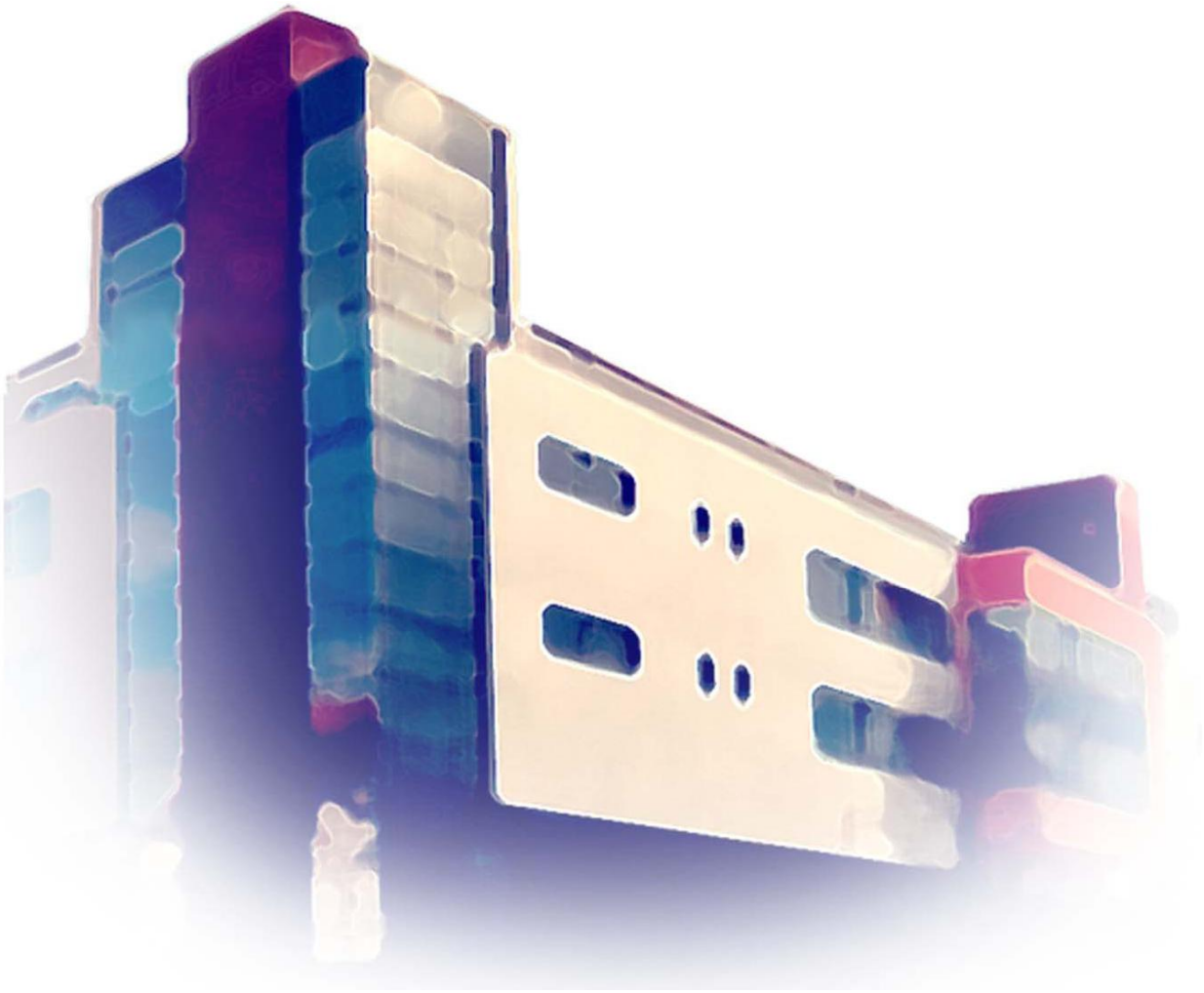
Arne Recknagel

An Approach to Efficiently Calculating Dodgson-Scores Using Heuristics and Parallel Computing



PICS

Publications of the Institute of Cognitive Science

Volume 1-2015

ISSN: 1610-5389

Series title: PICS
Publications of the Institute of Cognitive Science

Volume: 1-2015

Place of publication: Osnabrück, Germany

Date: March 2015

Editors: Kai-Uwe Kühnberger
Peter König
Sven Walter

Cover design: Thorsten Hinrichs

Universität Osnabrück
Department of Artificial Intelligence

Bachelor Thesis

in Cognitive Science

Topic: An Approach to Efficiently Calculating Dodgson-Scores Using Heuristics and Parallel Computing

Author: Arne Recknagel

Date: 2 October 2014

First Supervisor: Prof. Dr. phil. Kai-Uwe Kühnberger

Second Supervisor: Dipl.-Math. Tarek Richard Besold

Abstract

The objective of this thesis will be to measure the practical limits of algorithms computing the exact Dodgson scores from a number of votes. While the problem itself is theoretically intractable, this work will feature five different solution which try different approaches to solve it in an effective manner. Additionally, three of them can be run in parallel which has the potential of drastically reducing the problem size.

Contents

1 Introduction **3**
- 1.1 Objectives and Structure . . . 3
- 1.2 The History of Voting . . . 3
- 1.3 Technical Terms and Definitions . . . 5
- 1.4 The Frame of Dodgson's Rule . . . 6

2 Finding a Baseline **10**
- 2.1 The Baseline-Scorer . . . 10
- 2.2 Complexity . . . 11

3 Depth-First Search **14**
- 3.1 The Depth-First Approach . . . 14
- 3.2 Reason for the Speedboost . . . 15

4 Uniform-Cost Search **16**
- 4.1 The Uniform-Cost Approach . . . 16
- 4.2 Complexity . . . 17
- 4.3 Worst Case . . . 18
- 4.4 Best Case . . . 19

5 Smart Cache **22**
- 5.1 The Smart Caching Approach . . . 22

6 Iterative Cost Raise **24**
- 6.1 Improvements and Payoff . . . 24

7 Multi-Processing **26**
- 7.1 Threading . . . 26

8 Benchmarking **29**
- 8.1 Randomness . . . 29
- 8.2 Average Performance . . . 30
- 8.3 Maximum Range . . . 30
- 8.4 Analysis of the Search Space . . . 33

9 Result **38**
9.1 Sequential Algorithm Results 38
9.2 Multi-processing Algorithm Results 39

10 Conclusion **41**
10.1 Future Work 41

11 Usage **43**
11.1 Extension 43
11.2 Implementation of a New Scorer 44
11.3 Useful Functions and Structures 46

12 Appendix **47**
12.1 Tables 47
12.2 Uniform Cost Search 50

1. Introduction

1.1 Objectives and Structure

Conflict of interest is the permanent companion of any population. For that reason, the ability to compromise is of paramount importance, making voting as old as civilization. In the wake of the French revolution and the end of the monarchy, voting theory roused a new wave of interest in Europe, also sparking a still ongoing debate (Risse, 2005). This chapter will give an introduction to the matter of voting rules and some of their technical peculiarities, together with the idea, research question, justification, and point of this thesis.

A wrap-up and presentation of the work done will be given in chapters 2 (Finding a Baseline), 3 (Depth-First Search), 4 (Uniform-Cost Search), 5 (Smart Cache), and 6 (Iterative Cost Raise). These chapters include sub-chapters with information on the backgrounds and complexity of the algorithms used, which is necessary in order to interpret the results.

The behavior of the developed algorithms under parallel execution differs vastly from implementation to implementation, which is why chapter 7 (Multi-Processing) examines and discusses it in-depth.

The collected data on speed and performance has its own chapter 8 (Benchmarking), which leads to chapter 9 (Result), presenting all results in relation to each other. Finally, chapter 10 (Conclusion) discusses them and attempts to answer the thesis question.

Chapter 11 (Usage) explains how the program can be run and tested, in addition to showing how it can be extended with new implementations of Dodgson's rule.

The Appendix (chapter 12) contains a detailed presentation for the algorithm of chapter 4 (Uniform-Cost Search). A number of tables which were too long for chapters 7 (Multi-Processing) and 8 (Benchmarking) can also be found here. A full version of the code which receives updates is uploaded at sourceforge.net and can by reached via the link in chapter 10 (Conclusion).

1.2 The History of Voting

The goal of voting is to reach a conclusion which leaves most of the voters content. In essence this is the only constraint any vote-interpretation procedure — from now on called 'rule' — needs to fulfill, and also serves as a means of evaluation whenever two

rules disagree in their results.

In the year 1785, Nicolas de Condorcet proposed the following rule: When voting on different alternatives, a voter should specify for each alternative whether he prefers that alternative to any of the other ones or not.

Given a voter $v_1 \in V$, if there are a number of alternatives $a, b, c, ... \in A$, we can define a function $pref_{v_1}(b, a)$, written as $a < b$ which in this context means that voter v_1 prefers alternative b over alternative a. In any normal rule, this function is transitive, meaning that $a < b$ and $b < c$ implies that $a < c$. Keeping that in mind, we can shorten a voter's preferences to a list of the preference function in this manner: $v_1 : (a < b < c)$. This means that voter v_1 prefers b over a and c over b, and by transitive extension c over a.

If we collect all these preferences from all possible voters, we declare that alternative to be the winner of the vote which wins all pairwise comparisons against all other alternatives. In our example with only one voter, the winner would be alternative c, as c wins against both a and b in all cases. If there is more than one voter, a 'win' is defined as having more cases of wins than losses.

Given three voters v_1, v_2, $v_3 \in V$ with the profiles

$v_1 : (a < b < c)$

$v_2 : (b < c < a)$

$v_3 : (a < c < b)$

we would find the winner by ranking each alternative pairwise against all its opponents, as the rule states:

a against b: (v_1: loss, v_2: win, v_3: loss) $\rightarrow$ loss

a against c: (v_1: loss, v_2: win, v_3: loss) $\rightarrow$ loss

b against a: (v_1: win, v_2: loss, v_3: win) $\rightarrow$ win

b against c: (v_1: loss, v_2: loss, v_3: win) $\rightarrow$ loss

c against a: (v_1: win, v_2: loss, v_3: win) $\rightarrow$ win

c against b: (v_1: win, v_2: win, v_3: loss) $\rightarrow$ win

In this election, alternative c is the 'Condorcet winner', even though the majority did not vote for alternative c as its first choice. But if a vote was held which asked "would you prefer alternative x to win instead?", the majority would always vote 'no'. This is because the Condorcet method ensures that if a winner is found, that winner is preferred over any other alternative.

One other popular approach on how a vote on a number of alternatives using a full preference order can be resolved is the Borda count. It was proposed in the year 1770 by Jean-Charles de Borda, and is a popular method for ranking in sports events and the voting rule of choice in several Universities to this day.

The idea behind the Borda count is to assign a number of points to an option for each vote it receives, with the amount depending on its height in the preference list. If a rising amount of points is awarded in descending order, the alternative with the least amount of points wins. If an ascending order with shrinking amounts is chosen, the alternative with the highest amount of points wins[1].

[1]The classic Borda count uses a linear function for awarding points, meaning that the points y for an

If attaining first place with one voter gains an alternative 2 points, and each place lower gives one point less — the classic rule with $m-1$ to 0 points in descending order — we would get the scores in Table 1.1 for our exemplary election, here written as a preference profile, i.e. all preference lists in one matrix.

v_1	v_2	v_3
c	a	b
b	c	c
a	b	a

Table 1.1: a: 2 points, b: 3 points, c: 4 points

Using the Borda count, alternative c wins this election again. But while the Condorcet method can give an intuitive reason on why the chosen winner is the winner, the Borda count lacks such an explanation. It could be said that alternative c won because it was, on average, ranked higher than the others. However, it lacks the crisp, hard reasoning of the Condorcet method.

A last criticism is the fact that the linear order, which is presupposed for this rule can not be taken for granted. Especially in elections with many alternatives, a voter might not particularly care about the order of the last places, but by the nature of this rule, votes in the mid-field are still of comparably high importance.

1.3 Technical Terms and Definitions

The last example ended up producing the same winner using both the Condorcet method and the Borda count. This is not always the case. Given the preference profile in Table 1.2, we have c as the Condorcet winner, winning against both a and b three out of five times, and against d four out of five times in pairwise ranking. But using the Borda count, a wins the election with 11 points to c's 10.

v_1	v_2	v_3	v_4	v_5
c	a	b	c	a
a	b	c	a	c
b	d	a	b	b
d	c	d	d	d

Table 1.2: Borda count and Condorcet method disagree on who wins

Even worse: If voter v_4 changes his mind and decides that he actually prefers b over a, resulting in $v_4 : (d < a < b < c)$, a and c tie in the Borda count. And if further v_4 also decides that he prefers b over c, resulting in $v_4 : (d < a < c < b)$, we have no Condorcet

option ranked at place x are computed by $y = xm + b$, with $m, b \in Z$ and $m \neq 0$. Again, if m is positive, the alternative with the lowest count wins, and if m is negative, the alternative with the highest count wins.

winner! While a tie in the Borda count is actually not that bad, as we still know that both a and c are better than b and by far better than d, having no winner using the Condorcet method leaves us empty handed. This dilemma is known as the 'Condorcet paradoxon', as we end up with a cycle in a pairwise comparison. In this case, a wins against b, b wins against c and c wins against a.

