reached BKV (%)	$\overline{\Delta}$ (%)	σ_Δ (%)
20	100.00	–	–
25	100.00	–	–
30	99.05	0.09	0.04
40	90.95	0.14	0.14
50	89.09	0.24	0.17
80	78.79	0.25	0.32

Table 3.4: Parameters for the experiment on the number of groups.

Parameter	Value
n	20, 25, 30, 40, 50, 80
m	$2 + 2 \times [11]$
Total slack up (U)	0
Total slack down (L)	0
Space distribution algorithm	`DistributeUniformly`
Space distribution parameters	–
U distribution algorithm	`DistributeUniformly`
U distribution parameters	–
L distribution algorithm	`DistributeUniformly`
L distribution parameters	–
Distance generation algorithm	`GenerateRandomIntegerDistances`
Distance generation parameters	$v_{min} = 0, v_{max} = 10$
Random seed	0

3.3 Influence of the Randomness in the Instance Generation Procedure

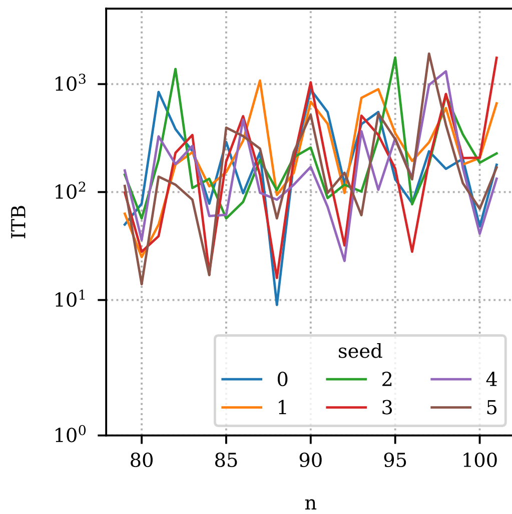
Since instances are generated using random numbers – as described in Section 2.3 – there is variation between instances generated from the same parameters but with a different random seed. We would like to know the impact of this randomness on the ITB. Take for example the instance in Figure 3.1 with $m = 4$ and $n = 90$. This instance's ITB is

an order of magnitude larger than the ITB of the instance with $m = 4$ and $n = 100$, even though the parameter is smaller in the first instance n.

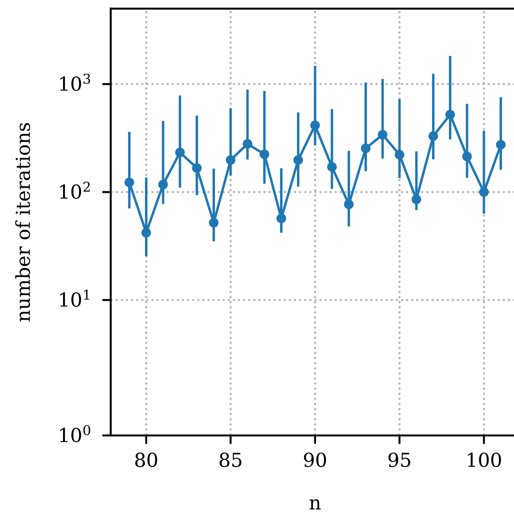
To assess how much of this variation can be attributed to randomness, we generated instances in the region of $n \in [79, 101]$ using 6 different random seeds and applied the same experimental procedure described in Section 3.1. Results are shown in Figure 3.3a. Error bars were omitted for legibility, but they are roughly the same size as those in Figure 3.1. The variation is substantial even among instances with equal number of nodes, which explains some of the noise in the experiments. Instances also appear to be easier around values of n which are multiples of m. Figure 3.3b shows the solution stabilization runs for all seeds combined. It is clear from the combined data that the median ITB tends to increase with n. The low ITB around values of n divisible by m is also clear. We can observe a similar behavior repeating the experiment for instances with $m = 5$, as shown in Figures 3.3c and 3.3d. The results are analogous to those with $m = 4$, presenting the same behavior around $n \mod m = 0$ and – although not as clear and pronounced in this short range – a slight tendency for the ITB to be higher for larger values of n.

All experiments in the following sessions run solution stabilization 15 times for each instance generated from a single set of parameters. Ideally, this process should be repeated for instances with different random seeds and the results combined like it was done in this experiment. However, since 15 runs of solution stabilization runs can take more than a day to compute for larger instances, replicating them for many seeds was not possible due to time constraints.

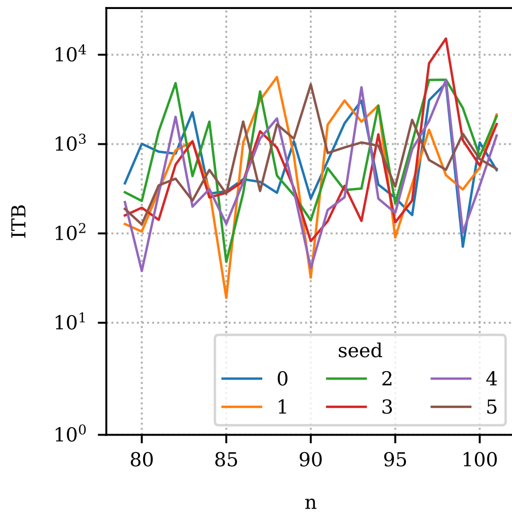
Figure 3.3: Effect of the randomness in the instance generation procedure on the ITB for instances built with different random seeds.