The reason why these two rules can produce different winners is that they value different aspects of the information from the election. The Condorcet method only considers the pairwise comparisons which are extracted from the voters' preferences either directly or using the transitive character of the preference function. The Borda count uses the distance from each alternative to the top of the list. Intuitively, the Borda count uses more information. In the Condorcet method, the amount by which one alternative wins against another one is considered irrelevant – a win is a win. For v_1, c doesn't win 'more' against d than b. To decide if this information actually carries any relevant meaning or not is up to the person who chooses the rule – Condorcet or Borda.

In actual real-world problems, the basic Condorcet and Borda rules are not used as is. Instead, new rules founded on them are implemented. Borda-based rules may develop a point-awarding order which better represents the problem at hand[2]. For Condorcet-based rules, the term 'Condorcet-consistent' was coined. This means, that the rule in question will select the Condorcet winner if he exists, but may also offer a winner if he does not. This leaves a lot of the decision making up to the reasoning of the designer, but offers a justifiable winner in most cases anyway. One such rule, which tries to account for the information loss using the classic Condorcet method while still being strictly consistent with it is the Dodgson's rule.

1.4 The Frame of Dodgson's Rule

Being completely ignorant of Condorcet's work, Charles Dodgson devised a scoring system named after himself in the year 1876 under the name "A method of taking votes on more than two issues". Its most appealing feature is its intuitiveness. The idea of Dodgson's method is to ask how far a certain alternative in a preference profile is from being a Condorcet winner. If it already is the Condorcet winner, the answer is zero. If it is not, we try to count how many voters would need convincing to place that alternative higher in their preference in order for that alternative to win — one act of convincing leads to the alternative being placed one position higher than it used to be. At this point, the rule uses the actual distances between alternatives we lose against as information[3].

[2]Such as the electoral system of Nauru, in which points are awarded the following way: The winner gains 1 point, the second place $^1/_2$, the third $^1/_3$ and the nth place $^1/_n$ points. This makes the exact order of the last places less relevant. Earlier, I mentioned that this was a weakness of the classic linear point-award system, so this approach is a workaround adapted to the problem at hand.

[3]In Fishburn (1977), rules follow a classification by the amount of information they use, under which Dodgson's rule would then be of type "C2"(distances between alternatives) instead of "C1"(only pairwise comparison).

If we took the altered preference profile from 1.2 (The History of Voting) in which v_4 changed his mind twice, we would have a profile with no Condorcet winner, as can be seen in Table 1.3.

v_1	v_2	v_3	v_4	v_5
c	a	b	b	a
a	b	c	c	c
b	d	a	a	b
d	c	d	d	d

Table 1.3: Slightly altered version of Table 1.2

However, if we took the Dodgson scores of all alternatives, we would know how many 'swaps' of adjacent alternatives would need to be applied to the preference profile in order to turn that alternative into a Condorcet winner. In the case of Table 1.3, we have $sc_D(a) : 1$, $sc_D(b) : 1$, $sc_D(c) : 1$ and $sc_D(d) : 8$. sc_D is shorthand for Dodgson score.

a, b, and c each only lose against one other alternative, and as any one of them is in at least one preference list adjacent to their winning opponent, we only need to perform one swap for each. d on the other hand loses against every single one of the other alternatives, and with 8 swaps we get close to the theoretical maximum amount of swaps[4].

Depending on the good reputation of the Condorcet method, we choose the Condorcet winner if he exists, and give a ranking if he does not, with the ranking offering an account of how far that alternative is from being a Condorcet winner. Again, we end up with a very intuitive reasoning on what the final answer means.

However, though on the surface very attractive, Dodgson's rule has a number of significant weaknesses. The most impacting one is its computational complexity, as the amount of possible swaps which can be made grows exponentially with the amount of voters and alternatives. If both are sufficiently high, the problem becomes intractable and thus finding a threshold of what problem sizes are actually computable is one task of this thesis. The other deficits will only be discussed in this chapter, as they are not central to this work they can be summarized as Dodgson's rule not satisfying many criteria which are desirable for a voting rule. We will consider one of them in place of all others: the proof of Dodgson's rule not fulfilling the criterion of homogeneity, which requires a rule to be indifferent to cloning. Cloning means that the amount of voters is multiplied. These cloned voters have the exact same preferences as their originators. And as all of the voters are cloned and not only a subset, common sense would be that any rule which elects a winner in the original preference profile elects the same winner in the bigger version. But Dodgson's rule is not indifferent to cloning, as the following example shows (taken from Brandt (2009)). Note, that in these two preference profiles

[4]In this case, that would be 9. We can obtain it by counting how many swaps would be needed to transform the worst possible alternative into a Condorcet winner. The simplest way to do this is to swap it to the top of $\frac{n}{2} + 1$ agents' preference list. The formula would be $scD(max) = (m-1)(\frac{n}{2}+1)$.

the numbers above the preference list indicate how many voters hold these preferences.

2	2	2	2	2	1	1
d	b	c	d	a	a	d
c	c	a	b	b	d	a
a	a	b	c	c	b	b
b	d	d	a	d	c	c

Table 1.4: Original

Table 1.4: $scD(a) = 3$ (Winner), $scD(b) = 4$, $scD(c) = 4$, $scD(d) = 4$

6	6	6	6	6	3	3
d	b	c	d	a	a	d
c	c	a	b	b	d	a
a	a	b	c	c	b	b
b	d	d	a	d	c	c

Table 1.5: Clone

Table 1.5: $scD(a) = 7$, $scD(b) = 8$, $scD(c) = 8$, $scD(d) = 6$ (Winner)

Taking a close look at the scores and how they arise, however, we can see that this scenario is a very isolated case. b and d are completely identical in their amount of wins. a is in so far unique as it is the only alternative which does not tie. d ties against everyone except a. Now it becomes clear, why the amount of voters has an influence: Breaking a tie always requires only one voter to change his preference. One voter is a lot if only few voters are present, but it grows negligible for a high number of voters. The Dodgson score of d will shrink in comparison to the others as the voters multiply. Thus, what is necessary for this inhomogeneity to arise is the following:

- close proximity of all Dodgson scores[5]
- even amount of voters, so that ties can actually happen
- original has few voters, clone has several times more

Additionally, I would like to quote from Caragiannis (2010):

> "However, the cases where the ranking is hard to approximate are cases where the alternatives have very similar Dodgson scores. We would argue that in those cases it is not crucial, from Dodgson's point of view, which

[5]The likeliness of this is further discussed in chapter 8, as well as part of the future work suggestions. Suffice it to say that it is comparably unlikely, especially in real-world scenarios.

> alternative is elected, since they are all almost equally close to being Condorcet winners. Put another way, if the Dodgson score is a measure of an alternative's quality, the goal is simply to elect a good alternative according to this measure."

Concerning the other shortcomings which are not covered here in detail, I would reiterate that Dodgson's method is not chosen for its precision. Corner cases will therefore not worry us too much — it is its intuitive appeal which makes Dodgson's an interesting rule.

Bearing that in mind, the validity of approximations to Dodgson's rule is raised as long as they assert that they can stay within a certain range of the actual Dodgson score. So the goal of this thesis is twofold, both realized by the same work. Firstly, the practical capacities of problems which can be solved with a Dodgson's rule implementation will be explored. Those can then also be used to test approximation algorithms. While there do exist proposals for approximations (McCabe-Dansted, 2006; Caragiannis, 2010; Tideman, 1987) which try to solve one or both of the basic rule's weaknesses, I fail to find the resulting implementations. This might be due to code being difficult to debug, as efficient Dodgson's rules to check the accuracy for bigger problem sizes are not trivial. In so far, the fact that Dodgson's rule is intractable is in a way solved by its lack of accuracy – or more, its independence on it.

The means by which I will test for the limit of Dodgson's rule implementations is to try out a number of search heuristics and benchmark performance. Some are classical approaches to improving search space requirements and/or speed, others are approaches specifically tailored for this problem, which only follow a rough scaffolding of established algorithms.

The final algorithms might also be of interest for multi-agent systems, which often employ voting rules to reach a decision. This is also the reason, why 'voter' and 'agent' is used interchangeably in this project.

2. Finding a Baseline

2.1 The Baseline-Scorer

Finding the Dodgson scores for all alternatives within a preference profile is a somewhat complicated task with a wide array of possible approaches. In order to reliably assess the quality of a solution, a baseline needs to be established, meaning the most basic approach possible on whose performance all others build. How basic exactly this should be is debatable, so I present my baseline here and provide reasons as to why I designed it this way.

In the Baseline — as in all other approaches in this thesis — the problem of finding the Dodgson score is treated as a search problem in the classical sense. The search space is the space of all possible profiles, and the winning condition for a specific profile is a Condorcet winner with respect for the alternative we are investigating. Additionally, no other profile in the search space which is also a Condorcet winner may have a lower Dodgson score[1]. Thus, the scorer needs to generate the search space, and then search through it until it can reliably satisfy the conditions for at least one solution. In my implementation, another constraint was implemented, as the preceding approach is just a bit too basic. The search space will largely consist of solutions which can't even be assessed via the Dodson score measuring algorithm, as they are permutations with changes on more than the alternative in question. Also, profiles where the alternative is 'swapped down' are ignored. 'Swapping down' means there is an incrementation in the Dodgson score, but there is no way this swap would lead to the alternative becoming a Condorcet winner.

v_1
c
a
b

(a) Original

v_1
b
a
c

(b) Permutation 1

v_1
c
b
a

(c) Permutation 2

v_1
a
c
b

(d) Permutation 3

Table 2.1

[1]Strictly speaking, only the final winning permutation has a Dodgson score. But as I will often speak about profiles which can potentially be the winner, I will refer to their transient score as Dodgson score in order to avoid wordiness and confusion.

Table 2.1 will serve as an example. Permutation 1 would not be part of the search space, since there was a change in ranking but no change to a. Permutation 2 neither, as a was swapped down. Permutation 3 on the other hand satisfies all requirements and would be part of the possible solutions.

The search space is generated and stored together with the Dodgson scores for each profile, and once it is computed, a second algorithm will check for each of the profiles if it is a Condorcet winner. If that is the case, and the Dodgson score is lower than the current minimum, the profile and its score are saved as the current solution. Once the algorithm has evaluated the whole search space, the current solution becomes the final solution.

The size of the search space and the speed of the Condorcet winner finding algorithm are the two parameters we need to look at closely in order to assess the size requirements and speed of this algorithm. However, if this algorithm is evaluated in relation to the heuristics-enriched improvements, we only need to look at the search space size, as the Condorcet winner finding algorithm is identical for all approaches (located in `Util::isCW` for further inspection at the Sourceforge archive).

2.2 Complexity

If the search is run for all solutions, we can calculate the total search space[2] size the following way: if the number of voters is n, and the number of alternatives is m, then

$$\Phi(\text{basic, worst case}) = \sum_{i=1}^{m} i^n$$

$$\Phi(\text{basic, best case}) = m!^{\frac{n}{m}} m$$

Worst Case

v_1	v_2	v_3	v_4	v_5
a	a	a	a	a
b	b	b	b	b
c	c	c	c	c
d	d	d	d	d
e	e	e	e	e

Table 2.2: $n = 5$ and $m = 5$

$\Phi(a) = 1^5 = 1$
$\Phi(b) = 2^5 = 32$
$\Phi(c) = 3^5 = 243$
$\Phi(d) = 4^5 = 1024$
$\Phi(e) = 5^5 = 3125$

[2]While Φ usually refers to the set of all possible solutions, i.e. the search space, while calculating runtime boundaries we are only concerned with its cardinality. In order to keep the formulae more readable, just Φ is used instead of the correct $|\Phi|$.

The total search space is the sum of all individual search spaces, so $\Phi(all) = \Phi(a) + \Phi(b) + \Phi(c) + \Phi(d) + \Phi(e)$, and since $\Phi(x)$ can be expressed via its position in all profiles, and every single position in the profile from 1 to m must be occupied, it can be rewritten as $\Phi(all) = \sum_{i=1}^{m} i^n$. In this example that would be 4,425. If there were 10 agents instead of 5, it would be 10,874,275.

Best Case

v_1	v_2	v_3	v_4	v_5
a	e	d	c	b
b	a	e	d	c
c	b	a	e	d
d	c	b	a	e
e	d	c	b	a

$\Phi(a) = 5! = 120$
$\Phi(b) = 5! = 120$
$\Phi(c) = 5! = 120$
$\Phi(d) = 5! = 120$
$\Phi(e) = 5! = 120$

Table 2.3: $n = 5$ and $m = 5$

As all alternatives are equal, we only need to study one and assume its value for all others. Looking exemplary at alternative a, we can compute $\Phi(a)$ by multiplying each position it occupies in each voter's ranking. In our example that would be $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$, or 5!. If there were 10 voters, it would be $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$, or $5!^2$. As the search space grows by 5! each time m new voters appear, we can rewrite the formula as $m!^{\frac{n}{m}}$, which leads us to the form $\Phi(all) = m!^{\frac{n}{m}} m$.