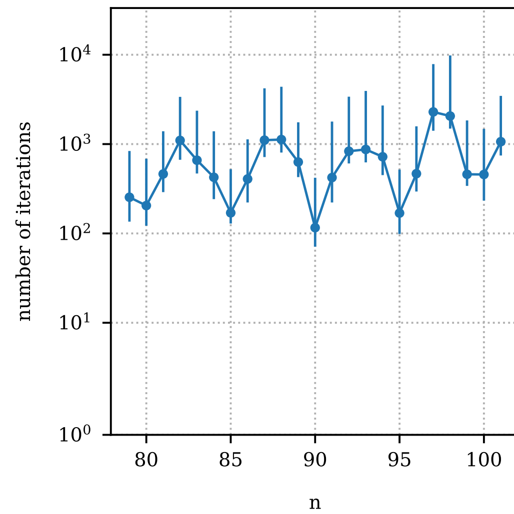
(a) Solution stabilization results for instances of each seed ($m = 4$).



(b) Combined solution stabilization results for all seeds ($m = 4$).



(c) Solution stabilization results for instances of each seed ($m = 5$).



(d) Combined solution stabilization results for all seeds ($m = 5$).

3.4 Influence of the Slack

A positive slack in an instance increases the number of possible solutions compared to an instance that's identical except with a slack of zero. Slack allows a group to contain more or less than a certain amount of nodes. An instance with slack zero would only allow groups to have a fixed number of nodes. To test the influence of the slack parameter, we generated instances varying the size of the slack in the range $[0, 20]$ for instances with $n = 100$. In terms of the instance generation parameters of Section 2.3, we set slack up U and slack down D to values in $[0, 10]$ and distribute those quantities using the distribution algorithm `DistributeUniformly` (Section 2.3.3.1) for simplicity.

The results are shown in Figure 3.4. As we can see, the influence of slack is not clear cut. For $m = 8$, we can see a small tendency for the ITB to decrease, although we have many instances that did not converge to the BKV, as shown in Table 3.6. There are two factors that could contribute to this low rate of convergence to the BKV. The first is the value of m, which makes the problem harder as seen by the influence of the number of groups in Section 3.2. Adding slack to groups increases the number of possible solutions, therefore it makes it more likely that IMS will settle on a local maximum.

Parameters used to generate the instances for this experiment are shown in Table 3.5. Other experiments set slack arbitrarily to 0, since it appears to have no major effect on ITB.

Figure 3.4: Runs that converged to the BKV.

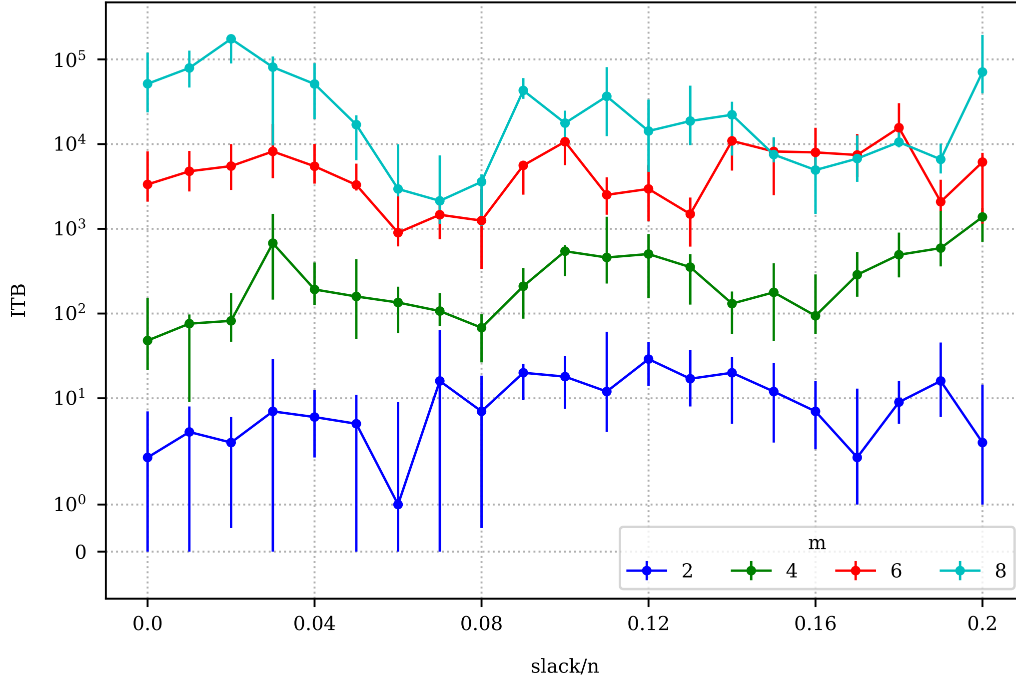


Table 3.5: Parameters for the experiment on slack.

Parameter	Value
n	100
m	2, 4, 6, 8
Total slack up (U)	$[0, 10]$
Total slack down (L)	$[0, 10]$
Space distribution algorithm	`DistributeUniformly`
Space distribution parameters	–
U distribution algorithm	`DistributeUniformly`
U distribution parameters	–
L distribution algorithm	`DistributeUniformly`
L distribution parameters	–
Distance generation algorithm	`GenerateRandomIntegerDistances`
Distance generation parameters	$v_{min} = 0, v_{max} = 10$
Random seed	0

Table 3.6: Summary of solution stabilization runs that did not reach the BKV for the experiment on slack.

m	Reached BKV (%)	$\overline{\Delta}$ (%)	σ_Δ (%)
2	100.00	–	–
4	96.83	0.04	0.03
6	75.87	0.08	0.06
8	73.02	0.14	0.12

3.5 Influence of the Difference in Group Sizes

This experiment tests the effect of the difference in size between groups. We use the algorithm `DistributeGeometrically` – described in Section 2.3.3.2 – to distribute space into the groups, varying the parameter p to obtain instances with many degrees of discrepancy between group spaces. Results are presented in Figure 3.5a. Remember that a value of p close to 1 generates instances with a very uneven distribution of space among groups, while a value of p close to 0 yields a more even distribution. All instances present similar behavior. In the range of p close to zero, we see a tendency for the ITB to increase. ITB starts to decrease after some point and quickly drops to zero when p is close to 0.8 (i.e., when groups sizes are very uneven).

Given the distribution of space (s_g), with a slack of zero, the number of possible solutions P is:

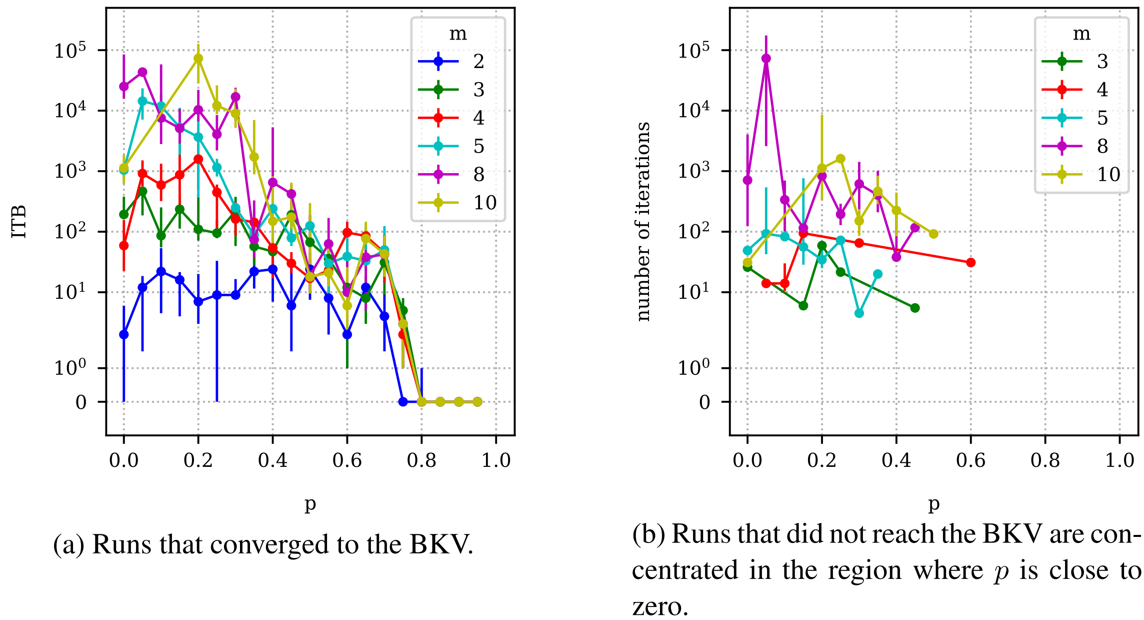
$$P = \frac{n!}{\prod_{g \in [m]} s_g!}$$

Figure 3.6 shows P versus p in logarithmic scale for the instances of $m = 5$ used in this experiment. As we can see, the number of possible solutions quickly drops, which could explain the rapid drop to zero of the ITB when p is very large. In that region, the problem becomes overconstrained, since we have less possibilities when there are extremely small groups. The summary in Table 3.7 tells us almost all solution stabilization runs converged to the BKV. Runs that did not reach the BKV are concentrated in the region of small values of p, as shown in Figure 3.5b. This can also be explained by the high number of possible solutions in that region. In Table 3.8 are the parameters used to generate the instances for this experiment. Slack is set to zero since, as we've seen, it doesn't seem to influence ITB vey much.

Table 3.7: Summary of runs that did not reach the BKV for the experiment on difference in group sizes.

m	% converged to BKV	mean Δ	σ_Δ
2	100.00	–	–
3	97.67	0.03	0.02
4	97.33	0.06	0.05
5	93.67	0.11	0.09
8	86.67	0.15	0.15
10	89.80	0.12	0.09

Figure 3.5: ITB vs. group size discrepancy



(a) Runs that converged to the BKV.

(b) Runs that did not reach the BKV are concentrated in the region where p is close to zero.

.