It only works out so nicely though if n is divisible by m without a remainder. If that is not the case, the calculation becomes more complicated, and the previous formula only serves as an approximation.

Example for $n = [5..10]$ and $m = 5$ with

- actual search space $\Phi(n, m)$[3]
- estimation by $m!^{\frac{n}{m}} m$
- diff $= (1 - {}^{\text{estimation}}/_{\text{search space}}) \cdot 100\%$

[3]The exact search space was computed by running a simulation of the search space, and grows substantially more difficult for bigger problems. This is why an easy to calculate formula is so desirable.

n, m	$\Phi(n, m)$	$m!^{\frac{n}{m}}m$	diff
5, 5	600	600	0%
6, 5	1800	~1563	~13%
7, 5	5400	~4072	~25%
8, 5	14400	~10608	~26%
9, 5	32880	~27637	~16%
10, 5	72000	72000	0%

Table 2.4: Estimation and actual values for the best case

Note that the inaccuracy reaches its maximum when furthest away from the exact estimates where n is divisible by m without remainder. I assume that the fluctuations in between these points do not leave the order of magnitude. At the very least, we then have a respectable lower bound estimation.

Finally, comparing the divergence between the best and worst case of a certain problem size it becomes apparent that depending on what kind of profile we get, computing all Dodgson scores using the Baseline can take vastly differing amounts of time. Benchmarking will show what kind of variance actually exists.

3. Depth-First Search

3.1 The Depth-First Approach

This first approach seems only like a small improvement over the Baseline, however, when running it turns out to speed up the process by a huge margin.

The search space is treated like a tree, as even in the Baseline approach the search space is generated with a recursive function, which already mimics a depth-first search (DFS) behavior. The predicted improvement would be in needed disc space, as in a DFS every node is treated as a new solution and the winning conditions can then be tested locally. By doing that, there is no need to construct the whole search space before running the evaluation algorithm. Thus, while the Baseline needs memory in the range of $O(n^m)$, the requirements for a DFS would be only the depth of the tree, which is $O(n)$.

However, as it turns out, accessing and moving the memory around takes up a significant amount of time. Benchmarking this approach with the Baseline and identical problems reveals that for a mid-range problem size, DFS is nearly 10 times faster (for exact numbers, check chapter 8, Benchmarking).

This seems peculiar, as the size of the search space is still the same and always will be, since it contains by definition all the profiles which can theoretically be a solution. Initially, I assumed that the only way I could improve the speed of the algorithm is by taking shortcuts, i.e. pruning the tree or stopping the search as soon as we can assert that we have already found the best solution. This is not possible in a DFS though, as we have no clue if a found solution is the best one until we have run all possible solutions — meaning we have to traverse the entire search space.

Since I could not find something tangible in the algorithms themselves, I have to assume that the speedup is due to technical reasons (see the next section for more information). All calculations considering best/worst case scenarios are identical to the Baseline, so they will be omitted here. This solution represents the best basic approach to the problem, as it follows the generic information theory approach of DFS very closely and manages to solve the problem without too specific alterations to the core of the algorithm.

As mentioned in the last chapter, it is not set in stone as to which algorithm is the Baseline for this problem. It can be argued that DFS forms an upper bound of sorts in this regard. For that reason, later implementations will also be compared to DFS.

3.2 Reason for the Speedboost

While designing the algorithm, I did not expect this approach to be faster. After some research[1] on why it is not only a few percent faster but close to 90%, the only reasonable explanation seems to be that the recursive filling of a vector, as it is done in the Baseline algorithm, is rather slow. The Baseline uses `vector::push_back()`, which runs in $O(1)$, so the operation itself could not be the reason. Instead, it was suggested that it is the non-linear filling of the vector.

A strength of vectors in C++ is that they are very local concerning their memory. Usually, when filling one using a loop, we will only work in L1 cache with barely any cache misses, as memory reallocation happens very rarely — in most C++ implementations vector size doubles with each overfill, so in x insertions, there will be at most $log_2(x)$ re-allocations. And as we run insertion linearly, we would expect that many cache misses. In a recursive filling on the other hand, there are many function instances holding a reference to where they think the cache is, but every time there is a re-allocation, that reference will 'turn cold' and miss, turning an operation in L1 cache into a cache miss and a search in L3.

L1 cache operations on modern systems usually take less than a nanosecond. A miss leads to delay and a read on L3 cache, with ~100 nanoseconds access speed[2]. This seems to be the only possible source of such a performance difference.

At this point, it is important to highlight that the improvement in speed is not due to something clever thanks to the algorithm design. Having studied and compared both algorithms for a long time, I cannot see how the DFS could possibly be so much faster if not for an unforeseeable issue, such as the one proposed.

[1] Credit for a lot of the information presented on this topic is due to user 'cmaster' of the Stackoverflow forums, given here: `http://stackoverflow.com/questions/23117387/explaining-performance-difference-in-two-nearly-identical-algorithms`

[2] Numbers are taken from Intel i7 performance analysis at `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`

4. Uniform-Cost Search

4.1 The Uniform-Cost Approach

Uniform-cost search (UCS) is based on a breadth-first search (BFS). Again, the search space is treated as a tree in order to use this algorithm. If the edges of the tree are weighted, a uniform cost search can be performed. My implementation sticks relatively closely to the classical one, using a 'Node'-structure to encapsulate node-content, and running a priority queue as storage container. As this algorithm is the most simple of the advanced algorithms, mostly because it is not recursive, an in-depth explanation of my implementation together with the classic approach is given as pseudo code in chapter 12 (Appendix).

Central to this approach are two things: The initial choice of UCS as an algorithm scaffold, and the transformation to the problem which was necessary in order to perform it.

In the problem of finding the Dodgson score, obtaining a solution is not good enough – it must be minimal as well. A classical depth-first approach puts no useful preference over which path to chose and when to backtrack, as it only keeps track of local data. A BFS has larger space requirements because it stores more information, which is only useful if our problem needs to fulfill a 'global' condition. That is the case for Dodgson's rule: We seek the globally smallest possible solution. The other criterion, being a Condorcet winner, can be locally resolved. But solving the global condition locally is obviously not possible. That is why, in a DFS, we need to traverse the entire search space, as any found solution is useless unless we have seen them all.

UCS as a BFS lends itself as an implementation, as keeping track of the scores (which can be designed to act as costs with little effort) and exploring the search space while keeping them minimal comes naturally to it. By exploring the nodes first which have the lowest cost, we can assert that each node which is currently being examined is a global minimum for the whole search space!

But as always, there are drawbacks. Keeping global data may help speed up the algorithm, which is useful as the problems get bigger, but bigger problems also produce more data. If the data we need to store exceeds the memory capacities we have, the algorithm is of little use. For that reason we are interested in reducing the space a preference profile requires.

Right now, each time we create a new node, we would have to

- copy the father profile, and perform the swaps on it accordingly
- compute the new current Dodgson score resulting from the change
- sort the profile into our memory

Another problem is the fact that preference profiles, when thought of in terms of change over time, are hard to work with. More code is necessary in order to transform or extract the information we currently need from them.

That is why a different representation was chosen for this approach, which also served me well in the following two implementations. Initially called 'swap-history', as it stores the history of swaps which the profile was subject to, it was renamed to 'swap-profile' when I noticed that it could replace the preference profile in all regards.

The idea is to only store the original profile once, together with the alternative we are examining. Then, all new profiles can be thought of as variants of that profile in which only the position of the current alternative per agent is changed. We obtain a vector of integers of cardinality n ($\rightarrow$ number of agents) per node, instead of the full profile. Additionally, the sum of all values in the vector is the Dodgson score, which makes comparison operations between nodes straight forward, thus facilitating the sorting procedure.

Formulating the problem in a way that makes it processable by UCS by treating the Dodgson scores as costs and using a swap-profile instead of a preference profile was the most important piece of work, and makes this approach non-trivial. Additionally, some other methods were used to speed up the algorithm, which can be studied in the full explanation of the code.

4.2 Complexity

This algorithm works in a vastly different way from the two discussed before, so a detailed account on best and worst cases will be given.

Probability of a best/worst case:

As the speed of this algorithm depends solely on the moment the first solution appears in the tree, it is important to check how good or bad it can actually get, and in how far that affects runtime. The answer to the first question is identical to the examples given in chapter 2 — the best and worst case profiles and the resulting search space size still look the same here. But since we stop at the first possible solution, we get a second number apart from Φ, which is the position of the first valid solution. As we traverse the search space in an ordered fashion, we can think of the search space as ordered, with the Dodgson score being the sorting criterion. Using UCS, we will only check the part of Φ which comes before the first solution. Thus, we obtain $C \subseteq \Phi \mid sc_D(c) \leq sc_D(\phi), \forall c \in C, \forall \phi \in \Phi \backslash C$, the actually traversed search space subset. The analysis in chapter 8 (Benchmarking) contains a number of graphics showing how big C and Φ are in comparison to each other. Calculating C is what I will try in this subsection.

4.3 Worst Case

v_1	v_2	v_3	v_4	v_5
a	a	a	a	a
b	b	b	b	b
c	c	c	c	c
d	d	d	d	d
e	e	e	e	e

Table 4.1: Worst case

Exemplary for alternative e

Since the majority of each solution has to be worse than e if e is supposed to be the Condorcet winner, it needs to be at the top of more than half of the voting agents. In this example, that would be three. In order to get e swapped to the top of three profiles, we need to check $5 \cdot 5 \cdot 5$ permutations, as e is always at the bottom. Hence for e, the number of checked solutions is $C(e) = 5^3$

Generalizing a bit, we can see that the base is the position the alternative has in all profiles — index of e, or i_e — and the exponent is the majority of agents, or $\frac{n}{2} + 1$, given integer division:

$C(x) = i_x^{\frac{n}{2}+1}$

With $C(all)$ then being: $C(all) = \sum_{i=1}^{m} i^{\frac{n}{2}+1}$

For comparison, for m ranging from 1 to 10 and $n = 5$, we have

- total search space $\Phi(n, m)$
- actually traversed space $C(n, m)$
- factor by which Φ is bigger than C
- the amount C is from Φ in percent: $\frac{100}{\Phi/C}$

n, m	$\Phi(n, m)$	$C(n, m)$	factor	diff
1, 5	1	1	1.0	100.0000%
2, 5	33	9	3.667	13.2727%
3, 5	276	36	7.667	13.0435%
4, 5	1300	100	13.0	7.6923%
5, 5	4425	225	19.667	5.0848%
6, 5	12201	441	27.667	3.6145%
7, 5	29008	784	37.0	2.7207%
8, 5	61776	1296	47.667	2.0979%
9, 5	120825	2025	59.667	1.6760%
10, 5	220825	3025	73.0	1.3699%

Table 4.2

Or, for any n and m it can be said that, in a worst case scenario with all other aspects being equal, an algorithm traversing C instead of Φ is — roughly — 'factor' times faster.

How probable is the worst case actually?

In a random setup, the worst case seems rather unlikely. While benchmarking, runs with 100 different random preference profiles were made, but their average duration would still vary substantially from run to run. The reason for that is that one or two really bad cases tend to dominate the whole outcome, leading to a standard deviation of 7000 and higher for UCS (more in depth information in chapter 8, Benchmarking). This of course implies that there were few but heavy outliers. After checking the values, it turns out that the outliers are the runs which had an unusually high duration. This however, does not mean that 'worst case scenarios' are as sparse in a real-world setup as well. It is probably a lot more likely in actual elections, as there exists a bias among alternatives. This means that an alternative which is liked by one is often also liked by others, and one which is disliked by one is often also disliked by others. This, taken to its extreme of all agents having the exact same preferences, leads to the worst case. Hence, it can absolutely not be discarded as improbable.

At this point it comes to mind that it is very unfortunate that realistic problems tend to shift towards the worst case scenario, given that the worst case is also a lot worse than a best case. Chapter 7 (Multi-Processing) includes a proposition on how to solve this problem.

4.4 Best Case

Same as for the worst case, I will try to estimate C in relation to Φ. Fulfilling the Condorcet condition is a more complex problem for the best case though, as the agents' preferences are not interchangeable. Finding a formula which counts all possible

solutions was my first approach to solving the problem, but it turned out to be a variant of the summand factorization problem — more information on that in the following chapter — so I abandoned it for a more bland and less descriptive, but at least feasible approach.

Since we work through the search space in an ordered way, we can assert that if the first winning profile has a Dodgson score of x, we will work through at most all profiles with a Dodgson score of x and lower until we are done.