Figure 3.6: The general shape of the number of possible solutions vs. the parameter p of the algorithm `DistributeGeometrically` (Section 2.3.3.2). The number decreases quickly as p approaches 1. This plot specifically takes the group sizes of the instances with $m = 5$ of this experiment.

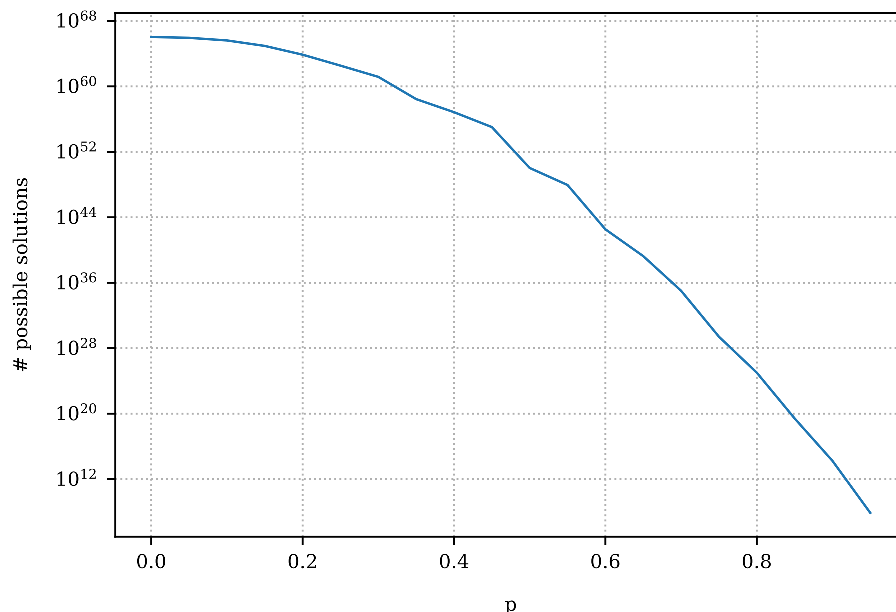


Table 3.8: Parameters for the experiment on the difference in group sizes.

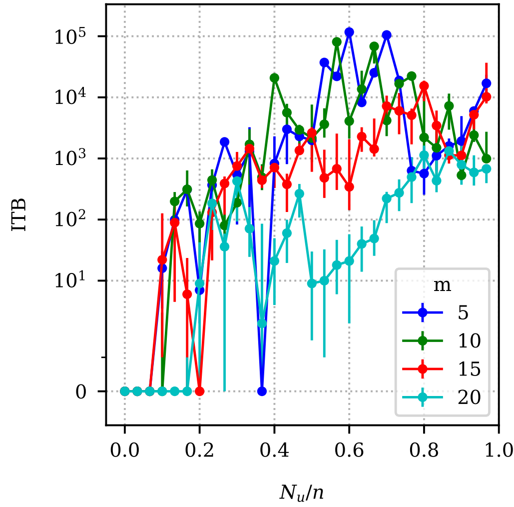
Parameter	Value
n	100
m	2, 3, 4, 5, 8, 10
Total slack up (U)	0
Total slack down (L)	0
Space distribution algorithm	`DistributeGeometrically`
Space distribution parameters	$p \in 10^{-6} + 0.5 \times [0, 19]$
U distribution algorithm	`DistributeUniformly`
U distribution parameters	–
L distribution algorithm	`DistributeUniformly`
L distribution parameters	–
Distance generation algorithm	`GenerateRandomIntegerDistances`
Distance generation parameters	$v_{min} = 0, v_{max} = 10$
Random seed	0

3.6 Influence of the Number of Unitary Edges Forming a Tree

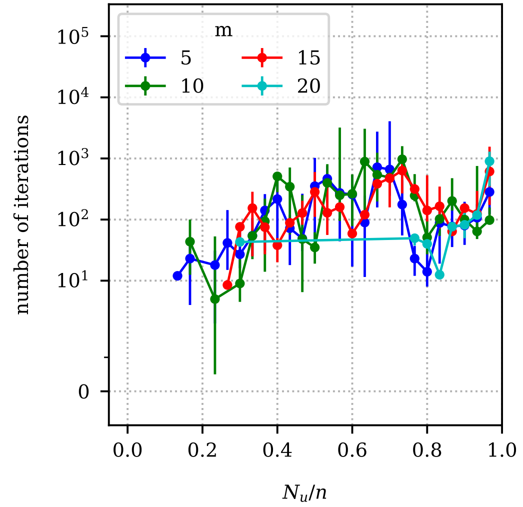
For this experiment, we use `GenerateDistancesWithUnitaryTree` (Algorithm 3) to generate the distances in a way such that a group of nodes is connected by edges of weight 1 forming a tree, while all other edges have weight 0. The principle behind this method of generating instances is that if two nodes have a distance of 0 between them, the value of the solution is not affected by them being on the same group or not. However, if two nodes have a positive distance between them, it matters for the solution quality that they are in the same group. We want to see the effect of making nodes "dependent" of each other in this sense by seeing the influence of the number N_u of edges of weight one in the graph.

Figure 3.7a shows the ITB as a function of N_u, the number of edges of weight 1, shown as a ratio of n. No nodes are connected by unitary edges when $N_u = 0$. ITB generally increases as number of unitary edges increases. When $N_u = 0$ (i.e., all distances are zero), the ITB is practically zero. Since we generate instances starting from the same random seed, the unitary edges in an instance with, say, $N_u = k$ are a subset of those in instances with $N_u > k$. This makes the instances more comparable, since they only differ on the extra or missing unitary edges. Even with that, we see significant variation among similar instances. Parameters used to generate instances for this are shown in Table 3.9.

Figure 3.7: ITB vs. number of edges in unitary tree



(a) Runs that converged to the BKV.



(b) Runs that did not reach the BKV are concentrated in the region where N is close to 1.

.

Table 3.9: Parameters for the experiment on number of unitary edges forming a tree.

Parameter	Value
Experiment	ExperimentNumEdgesInUnitaryTree
n	100
m	5, 10, 15, 20
Total slack up (U)	0
Total slack down (L)	0
Space distribution algorithm	`DistributeUniformly`
Space distribution parameters	–
U distribution algorithm	`DistributeUniformly`
U distribution parameters	–
L distribution algorithm	`DistributeUniformly`
L distribution parameters	–
Distance generation algorithm	`GenerateDistancesWithUnitaryTree`
Distance generation parameters	$N \in [0, 100]$
Random seed	0

Many instances did not converge to the BKV for this experiment, as can be seen in Table 3.10. The median number of iterations for those runs is shown in Figure 3.7b.

Table 3.10: Summary of the runs of solution stabilization that did not reach the BKV for the experiment on the number of unitary edges forming a tree.

m	Reached BKV (%)	$\overline{\Delta}$ (%)	σ_Δ (%)
5	46.06	3.34	1.60
10	60.61	2.61	1.07
15	56.36	3.12	1.49
20	60.61	2.84	0.75

In most cases, this number is roughly an order of magnitude smaller for runs that did not reach the BKV than for the ones that did. This is a reflex of our stopping criteria for IMS, which stops the search after no improvement has been made for more than 93% $(15/16)$ of the iterations (Section 2.1). This means IMS found a solution with quality less than the BKV, but was not able to make further improvements to the solution before reaching the stopping criteria. The high number of runs not reaching the BKV, and high variance of the quality among those runs, indicate the algorithm found many good local maxima solutions for those instances. This points to the possibility that this procedure generates instances that are hard for IMS.

3.7 Influence of the Standard Deviation of Distances

We now look at the effect of the distribution of the values for distances. The relevant parameter in our instance generation procedure is the distance generation algorithm. We use the algorithm `GenerateNormallyDistributedDistances` (Section 2.3.2.2), to distribute distances according to a normal distribution. We set the mean of the distribution μ to 100 and vary the standard deviation σ of the distribution. We normalize the distances according to the results of Section 1.3.2. We can see ITB as a function of the standard deviation of distances in Figure 3.8. The standard deviation has minimal influence on ITB, since almost all points are within each others' error bars. We again see that larger values of m make it more difficult for IMS to converge to the BKV, as shown in Table 3.11. The parameters for these instances is shown in Table 3.12. Parameters used for this experiment are listed in Table 3.12.

Table 3.11: Summary of solution stabilization runs that did not reach the BKV for the experiment on standard deviation of distances.

m	% converged to BKV	mean Δ	σ_{Δ}
2	100.00	–	–
4	99.05	–	–
5	65.71	0.00	0.01
8	65.71	0.04	0.05
10	61.90	0.04	0.06

Figure 3.8: Runs that converged to the BKV.

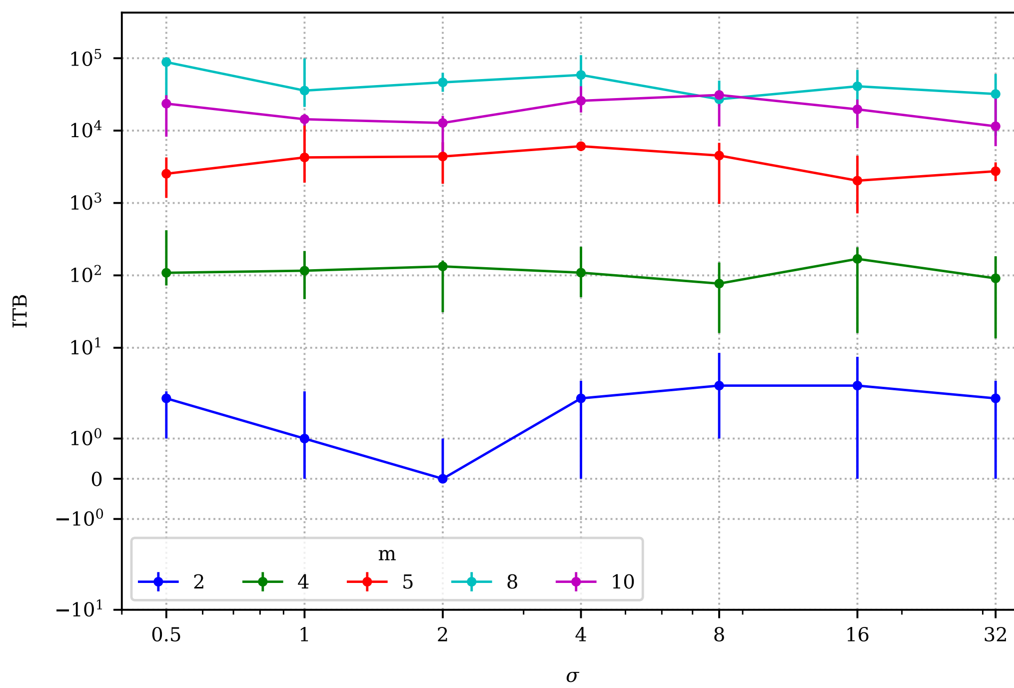


Table 3.12: Parameters for the experiment on standard deviation of distances.