The minimal Dodgson score for a best case scenario with n being a multiple of m — which are the only ones we consider, for simplicities' sake — can be computed the following way:

$$sc_d min(x) = \frac{n}{m} \sum_{i=1}^{n/2} i$$

We then sort solutions by their Dodgson score and count how many have a smaller or equal and how many a bigger score than our minimal one. As I said, I have no formula which calculates the ratio between C and Φ. Instead, I will give the first 12[1] examples which compute the space given $n = m$. Chapter 8, Benchmarking, also contains a few simple plots that give a rough account on how the profiles sorted by Dodgson score are distributed over the search space, and where C is located. 'Diff' shows how much of the search space in percent we skip at least using this algorithm.

n, m	Φ	C	diff
1	1	1	0.0%
2	2	2	0.0%
3	6	3	50.0%
4	24	15	37.5%
5	120	29	75.8%
6	720	259	64.0%
7	5040	602	88.1%
8	40320	8039	80.1%
9	362880	21671	94.0%
10	3628800	392588	89.2%
11	39916600	1200900	97.0%
12	479001600	27770328	94.2%

Table 4.3: Maximum possible gain

As can be seen, as soon as the problems get bigger, we can save quite a bit by stopping early. Up until now, this is the most we can hope for using an algorithm which stops early. In any of the searches using such a global minimum policy (besides UCS also the two following ones), this case can occur, and often enough it also does. Counting

[1] After $n = m = 12$, counting permutations took too long, so I only did it up to 12×12.

the amount of checks performed by the scorer, we have exact information on how many 'nodes' were explored in the search space tree.

5. Smart Cache

5.1 The Smart Caching Approach

This algorithm focuses even more on the fact that we don't directly handle the preference profiles anymore and use the swap profiles instead. In this chapter, I will refer to the search space as 'swap space', as we treat it as such.

Smart caching (SC) is an alternative way to make a semi-informed search on the swap space while pruning it at the same time. The swap space is a representation of the search space which is defined by three things:

1. the initial preference profile with the currently examined alternative
2. swap profiles (as opposed to preference profiles) which represent the possible solutions
3. a position table, which is used to translate the swap-profiles back into a preference profile

The idea behind SC is that by treating the search space as a swap space, we can more easily sort our potential solutions. The new core property is that it is now possible to generate swap profiles with a specific Dodgson score, as the Dodgson score is a very natural part of each swap-profile: It can be obtained by summing up all of its entries. Thus, a swap profile can be generated by starting with the Dodgson score, and then distributing a summand partition of the score among the n agents.

This would already be the general core of the SC algorithm. We start with the lowest possible Dodgson score, and generate all possible swap-profiles from it. If no solution was found among them, we go on to the next highest Dodgson score, and so on. This gives the approach its name, as we are caching a certain subspace of all possibilities in order to search a valid solution among them.

The position table is critical to the usefulness of this approach, too, as with it we can make sure that no impossible profile is generated. By knowing where the alternative is located at all times, we can stop generating permutations as soon as they are no longer possible. If we don't, we end up with many useless profiles, which all have to be removed. As I currently use a recursive generating algorithm, the gain from just one early stop can be immense, as all consecutive faulty builds are pruned away.

Smartly generating summand partitions of the Dodgson score[1] is the first step of the algorithm. We restrict the amount of summands, and further have an upper bound for each agent. The result is the set of all possible factorization of any Dodgson score we use as input, which translates directly into the set of all swap profiles of the same Dodgson score. We start at Dodgson score zero and generate the corresponding set. If no Condorcet winner is found among its elements, we increment the current score and generate the next set, check again, and so on. Since we can assert that any profile which satisfies the Condorcet criterion in the current set is also a global minimum, we can stop as soon as we find the first solution.

Looking at the big picture, we no longer see the search space as a tree which we try to traverse in an ordered fashion, backtracking when necessary. Instead, the search space is now a layered set, with no necessary connectivity between its elements.

The finally generated space size depends on how soon we find a solution. Analysis shows that the amount of profiles plotted against their scores behaves like a quadratic function[2], which is why finding an early solution has such a huge payoff. In a way, this solution is comparable to the Baseline, as they both cache the search space in one algorithm and search for a solution in another one. But even in a bad case, we only cache a fraction of the total space at any point in time, and since we moved from preference profiles to swap-profiles, we need even less memory for those we do store.

While I did say that we can stop the searching algorithm as soon as we find the first winner, the current version finishes only after checking the complete current layer in order to collect all minimal solutions. This was necessary, as comparing solutions with the other scorers previously led to constant discrepancy, since all use different ways of looking for theirs. While they all had the same Dodgson score, debugging was easier if all winning profiles were collected, and not just the first random one encountered. It could also be important in some settings to find all, and as it does not dramatically raise the problem size[3], I decided to keep it that way.

But, same as in the Baseline, the amount of overhead and delay caused by recursively storing, accessing, and transforming solutions will lead to a substantial penalty in speed.

[1] Writing the algorithm which does the factorization was not trivial, as the amount of permutations is immense, and the order of the summands is relevant as well. In the end, a rather complicated recursive algorithm heavily reliant on the position table was used. Developing a formula which computes the amount of possible factorizations using the problem size and the position-table as input should at this point be possible, and would have been useful for estimating C in the last chapter, but I could not come up with one – so simulating lower problem sizes must suffice at this point.

[2] Estimation only, see chapter 10 (Conclusion) for more information. For the plots, see chapter 8, Benchmarking.

[3] In fact, in big-O notation the complexity stays the same, as it could be written as $O(n \cdot (1 - p))$, with n being the layer size and $p = [0..1]$ the probability of finding a solution at any point in the layer. As factors are dropped, this is considered equal to $O(n)$, the amount of work it would take to search through the complete layer.

6. Iterative Cost Raise

The Iterative Cost Raise (ICR) algorithm is an improvement to the SC approach in the way that DFS is an improvement to the Baseline. On the surface, it behaves like an iterative deepening depth first search, which also served as inspiration for its name. But as no depth first search is employed, the surface is where the similarity ends. Also, classic iterative deepening needs to re-explore previous checked nodes, whereas the core attribute of both SC and ICR is, that we can chose precisely which states to explore, so we don't have to repeat any previous work.

6.1 Improvements and Payoff

While SC starts with collecting all profiles for a certain Dodgson score, ICR sends a profile to an evaluation function as soon as it is generated. This should hopefully give us the arcane benefit of not recursively storing values in a large vector (see section 3.2 (Reason for the Speedboost) for more information), and we also get to stop as soon as a solution is found in the swap-space before the layer is fully checked.

SC could only stop after already having generated all profiles for the current Dodgson score. The gain of stopping early in the generation algorithm can go from very small to very large, depending on how fast a solution appears in the final layer. If worse comes to worst, and the last profile of the layer contains the solution, there is no gain at all[1]. Compared to DFS, the benefits of not storing the results should have a much less significant impact though. First of all, the evaluation algorithm is more complex for ICR and SC, so the profile access should play less of a role. Secondly, as we only cache subsets of the search space with smaller representations, the maximum of what we can possibly save is smaller as well.

But that is purely hypothetical. The benchmarking with the Baseline compared to DFS shows that the result could still be surprising.

While an in-depth comparison between all scorers performance will be given in the next chapter, I will show one result here already in order to highlight an important point.

[1]Same as in Smart Cache, this does not change the complexity of the problem. But as the amount of winning profiles also grows with problem size, the probability to find an early solution is actually quite high even if it does only speed up the process by a factor. Just to give an idea, the 3×3 profile I used in section 1.2 of The History of Voting, only has 1 possible profile for the Dodgson winner. But during the benchmarking with 8×5 profiles there would often be more than ten valid solutions as soon as the scores reached four or higher.

When testing SC and ICR with a 8×5 preference profile, we got averaged results such as these:

type	average duration	average function calls
SC	9723 clock ticks	67235
ICR	1057 clock ticks	49032

Table 6.1: Performance compared to computation steps

As it turns out, ICR performs better than expected. A useful bit of information is that the shortcuts which ICR takes only lead to an average of ~27% less function calls, and thus only that much of the nearly tenfold speedup can be attributed to it. It seems that a big part of the unaccounted improvement is due to 'technical reasons' again.

7. Multi-Processing

7.1 Threading

Chapter UCS mentioned a weakness of all current implementations, namely that the cases which are most likely in a real world scenario are the most difficult ones to solve. This is also counter intuitive, because as a human, those cases are the easiest to solve, since there often is a clear winner with a low, easily calculable Dodgson score. It is important to realize though, that we are only interested in that winner. This is not obvious, resolving a tournament usually requires ranking all alternatives, and not every rule even allows to meaningfully rank an individual alternative while ignoring the ranking of the others. This is the case for the naive Dodgson's rule. Even if we finished obtaining the score for an alternative, the most important information is whether it wins the tournament with that score. However, in the final three implementations we can assert the minimality of the current solution. This enables us to:

- Run all alternative searches simultaneously, which was intended to be possible for them. The core algorithms only take the alternative as parameter, and are then called m times in order to produce a solution. As soon as a solution is found, all algorithms whose current Dodgson score exceeds that solution's score can stop running.
- Run the solutions procedurally, but stop as soon as the Dodgson score of an already computed alternative is exceeded. This can be further improved by estimating which solutions will produce a low Dodgson score and running those first.[1]
- Do both.

[1] An easy heuristic would be to sum up their distances to the top for all voters, i.e. sort them using the Borda-Count.

v_1	v_2	v_3	v_4	v_5
a	a	a	a	a
b	b	b	b	b
c	c	c	c	c
d	d	d	d	d
e	e	e	e	e

Table 7.1

If we did this with our current worst-case scenario (e.g. Table 7.1) it would, curiously enough, turn into a best case. BordaCount(a) would return the highest value of all alternatives, and running a first would then produce an instant solution and halt computation. Even more interestingly, the very balanced former best case turns into a worst case, in which all solutions need to be fully computed, as none is better than the next.

Effectively, this means that what before was the best we could hope for is now our — comparably manageable — worst case! Implementing these changes, at the cost of only getting the Dodgson winner, gives us an incredible speed boost. An additional benefit is how probable our new best- and worst-cases are.

New complexity

A best case occurs each time the profile contains a Condorcet winner, as that leads to an immediate halt. How probable this is depends on the number of agents and alternatives: the more there are, the less likely it gets[2]. In many cases, the chances for a best case are good though. This is also very apparent when benchmarking the scorers implementing multithreading, as a good amount of computations end after ~0.03 seconds, which only really leaves enough time for initializing the scorer and then directly finding a solution. The worst case, with each alternative being perfectly symmetric to all others, is a lot less likely, as it is a constraint for every single alternative — only one of them needs to break the pattern, and we end up with a much more favorable case.

Wrapping up, we have a very good situation at hand, in which the total speed solely depends on the easiest to compute alternative. And since alternatives do not exist independent of each other, we cannot find ourselves with only bad alternatives. The worst we can get is having many mediocre alternatives. And even if that happens and we cannot compute a solution, we can halt computation when a set time expires and assert that no alternative is as good as the score at which we interrupted. Depending on the problem, that might be useful information.

As an example of how much we gain if the Dodgson score of the minimal alternative shrinks, the worst case with one alternative braking the pattern will be shown. Again, C is used to denote the subset of the search space which contains all permutations lower

[2]Precise numbers and formulas can be found in Gehrlein, 1999, Approximating the Probability that a Condorcet Winner Exists. A number of exemplary values can be checked in 12.1(Tables). Note that these are extremely good probabilities for a best case scenario.

or equal to a specific Dodgson score. This example is very minimalist, and builds on much of the information offered in previous chapters without further explanation.

v_1	v_2	v_3	v_4	v_5	v_6
a	f	e	d	c	b
b	a	f	e	d	c
c	b	a	f	e	d
d	c	b	a	f	e
e	d	c	b	a	f
f	e	d	c	b	a

Table 7.2: New worst case for 6×6 with minimal scD = 6

$$C(6) = 6! = 720$$

We can assume that this amount of iterations through the algorithm is necessary until a solution is found. If even one single ranking in this profile is different, the minimal scD falls to 5.

$$C(5) = 5! = 120$$

The immediate gain from this small change is a speedup of 83%! Given that this worst-case is one of 6^6 possible permutations of this profile, we can claim that worst case scenarios in this setup are sparse. In benchmarking, I tested under impartial culture assumption, so no bias away from worst cases is given. In a real world scenario, we can assume that such a completely neutral preference profile is even more unlikely than its probability under impartial culture.

Amount of Threads

The amount of initiated threads is an important parameter and should be chosen according to the number of cores on the running system. The algorithms are optimal if a solution is found as soon as possible. The heuristic can be off, so we would want to run as many candidates as possible.

However, running too many can actually slow the algorithm down, especially if the number of threads exceeds the number of cores. Firstly, having the best candidates not evaluated as fast as possible is bad, as only the winner winning stops the solver. Secondly, the recursive algorithms cannot terminate at any point in time, they can only check for early abort after a complete round of backtracking, which can take a while when testing for higher Dodgson scores. Those algorithms will profit from unlikely to win alternatives not running at the same time as their more promising counterparts.