Parameter	Value
n	100
m	2, 4, 5, 8, 10
Total slack up (U)	0
Total slack down (L)	0
Space distribution algorithm	`DistributeUniformly`
Space distribution parameters	–
U distribution algorithm	`DistributeUniformly`
U distribution parameters	–
L distribution algorithm	`DistributeUniformly`
L distribution parameters	–
Distance generation algorithm	`GenerateNormallyDistributedDistances`
Distance generation parameters	$\mu = 100$, $\sigma \in \{0.5, 1, 2, 4, 8, 16, 32\}$
Random seed	0

3.8 Influence of the Diversity of Distance Values

In other experiments we came to the conclusion that MDGP instances that have all distances equal are easy for IMS to solve. Those same instances can be solved in polynomial time using the algorithm proposed in Section 1.3.3. In this experiment, we want to see how the problem becomes harder for larger sets of possible values for distances. If instances are easy when all distances have the same value, then we want to see if having two (three, four, etc.) possible values for distances makes the problem harder. We generate instances with a fixed set of possible values for distances between nodes using the algorithm `GenerateDistancesLimitedToValues` (Section 2.3.2.3), varying the parameter c – the number of possible values for distances, or distance *diversity*. For example, if $c = 3$, distances between nodes are assigned values of 1, 2 and 3 in the same proportion (or as close as possible). Here we also normalize the distances according to the results of Section 1.3.2.

The solution stabilization runs for this experiment can be seen in Figure 3.9. When distances are all equal, we have ITB close to zero as we've seen before in other experiments. If we allow distances to have two possible values instead of one ($c = 2$), instances have a sharp rise on hardness. We attribute this to the fact that the problem is solvable in polynomial-time when $c = 1$, as seen in Section 1.3.3, but in the general case it is NP-complete. Adding more possible values ($c > 2$) does not produce any significant effect on hardness. Table 3.14 shows the parameters used in the instances for this experiment.

Table 3.13: Summary of solution stabilization runs that did not reach the BKV for the experiment on diversity of distance values.

m	Reached BKV (%)	$\overline{\Delta}$ (%)	σ_{Δ} (%)
2	100.00	–	–
4	99.17	0.01	–
5	92.50	0.07	0.05
8	66.67	0.17	0.15

Figure 3.9: Runs that converged to the BKV.

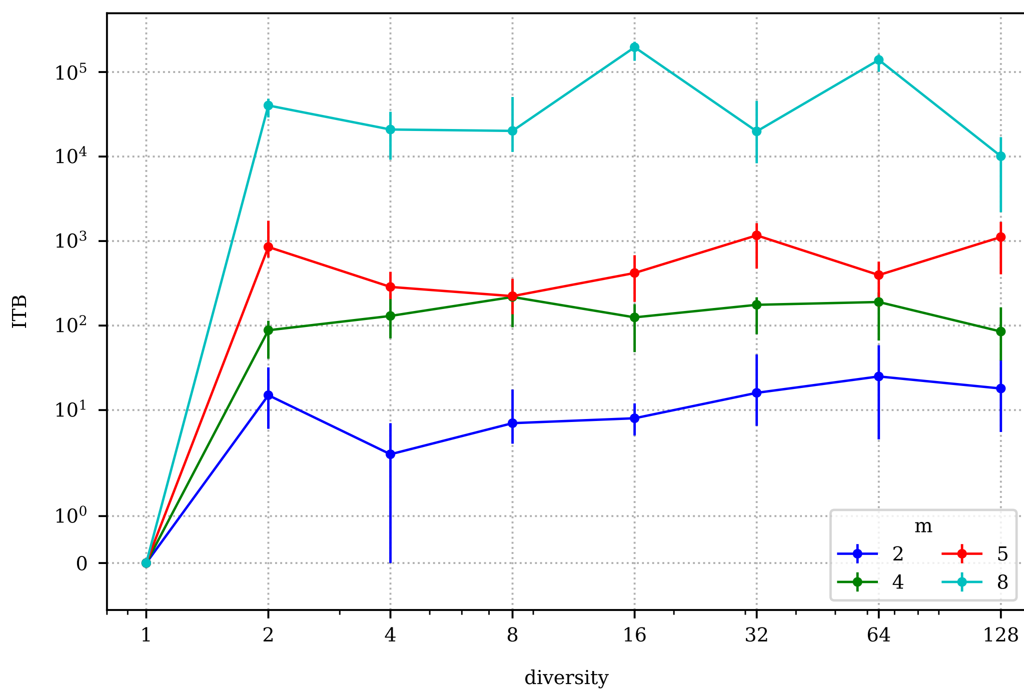


Table 3.14: Parameters for the experiment on diversity of distance values.