8. Benchmarking

8.1 Randomness

Even though it might not be realistic, I assume impartial culture for all agents in my benchmarking. While this makes most sense for testing the algorithms, it needs to be considered if the runtime should be expected to be higher or lower in a more realistic setup. The assumption for a realistic scenario is that agents will more often than not have similar preferences. For the normal variants of all algorithms, this approaches the worst case. For the threaded variants, it approaches the best case.

The algorithm which generates the preference profiles works the following way, given n and m as dimensions of the profile:

1. create n vectors of size m
2. create a pseudo random number generator using the current time as initial seed.
3. for each of the n vectors, do the following:
   (a) run a loop from 0 to m-1
   (b) in each iteration, get a new random number modulo m, and put the number of the current iteration into the current vector, taking the randomized number as position
   (c) if a number was already put there, re-do the loop iteration with the next random number.

As far as I can see, this algorithm is completely unbiased with the distribution of its values.

When benchmarking several scorers in turn, the initialization seed is saved before the loop runs all scorers, and is re-used each turn. That way, the generation algorithm will create the exact same profiles for their randomized runs. This is a huge advantage of using pseudo-random number sequences, as it assures absolute fairness. Otherwise, results may differ substantially from run to run, especially in this problem where heavy outliers are not too unlikely.

8.2 Average Performance

The first criterion by which I tested the scorers was average performance. A problem size was picked which uses a good amount of time, so that rounding errors or performance lows don't influence the outcome too much. At the same time, it needed to be small enough that enough data can actually be collected in a reasonable time.

My choice fell on a 8×5 preference profile, and the following values were obtained for 1000 runs using the standard algorithms. The unit of measure is clock ticks, with 1 clock tick taking one millisecond on the used system. Best performance is bolded. The standard deviation is written as σ.

scorer	*min*	*median*	*max*	*mean*	σ	*avg.calls*
Baseline	6377	15805	137865	18135	10759	**16425**
DFS	847	1608	**13564**	2028	**1193**	16527
UCS	204	1685	85254	3693	6241	17540
SC	76	1642	320828	6313	20020	76409
ICR	**50**	**552**	27557	**1202**	2042	55706

Table 8.1: Average performance of normal algorithms

The last three algorithms were also tested on their average performance on the same problem using their multi-processing algorithm. Baseline and DFS were run with their normal algorithm, as they then test on the exact same profile in order to give the best possible comparison. As expected, their performance is not much different from the former test – multi-processing approaches on the other hand show their impressive power. Again, an 8×5 profile with 1000 runs was chosen.

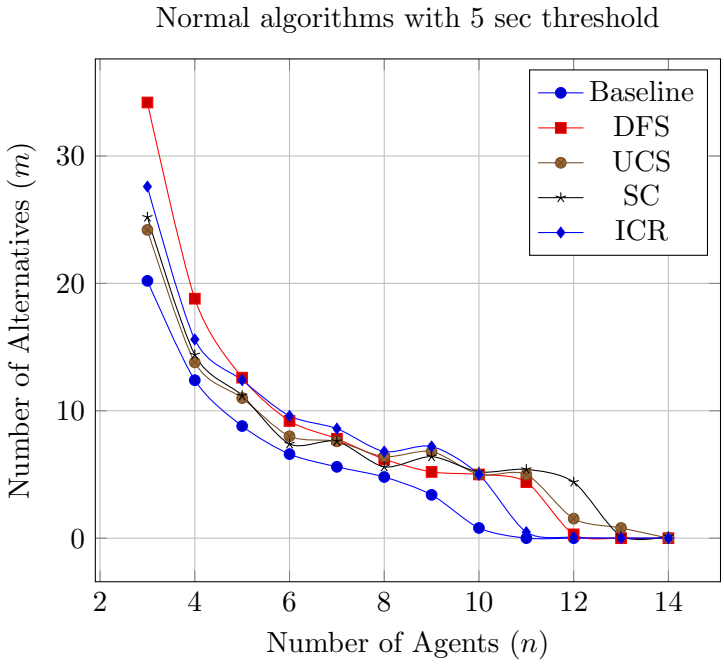
scorer	*min*	*median*	*max*	*mean*	σ	*avg.calls*
Baseline	7106	15090	110112	18016	9707	16536
DFS	913	1836	10712	2167	1169	16558
UCS	3	14	145	20	21	**225**
SC	**0**	3	**55**	**5**	**6**	434
ICR	**0**	**2**	67	**5**	**6**	407

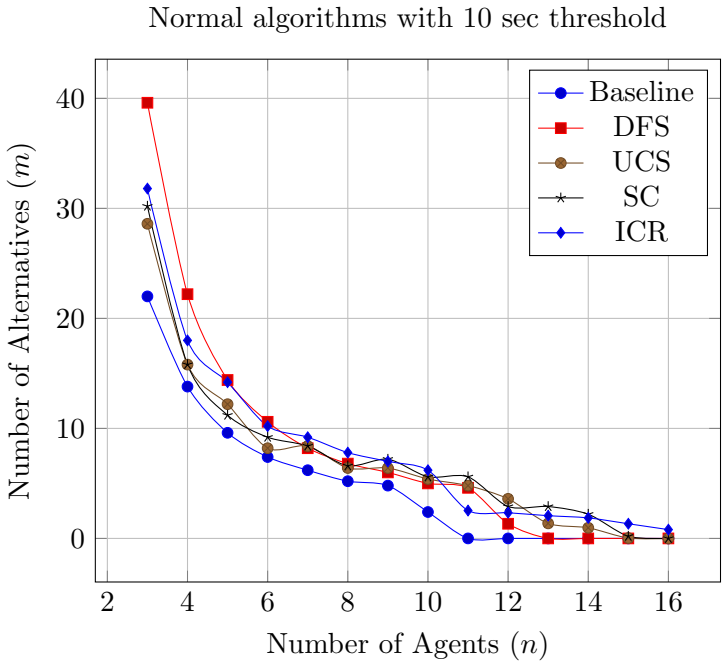
Table 8.2: Average performance of threaded algorithms

8.3 Maximum Range

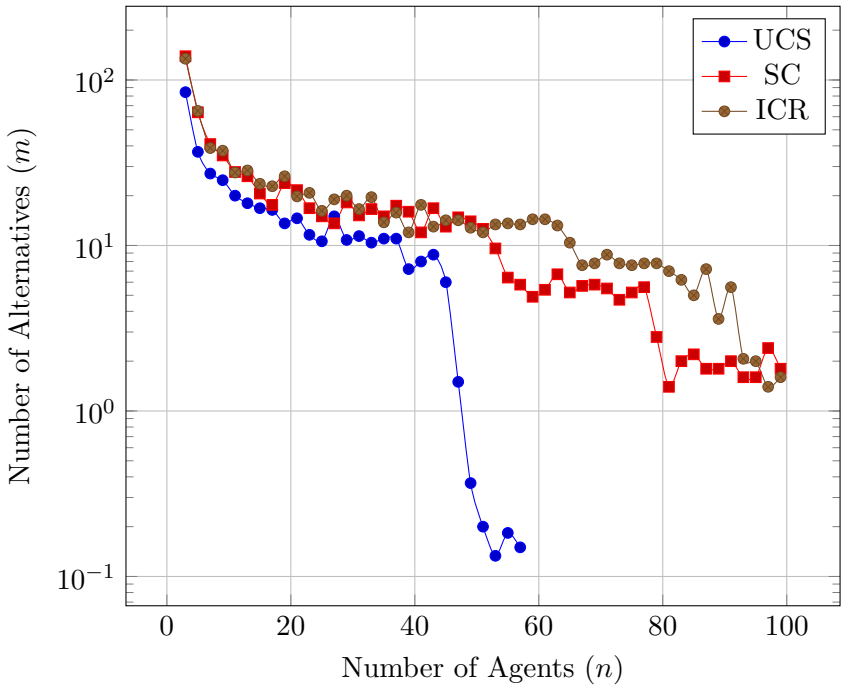
As our problem's complexity scales exponentially, knowing performance on one data point alone does not serve us very well. The following numbers were obtained from running each algorithm with increasingly complex problems. In this test, n is kept constant and m is increased by one each time the currently tested algorithm finds a solution. But if a previously set time window is exceeded, the final value of m is stored,

and the process is repeated with $n+1$. The process is stopped for the current scorer, if m could not exceed 4. The averaged results for 5 iterations are presented in the following graphs:

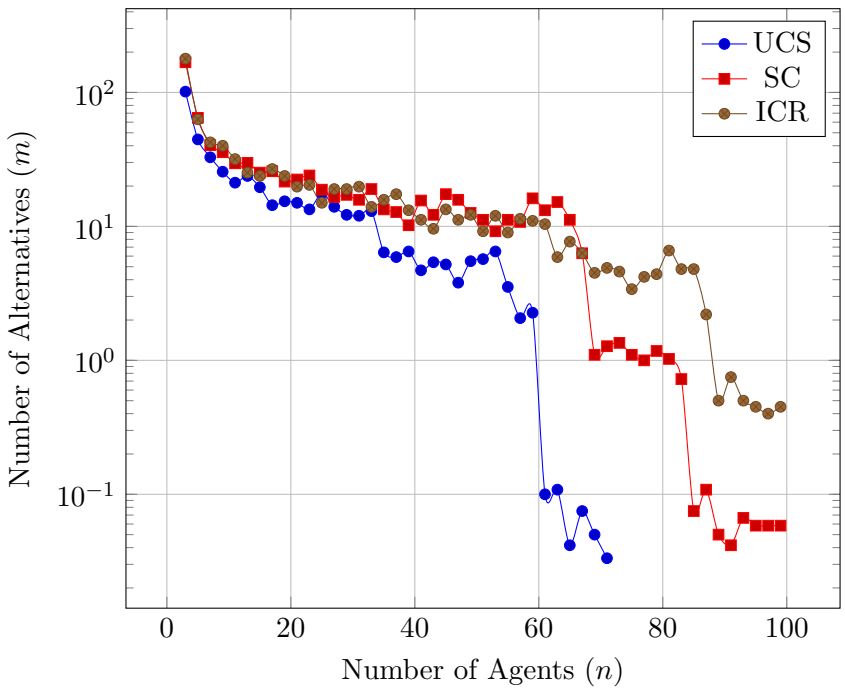




Threaded algorithms with 5 sec threshold



Threaded algorithms with 10 sec threshold

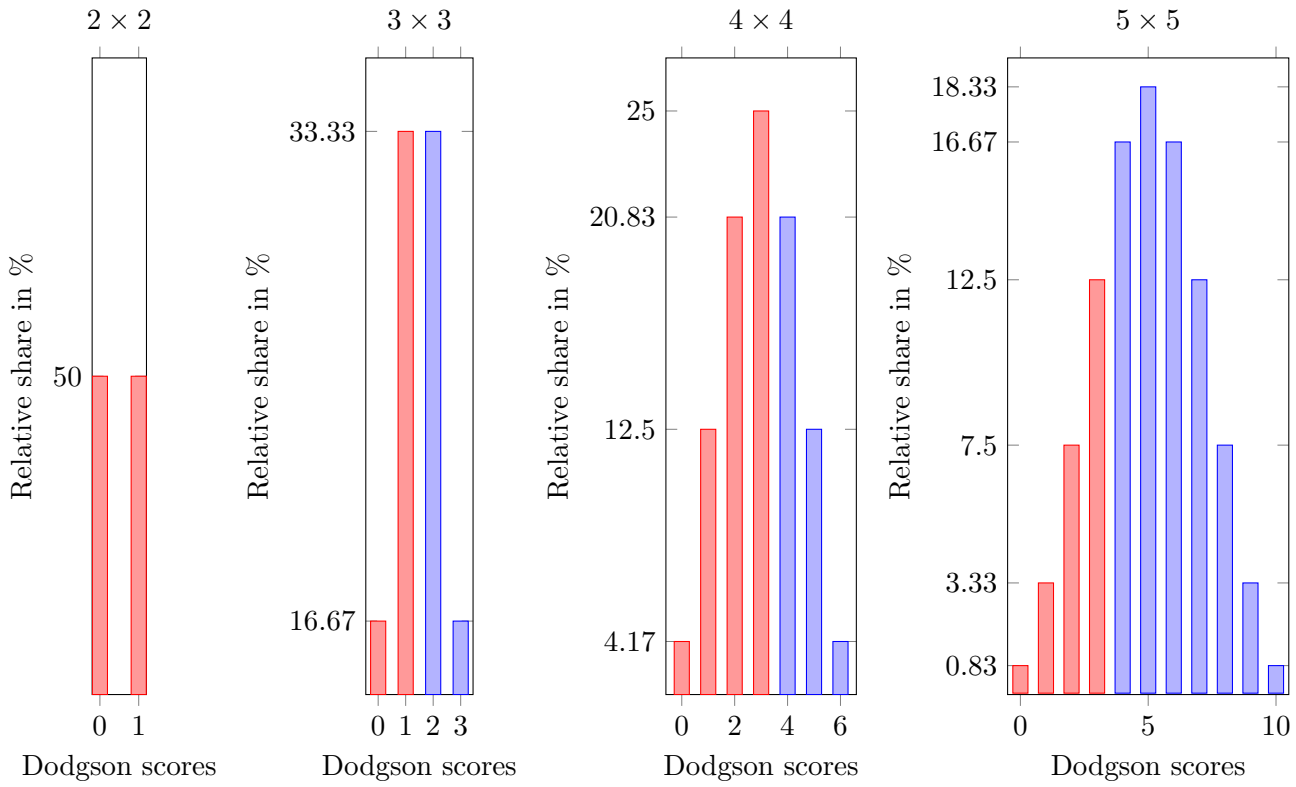


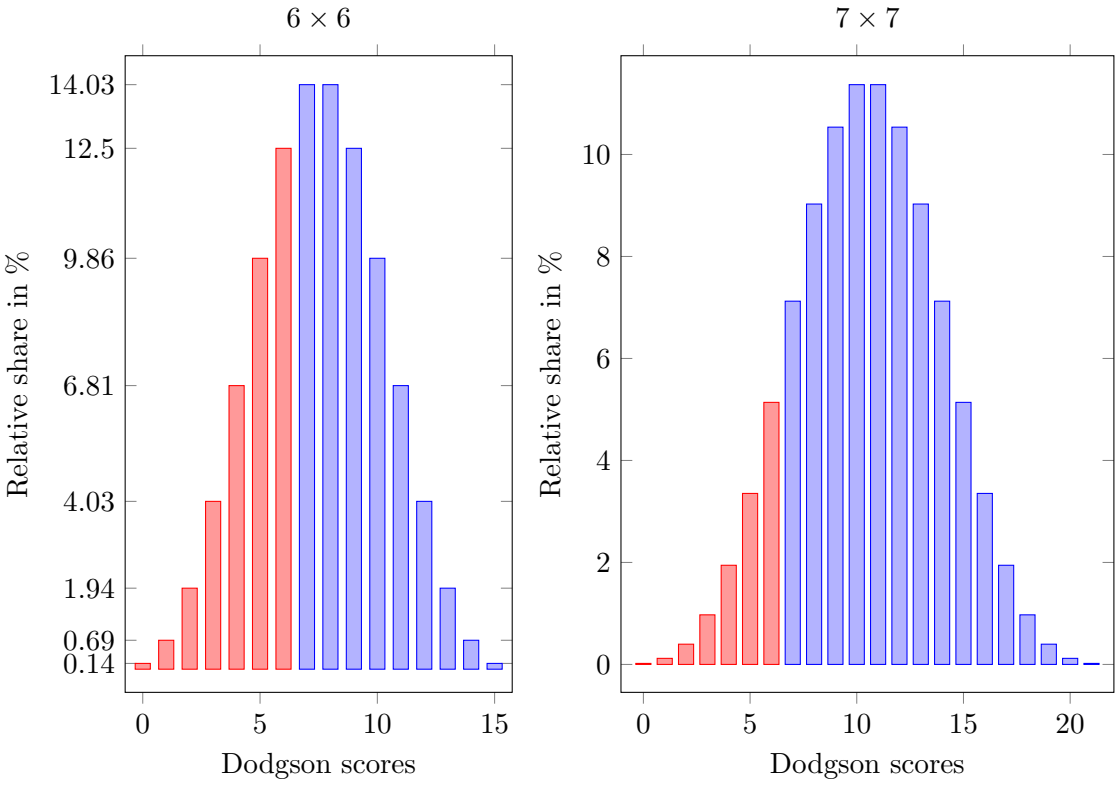
In the threaded graphs, uneven values for n were omitted from the data. This is done because the performance of multi-processing algorithms is heavily influenced by the probability of there being a Condorcet winner. An uneven number of agents makes that a lot less likely due to ties, which results in the graph spiking up and down after every value. Note, that in the tables of Gehrlein (1999), all preference profiles with uneven numbers of agents were omitted as well.

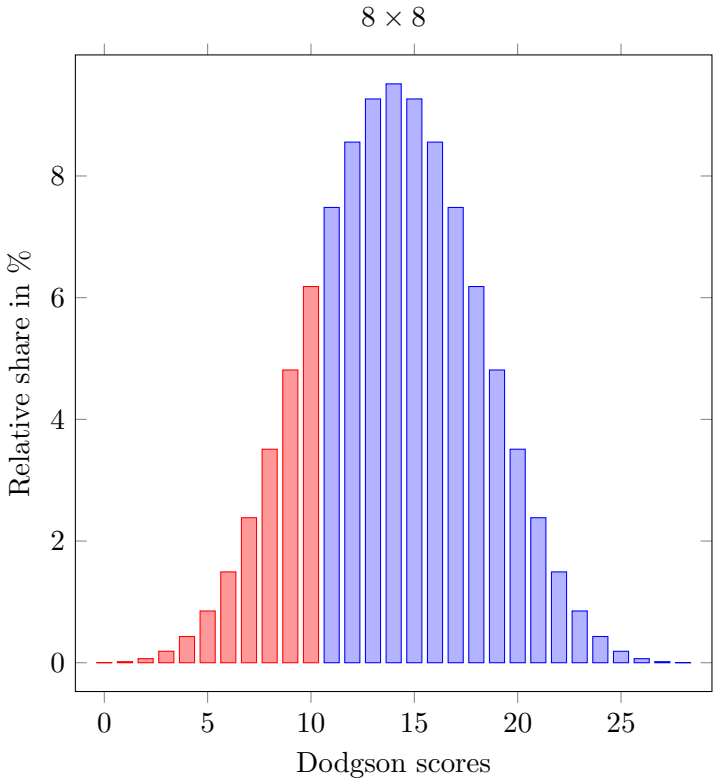
The exact values obtained from the test, including those with uneven n, can be checked at 12.1 (Tables).

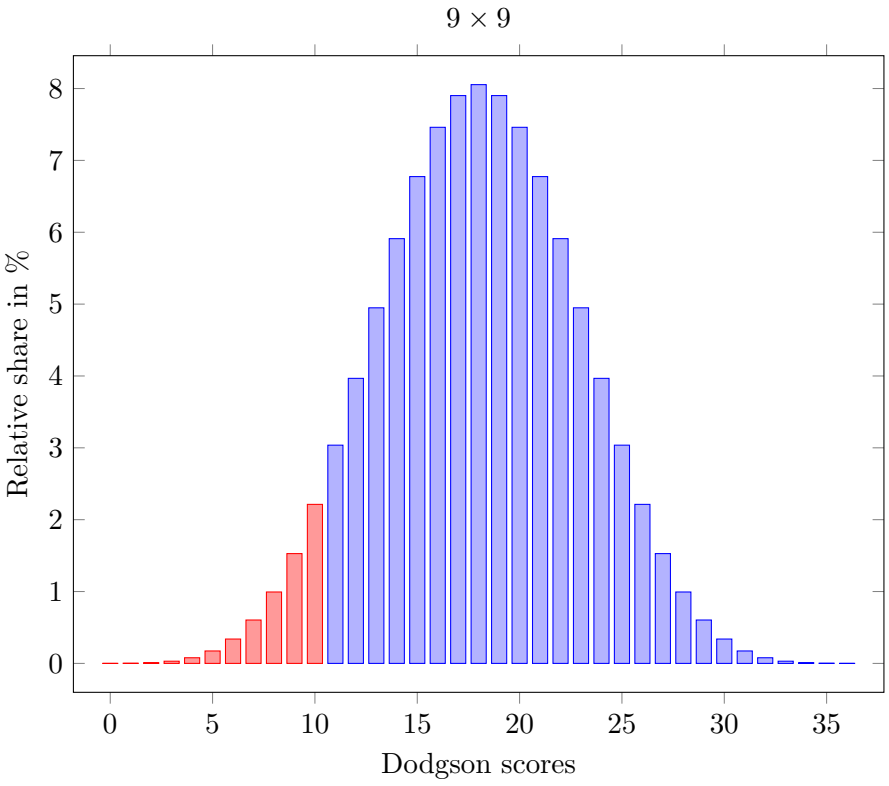
8.4 Analysis of the Search Space

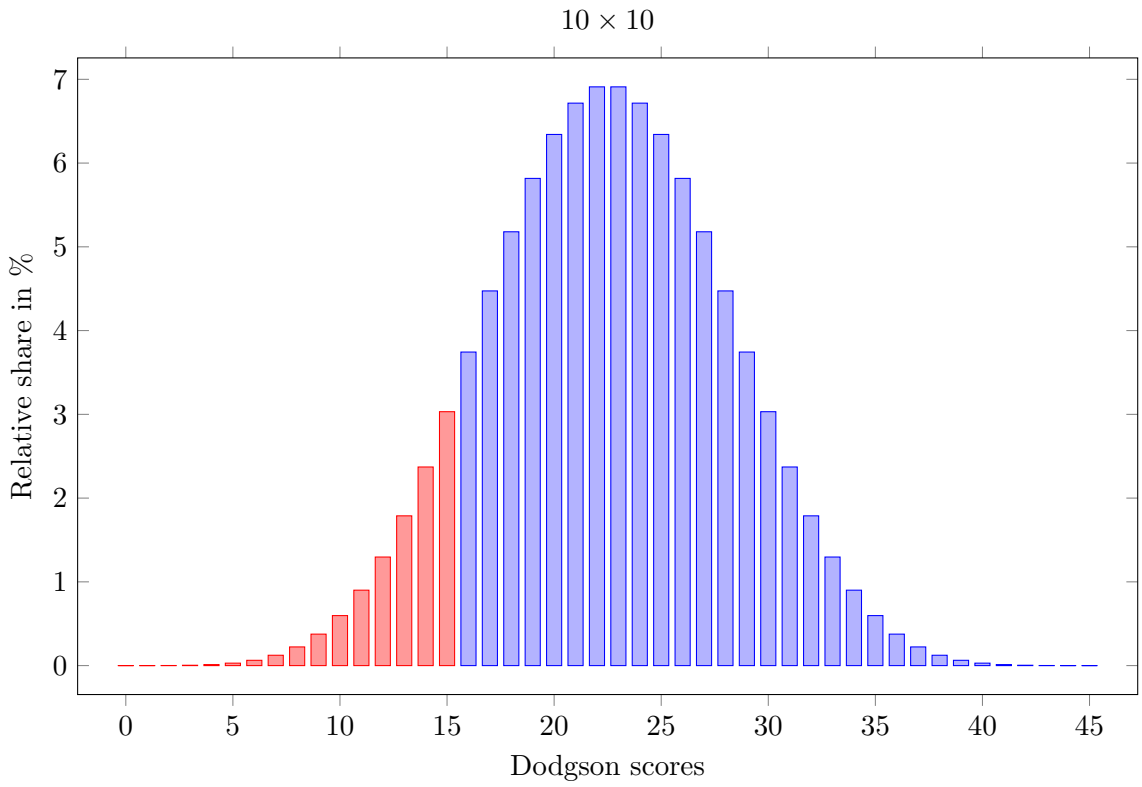
In order to visualize the relation of the total search space (Φ) to the actually traversed search space (C) of the latter algorithms, I wrote a program which models a couple of simple and regular model cases. Each column represents the relative amount of profiles which have the Dodgson score indicated by the column's label. The arrow indicates the position of the Dodgson winner, thus marking the end of C. Note that the relative size of C gets very small as the problem size grows, as can also be seen by the exact values listed in chapter 4 (Uniform-Cost Search). The relevant source code is located at `Menu::analyzeDistribution` and `Menu::fillBuckets`, and can be inspected at the Sourceforge archive. In the graphs, the orange bars represent C and the blue bars all of Φ which is not part of C.