Parameter	Value
n	100
m	2, 4, 5, 8
Total slack up (U)	0
Total slack down (L)	0
Space distribution algorithm	`DistributeUniformly`
Space distribution parameters	–
U distribution algorithm	`DistributeUniformly`
U distribution parameters	–
L distribution algorithm	`DistributeUniformly`
L distribution parameters	–
Distance generation algorithm	`GenerateDiverseDistances`
Distance generation parameters	$c \in \{1, 2, 4, 8, 16, 32, 64, 128\}$
Random seed	0

4 CONCLUSIONS AND PERSPECTIVES FOR THE FUTURE

In this research, we have created a methodology to investigate the influence of features of instances of the MDGP on the solving time of the IMS heuristic. Our methodology involved the solution stabilization procedure to measure hardness, an instance generation procedure to control the features of the instances, and experiments combining those two procedures. We were able to confirm our intuitions that hardness is proportional to the number of groups and number of nodes. We saw the size of connected group in Section 3.9 is a parameter that produces interesting behavior. Differences in group size also have some influence over ITB. Other features are not clearly tied to, or have no influence over, the hardness of the instance.

A point of improvement upon this work would be to investigate more clearly the source for variations in ITB for even similar instances, as mentioned in Section 3.3. We could run the experiments with more than one instance with the same parameters, but different random seeds and observe the behavior. More fundamentally, reducing the influence of randomness in the instance generation procedure described in Section 2.3 could be another approach to sharpen the results of these experiments. One approach could be to generate one instance and use it as a basis for the construction of others in the same experiment, instead of generating entirely new instances from a set of similar parameters. This could help reduce the effect of the random generation of distances, and distribution of space and slack.

An explanation of why ITB presents a periodic behavior sometimes – as seen in Section 3.3 – is still lacking. It is possible that this is tied to some peculiarity of IMS, such as the way nodes are exchanged between groups during the search. A replication of this experiment with other heuristics to see if the behavior persists could bring some light into this matter.

More simulation time can be spent on experiments with many instances not converging to the BKV, such as the one presented in Section 3.6. The stopping criteria for IMS could be altered to allow more runs to converge to the BKV and have a more accurate estimate of the ITB.

Future work could rely on this methodology and apply it to heuristics other than IMS. It would be interesting to compare two different heuristics to map on which instances each heuristic works better. The instance generation procedure gives us an easy way to control the parameters of instances. It is also possible that analogous methodologies can

be created for optimization problems other than MDGP.

BIBLIOGRAPHY

Achlioptas, Dimitris, Assaf Naor, and Yuval Peres (2005). “Rigorous location of phase transitions in hard optimization problems”. In: *Nature* 435.7043, pp. 759–764. ISSN: 1476-4687. DOI: <10.1038/nature03602>.

Arora, Sanjeev and Boaz Barak (2009). *Computational complexity: a modern approach*. Cambridge University Press. ISBN: 978-0-521-42426-4.

Balasubramanian, R., M. R. Fellows, and Venkatesh Raman (1998). “An improved fixed-parameter algorithm for vertex cover”. In: *Information Processing Letters* 65.3, pp. 163–168. ISSN: 00200190. DOI: <10.1016/S0020-0190(97)00213-5>.

Berger, Robert (1966). “The undecidability of the domino problem”. In: *Memoirs of the American Mathematical Society* 0.66, 0–0. ISSN: 0065-9266, 1947-6221. DOI: <10.1090/memo/0066>.

Cassaigne, Julien et al. (2014). “Tighter Undecidability Bounds for Matrix Mortality, Zero-in-the-Corner Problems, and More”. In: *arXiv:1404.0644 [cs, math]*. arXiv: 1404.0644. URL: <http://arxiv.org/abs/1404.0644>.

Cheeseman, Peter, Bob Kanefsky, and William M. Taylor (1991). “Where the really hard problems are”. In: Morgan Kaufmann, pp. 331–337.

Chen, Yuan et al. (2011). “A hybrid grouping genetic algorithm for reviewer group construction problem”. In: *Expert Systems with Applications* 38.3, pp. 2401–2411. ISSN: 0957-4174. DOI: <10.1016/j.eswa.2010.08.029>.

Coja-Oghlan, Amin, Michael Krivelevich, and Dan Vilenchik (2010). “Why Almost All k-Colorable Graphs Are Easy to Color”. In: *Theory of Computing Systems* 46.3, pp. 523–565. ISSN: 1433-0490. DOI: <10.1007/s00224-009-9231-5>.

Cook, Stephen A. (1971). “The Complexity of Theorem-proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. event-place: Shaker Heights, Ohio, USA. ACM, pp. 151–158. DOI: <10.1145/800157.805047>. URL: <http://doi.acm.org/10.1145/800157.805047>.

Cormen, Thomas H et al. (2009). *Introduction to algorithms*. 3rd ed. MIT Press. ISBN: 978-0-262-03384-8.

Downey, R. G. and M. R. Fellows (1999). *Parameterized complexity: with 68 figures*. Softcover reprint of the hardcover 1st ed. 1999. Monographs in computer science. Springer. ISBN: 978-1-4612-6798-0.

Edmonds, Jack (1965). “Paths, Trees, and Flowers”. In: *Canadian Journal of Mathematics* 17, pp. 449–467. ISSN: 0008-414X, 1496-4279. DOI: <10.4153/CJM-1965-045-4>.

Feo, Thomas A. and Mallek Khellaf (1990). “A class of bounded approximation algorithms for graph partitioning”. In: *Networks* 20.2, pp. 181–195. DOI: <10.1002/net.3230200205>.

Gomes, Carla P. and Bart Selman (2001). “Algorithm portfolios”. In: *Artificial Intelligence*. Tradeoffs under Bounded Resources 126.1, pp. 43–62. ISSN: 0004-3702. DOI: <10.1016/S0004-3702(00)00081-3>.

Hartmanis, J. and R. E. Stearns (1965). “On the computational complexity of algorithms”. In: *Transactions of the American Mathematical Society* 117, pp. 285–306. ISSN: 0002-9947, 1088-6850. DOI: <10.1090/S0002-9947-1965-0170805-7>.

Kanda, Jorge et al. (2011). “Selection of algorithms to solve traveling salesman problems using meta-learning”. In: *Int. J. Hybrid Intell. Syst.* 8, pp. 117–128. DOI: <10.3233/HIS-2011-0133>.

Karp, Richard M. (1972). “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Springer US, pp. 85–103. ISBN: 978-1-4684-2003-6. DOI: <10.1007/978-1-4684-2001-2_9>. URL: <http://link.springer.com/10.1007/978-1-4684-2001-2_9>.

Kromer, P., J. Platos, and M. Kudelka (2017). “Network measures and evaluation of traveling salesman instance hardness”. In: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–7. DOI: <10.1109/SSCI.2017.8285352>.

Lai, Xiangjing and Jin-Kao Hao (2016). “Iterated maxima search for the maximally diverse grouping problem”. In: *European Journal of Operational Research* 254.3, pp. 780–800. ISSN: 0377-2217. DOI: <10.1016/j.ejor.2016.05.018>.

Laporte, Gilbert (1992). “The vehicle routing problem: An overview of exact and approximate algorithms”. In: *European Journal of Operational Research* 59.3, pp. 345–358. ISSN: 0377-2217. DOI: <10.1016/0377-2217(92)90192-C>.

Matplotlib (2020). *Matplotlib 3.1.0 documentation*. Accessed: 2020-01-19. URL: <%5Curl%7Bhttps://matplotlib.org/3.1.0/api/scale_api.html#matplotlib.scale.SymmetricalLogScale%7D>.

Mitchell, David, Bart Selman, and Hector Levesque (1992). “Hard and Easy Distributions of SAT Problems”. In:

Pisinger, David (2005). “Where are the hard knapsack problems?” In: *Computers Operations Research* 32.9, pp. 2271–2284. ISSN: 0305-0548. DOI: <10.1016/j.cor.2004.03.002>.

Poonen, Bjorn (2012). *Undecidable Problems: A Sampler*.

Prosser, Patrick (1996). “An empirical study of phase transitions in binary constraint satisfaction problems”. In: *Artificial Intelligence*. Frontiers in Problem Solving: Phase Transitions and Complexity 81.1, pp. 81–109. ISSN: 0004-3702. DOI: <10.1016/0004-3702(95)00048-8>.

S. Paterson, Michael (1970). *Unsolvability in 33 matrices*. URL: <https://www.researchgate.net/publication/268549900_Unsolvability_in_33_matrices>.

Sipser, Michael et al. (2006). *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston.

Smith-Miles, Kate, Jano van Hemert, and Xin Yu Lim (2010). “Understanding TSP Difficulty by Learning from Evolved Instances”. In: *Learning and Intelligent Optimization*. Ed. by Christian Blum and RobertoEditors Battiti. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 266–280. ISBN: 978-3-642-13800-3.

Turing, A. M. (1937). “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1, pp. 230–265. ISSN: 1460-244X. DOI: <10.1112/plms/s2-42.1.230>.

Turner, Jonathan S (1988). “Almost all k-colorable graphs are easy to color”. In: *Journal of Algorithms* 9.1, pp. 63–82. ISSN: 0196-6774. DOI: <10.1016/0196-6774(88)90005-3>.

Vizel, Yakir, Georg Weissenbacher, and Sharad Malik (2015). “Boolean Satisfiability Solvers and Their Applications in Model Checking”. In: *Proceedings of the IEEE* 103.11, pp. 2021–2035. ISSN: 1558-2256. DOI: <10.1109/JPROC.2015.2455034>.

Weitz, R. R. and S. Lakshminarayanan (1997). “An empirical comparison of heuristic and graph theoretic methods for creating maximally diverse groups, VLSI design, and exam scheduling”. In: *Omega* 25.4, pp. 473–482. ISSN: 0305-0483. DOI: <10.1016/S0305-0483(97)00007-8>.

Xu, L. et al. (2008). “SATzilla: Portfolio-based Algorithm Selection for SAT”. In: *Journal of Artificial Intelligence Research* 32, pp. 565–606. ISSN: 1076-9757. DOI: <10.1613/jair.2490>.