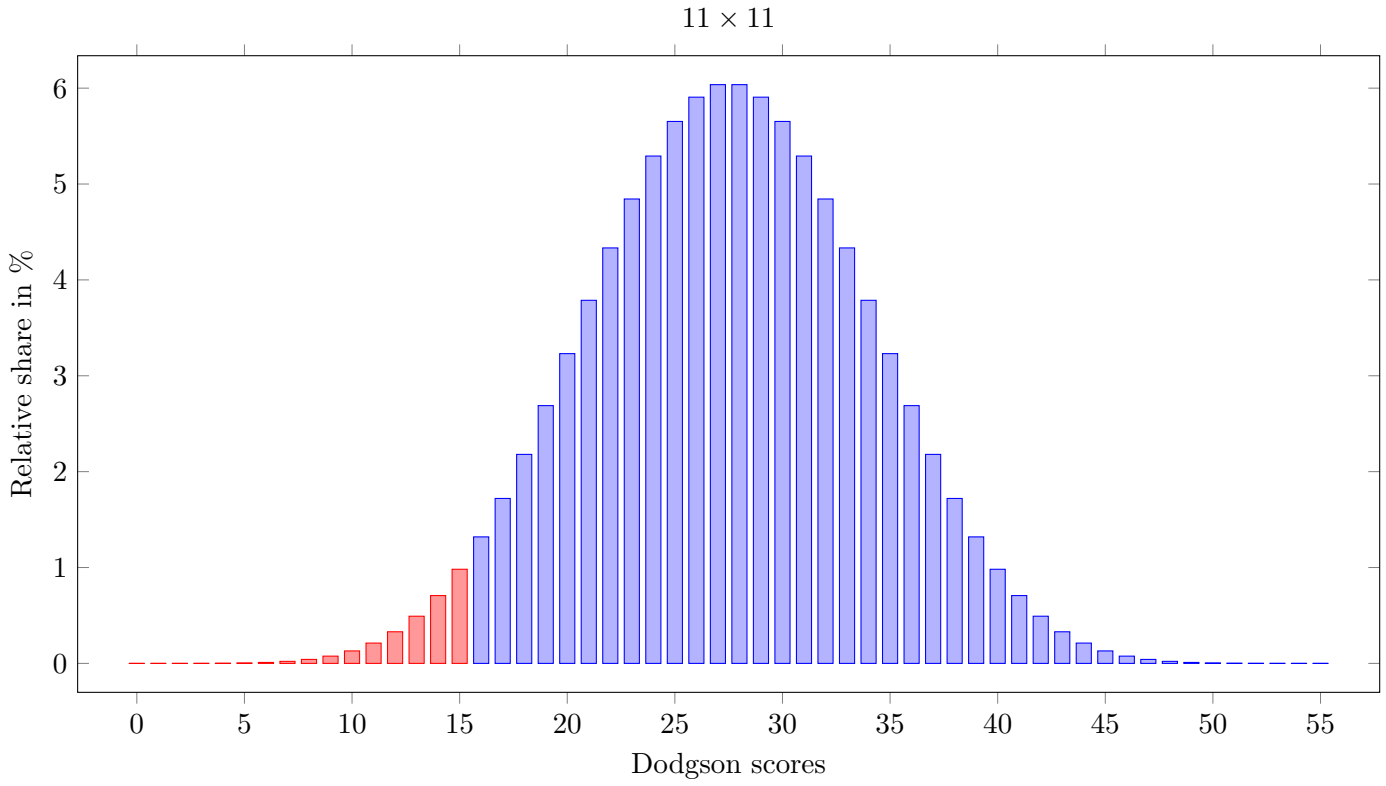

2 × 2
3 × 3
4 × 4
5 × 5
Relative share in %
50
33.33
16.67
25
20.83
12.5
4.17
18.33
16.67
12.5
7.5
3.33
0.83
0 1
0 1 2 3
0 2 4 6
0 5 10
Dodgson scores

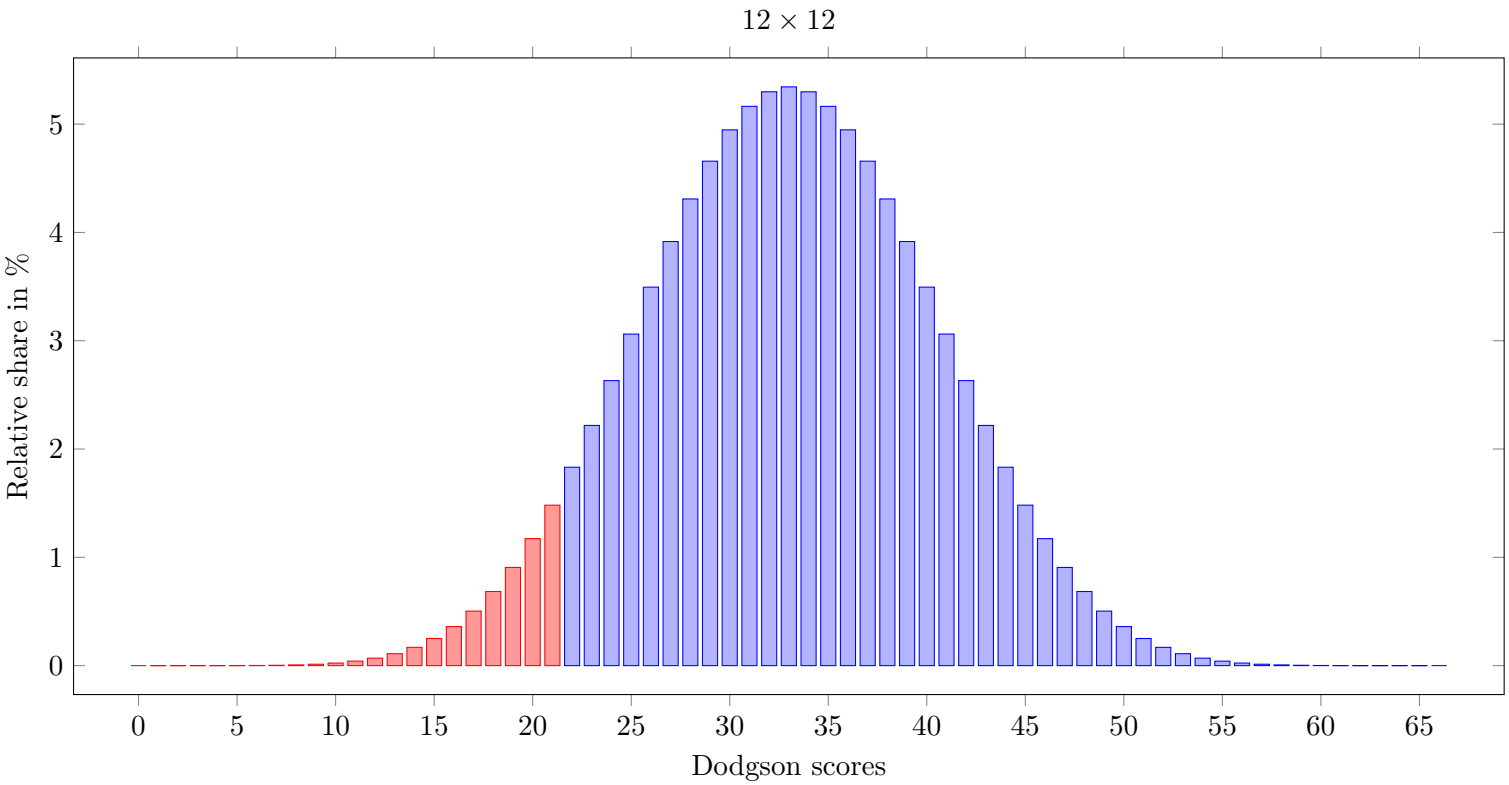

6 × 6
7 × 7
Relative share in %
14.03
12.5
9.86
6.81
4.03
1.94
0.69
0.14
10
8
6
4
2
0
0 5 10 15
0 5 10 15 20
Dodgson scores


8 × 8
Relative share in %
0
2
4
6
8
0
5
10
15
20
25
Dodgson scores


9 × 9
Relative share in %
0
1
2
3
4
5
6
7
8
0
5
10
15
20
25
30
35
Dodgson scores


10 × 10
Relative share in %
0
1
2
3
4
5
6
7
0
5
10
15
20
25
30
35
40
45
Dodgson scores


11 × 11
Relative share in %
0
1
2
3
4
5
6
0
5
10
15
20
25
30
35
40
45
50
55
Dodgson scores


12 × 12
Relative share in %
0
1
2
3
4
5
0
5
10
15
20
25
30
35
40
45
50
55
60
65
Dodgson scores

9. Result

When inspecting and summarizing the numbers, a couple of our predictions can be confirmed, and some also can be refuted.

9.1 Sequential Algorithm Results

Theoretically, the amount of saved time should be tremendous with the scorers using more sophisticated algorithms. However, if we look at the averaged algorithm performances (Table 8.1, and the plots in 8.2 (Average Performance)), the actual gain seems rather small. A speedup by the factor 3 - 17 sounds impressive at first, but considering that the problem size we can solve with that difference in speed is only slightly bigger, it turns out to be underwhelming. In my exemplary maximum range testing (Tables 12.2 and 12.3 in chapter 12 (Appendix)), the Baseline can solve problems up to size 3×22 - 9×5 reliably[1] in under 10 seconds. The best scorer during my basic testing was SC, which could solve problems up to size 3×30 - 13×3. While definitely an improvement, it is obvious that staying in the same order of magnitude results in all advancements being overall not too impressive. But such is the nature of exponentially growing problem sizes.

Most notably, UCS and SC don't beat a simple DFS in the average performance test. ICR does, but by far less then what I expected when analyzing the algorithms on paper. As a common theme, all three advanced algorithms have significantly higher max-values than DFS. And these bad cases occur often enough to cripple the overall outcome.

Reason for the unexpectedly low performance then could be

- As already stated, while UCS and Smart cache are as fast as or faster than DFS in most cases, their average performance is dragged down by extremely time-consuming bad-cases, as reflected by their high standard deviation.
- A best case for them, on the other hand, is incredibly rare. Only one single alternative from the whole profile needs to lean towards a higher Dodgson score, and the whole process gets several times slower, since the complexity grows exponentially

[1] Reliably means, that m is at least 3, since profiles with m being 1 or 2 will always contain a Condorcet winners, so solving them is trivial. Having a considerably higher test count would help solidify the exact numbers, but as I just use them to illustrate the generally possible range, the current amount of tests has to suffice.

with the highest score of the preference profile. If a good case occurs however, it is solved very quickly, as can be seen by their low minima.

- In UCS, even if the amount of explored leaves (as that is what we count) is considerably low, the hidden overhead are all the paths which lead to them, which can be quite big, since we use a breadth-first search. In any tree with depth n and branching factor m, the number of total nodes, is $\sum_{i=1}^{n} m^i$. So, if we have m^n leafs, we can have as many as $\sum_{i=1}^{n-1} m^i$ hidden nodes. These were not present in DFS, since in that approach only changes in the actual profile were explored. The overhead for Smart cache and ICR might be even bigger, as after each layer the intermediate swap-profiles need to be re-generated.

- The overhead for using swap-profiles could impair performance more than I initially assumed. Evaluation becomes more time consuming, since we cannot simply pass our node-data to the Condorcet condition checker. The necessary transformation is one thing DFS can skip, as it directly handles preference profiles as node data.

Comparing loop iteration counts, it turns out that UCS needs on average ~1.6 times longer per checked node than DFS, Smart cache on the other hand only 0.7 times and ICR even 0.16 times. The size of core function calls is vastly different for all of them though. UCS only has little more than the Baseline and DFS, but as SC and ICR use intermediate profiles their iteration count is considerably higher. In order to build one permutation with n agents, they need a total of $n-1$ intermediate profiles. Since many intermediate profiles are shared among similar profiles, the final amount of checked instances is not as high as $n-1$ times the expected amount, but it apparently goes quite high none the less.

9.2 Multi-processing Algorithm Results

In order to get a substantial increase in the problem size we can tackle, we need a high-level change. Weakening the winning condition to only searching the Dodgson winner and having multiple threads search simultaneously lowers the worst case to what was our best case before. The numbers reflect that – the scorers which employ those techniques are around 1000 to 4000 times faster than the Baseline in the average performance tests! And this discrepancy is bound to grow with the problem size. The maximum range of solvable problems reflects this: Compared to the Baseline with 3×22 - 9×5, the best multithreading scorer can solve preference profiles up to a complexity of 3×172 – 92×4.

The reason of how the speedup is achieved is obvious. We do not change anything on the algorithms themselves; we just dramatically reduce the amount of necessary loop iterations to find a solution. Uniform cost search for example reduces its average loop-cycle count from 17540 to 225, a decrease by the factor ~78! And as these 225 function calls were not performed sequentially but by multiple parallel processes, the final speedup factor is with ~185 even bigger.

Wrapping the results up, the three complex algorithms performed below the expectations in the normal rule, but showed great results in the multi-processing approaches.

10. Conclusion

The initial goal was to provide algorithms that give an idea of how complex problems can be solved with Dodgson's rule. If the obtained problem size was big enough, the algorithms developed in the course of this thesis could then be used to help in the development of rules which approximate the Dodgson score. I think it is safe to claim that both goals were reached. Especially if only the Dodgson winner is sought, the threaded implementations can offer a wide range of possible problem sizes which can still be solved in reasonable time. The last project version which was used for the benchmarking done in this thesis is uploaded to sourceforge.net, and can be reached and downloaded at `https://sourceforge.net/projects/dodgsonscoring/`.

Conclusion

Apart from fulfilling the goals, it was also managed to effectively reduce the problem size. While I cannot offer a full mathematical analysis, it would be interesting to check if the reduction also led to a lower complexity class of the whole problem. It is important to say that it can only be achieved with the constraint of seeking the Dodgson winner and not the Dodgson score for any other alternative[1]. But even then the answer would be interesting.

Ultimately, the time needed for finding the Dodgson winner might still be too high. However, as stated in section 1.4 (The Frame of Dodgson's Rule), approximations to the Dodgson score are justifiable, so difficult problems can be solved by algorithms which are tractable. And since computation can be stopped at any point, we can extract the maximal score which was tested and conclude that no solution with a lower score exists. Never being empty handed might be especially interesting for multi-agent systems, which could then decide to throw a coin or let someone else decide, since no obvious solution could be reached by casting a vote.

10.1 Future Work

Future work would include the already mentioned analysis of the complexity of the currently best algorithms. Since the code is open source and is supposed to encourage

[1]This in a way contradicts Caragiannis (2010), which implies that finding the Dodgson score of one alternative is harder than finding the Dodgson winner. This is not true, as finding the Dodgson winner is as hard as finding the Dodgson score of the alternative with the lowest Dodgson score.

own implementations, new ideas can also be explored with relative ease. Finally, an analysis of the probability of good and bad scenarios is also important. In the chapters 4 (Uniform-Cost Search) and 5 (Smart Cache), I tried to investigate the matter a bit and offered part of the solution: the probability of Condorcet winners (Gehrlein, 1999) and counts for a number of small model cases. However, a complete analysis of distributions for variable problem sizes was not covered. The solutions reached there would give a better account of what problem sizes are actually solvable than any amount of benchmarking ever could. Finally, a re-evaluation of Dodgson's rule, in light of the changes in practical use initiated by this project, together with the claim of validity of approximations would be of interest, as its usefulness as a voting rule was criticized (Brandt, 2009).

11. Usage

11.1 Extension

The project is written in C++11, so a compiler who supports it is needed to build it. I compiled it with MVC++ and did not test any other compiler for compatibility, so in the worst case MVC++ needs to be used. The class `Util` and all implementations of `DodgsonScorer` can be seen as the library of this project. They are imported and used in the `Menu` class, which offers a usage interface. If another scorer needs to be added to the project, a couple of things need to be done, most of which are documented in the `Menu` class as well:

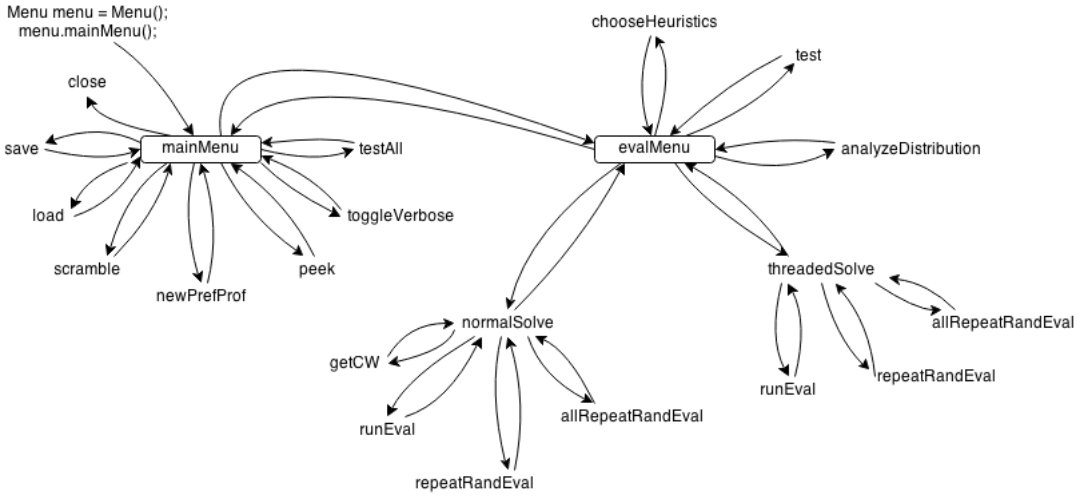
- In `Menu.h`, an enum called `Heuristic` is defined, with currently 6 values. The first 5 are already in use for the current scorers, the 6th, `NEWSCORER`, should be renamed according to the new scorer.
- The constructor of the `Menu` class (from now on everything is located in the file `Menu.cpp`) needs to be advised that a new scorer is introduced. Import the new scorer's header, then add it as the 6th scorer to the scorer-container as indicated by the instruction comment.
- Add the scorer to the `load`-function as indicated.
- Add the scorer to the `newPrefProf`-function as indicated.
- Add the enum to the `chooseHeuristics`-function as indicated.

In case a 7th scorer is added, the instructions just need to be repeated for that scorer, with the difference that in step one, there is no seventh value to be renamed, instead a new one needs to be introduced.

The program is used via the command line. To start it, we need to create an instance of `Menu` by calling one of its constructors, and then call that instance's `mainMenu`-function. The console should then start user interaction over the standard input and output streams. As all core functions require a `preferenceProfile` in order to work, one needs to be supplied. Either directly over the `Menu` constructor (which is what happens in the exemplary `main.cpp` file), or by loading one using option 2, `load profile`, or by creating one using option 4, `create new profile`. The behavior for most algorithms when no preference profile is given is undefined, so this should happen first before

anything else. If a specific profile is supposed to be solved with some scorer, it should be imported with the `load profile` option.

Here is a full graphical representation of the menu, in order to show all of its functionality. Most options are self-explanatory, e.g. `toggle verbose mode`, for others the documentation in their respective header files can be consulted.



11.2 Implementation of a New Scorer

The header file `DodgsonScorer.h` contains the base class for all scorers, and takes together with `DodgsonScorer.cpp` care of all things which are not part of the rule-implementation. Additionally, a couple of useful things are offered which are supposed to make the implementation easier. Following is a wrap-up of all functionalities which should be known if one wishes to implement a new Dodgson scorer.

After including the base class and declaring it the parent, only two things are strictly necessary. The first is the implementation of the function `getSCD(int a)`, which returns the Dodgson score of the alternative 'a'. The second is the implementation of the function `getFastSCD(int a)`, which implements multithreaded search, and has no return value. That functions contains the code which all threads in multi-processed search will run. Both functions are true virtual functions, so the code won't compile unless some implementation is given. But even if a multithreading approach is supposed to be implemented, it makes sense to offer an empty implementation for `fastSCD` first[1] in order to get the simple `getSCD` to work properly before tackling multithreading.

Constructors for the new class will also be necessary, but those can be borrowed with

[1] An empty implementation means, that a function is offered in the following way:
`void NewScorer::getFastSCD(int a) {}`
Additionally, the data-field `m_supportsThreads` should be set to false in the constructor, in order to avoid confusing behavior.

```
NewScorer::NewScorer(/*arguments*/):  DodgsonScorer(/*arguments*/) {}
```

In order to conduct the search, a number of data fields are offered from the base class, which are necessary or may help in some way. A documentation is present in the code, but as a quick reference:

`m_pp` is a pointer to the preference-profile, and the only strictly necessary data field for the search. It is of the type Util::preferenceProfile. [2]

Debugging

`m_scDs` should be used to collect the solutions. It automatically has the size m. Thus, each alternative has at its index a place to store its respective solution. It is of the type `vector<int>`. Its main use lies in printing the solution to the command line when ready, using `DodgsonScorer::print()`. The adding of solutions is taken care of by the base class, so usually there should be no need to access this field directly.

`m_count` should be incremented each time a permutation of the initial profile is created and/or checked. It gives an account of how much of the search tree is traversed, and in comparison to the others scorers can be used to make sure if everything was explored. It is of the type `int`.

Threading

`m_supportsThreads` should be set to 'true' in the constructor if the scorer is supposed to support multithreading. If that was not done, `Menu` functions of the `threadedSolve` submenu will not call the functions properly. It has the type `bool`.

`m_futures` collects all running threads, and is handled outside of the `getFastSCD` function. Usually, there should be no need to ever do something with it. It is of the type `vector<future<void>>`.

`m_minimum` should be used if the approach tries to work with a shared lower bound in order to find an early solution and halt all worse threads. This value should then also be checked first thing in the `getFastSCD` code, ensuring that the current score doesn't exceed the global minimum established by this variable. Else, the current thread can exit. It is of the type `atomic<int>`.

`m_winningPairs` offers a possibility to collect solutions from the threads. Since `std::future` was chosen as thread implementation, this is the easiest way to collect 'return' values from the threads. It is of the type `vector<pair<int,int>>`, where the first `int` is the alternative index and the second `int` the associated Dodgson score. This field is not very crucial at the moment and serves mainly debugging purposes.

`m_mtx` is a mutex to control access to the shared fields `m_minimum` and `m_winningPairs`. It should be locked before accessing them for write operations.

[2] See 11.3 for information on `Util::preferenceProfile`

Special

`m_abort` can be set to 'true' in order to more or less immediately halt all computations. It needs to be reset to 'false' if computations are resumed. This variable should be handled with care, the only use for it at the moment is to stop after a certain time in the benchmarking algorithms is exceeded, as letting some problems run their merry way can take hours or days otherwise.

11.3 Useful Functions and Structures

All functions in `Util.h` should be at least looked at, as they are intended to take care of the problems one is most likely to run in while searching for Dodgson scores. Most important are:

`Util::preferenceProfile` is a 'struct' which contains a two dimensional vector and two `int`s n and m which represent a preference profile's data and its size in $n \times m$. It is always used internally as preference profile representation, so in most cases, there is no need to know exactly what happens inside. The data can be accessed via `preferenceProfile::data`, and a new one can be constructed by submitting the dimensions to the constructor. Most importantly, if `getSCD` only uses two dimensional vectors to represent data, these can also be used to check for the Condorcet criterion, which is the second most important part of the `Util` class. All testing algorithms scramble a field of this type for randomized runs.

`IsCW()` takes either a `Util::preferenceProfile` and an `int`, or a two dimensional vector and an `int`. In both cases, the first argument represents the preference profile, and the `int` the alternative which is supposed to be checked. This function returns 'true' if the submitted alternative is the Condorcet winner for the profile.

12. Appendix

12.1 Tables

n	m	$prob(CW)$
3	3	94.4%
3	7	76.1%
3	11	65.4%
3	15	58.3%
3	19	53.1%
3	23	49.2%
9	3	92.2%
9	7	67.0%
9	11	53.1%
9	15	44.6%
9	19	38.6%
9	23	34.3%
25	3	91.6%
25	7	64.4%
25	11	49.9%
25	15	40.9%
25	19	34.9%
25	23	29.6%
∞	3	91.2%
∞	7	63.1%
∞	11	48.1%
∞	15	39.1%
∞	19	33.1%
∞	23	28.8%

Table 12.1: Probability of a Condorcet winner

time window	n	max m				
		Baseline	DFS	UCS	SC	ICR
5	3	20.2	34.2	24.4	25.3	27.6
	4	12.4	18.8	13.8	14.4	15.6
	5	8.8	12.6	11.0	11.2	12.4
	6	6.6	9.2	8.0	7.4	9.6
	7	5.6	7.8	7.6	7.6	8.6
	8	4.8	6.2	6.4	5.6	6.8
	9	3.4	5.2	6.8	6.4	7.2
	10	0.8	5.0	5.0	5.2	5.0
	11	0.0	4.4	5.0	5.4	0.5
	12	0.0	0.3	1.5	4.4	0.1
	13	0.0	0.0	0.8	0.2	0.0
	14	0.0	0.0	0.0	0.1	0.0
10	3	22.0	39.6	28.6	30.2	31.8
	4	13.8	22.2	15.8	15.8	18.0
	5	9.6	14.4	12.2	11.2	14.2
	6	7.4	10.6	8.2	9.2	10.2
	7	6.2	8.2	8.4	8.4	9.2
	8	5.2	6.8	6.4	6.6	7.8
	9	4.8	6.0	6.4	7.2	7.0
	10	2.4	5.0	5.4	5.6	6.2
	11	0.0	4.6	4.8	5.6	2.5
	12	0.0	1.3	3.6	2.9	2.3
	13	0.0	0.0	1.4	2.9	2.1
	14	0.0	0.0	1.0	2.2	1.9
	15	0.0	0.0	0.1	2.2	1.3
	16	0.0	0.0	0.0	0.0	0.8

Table 12.2: Normal algorithm runs

time window: 5								time window: 10							
n	max m			n	max m			n	max m			n	max m		
	UCS	SC	ICR		UCS	SC	ICR		UCS	SC	ICR		UCS	SC	ICR
3	84.40	139.20	134.40	52	0.12	9.40	8.80	3	101.40	168.00	178.80	52	4.00	8.00	6.80
4	31.40	45.80	41.80	53	0.13	9.60	13.40	4	38.60	47.20	50.00	53	6.50	9.20	12.00
5	36.80	63.80	65.00	54	0.10	6.00	9.80	5	44.60	64.60	63.00	54	3.20	11.00	8.80
6	20.80	27.20	25.20	55	0.18	6.40	13.60	6	22.60	29.00	32.60	55	3.53	11.20	9.00
7	27.20	41.00	38.80	56	0.10	3.80	7.40	7	32.80	40.40	42.40	56	2.27	9.80	8.00
8	16.40	22.80	25.00	57	0.15	5.80	13.40	8	17.80	21.40	22.00	57	2.07	10.80	11.40
9	24.80	35.00	37.40	58	0.05	5.00	10.40	9	25.60	35.80	40.00	58	1.80	8.00	8.40
10	14.20	19.00	21.80	59	0.00	4.90	14.40	10	13.80	22.80	20.40	59	2.27	16.20	11.00
11	20.00	27.80	27.60	60	0.00	3.60	9.00	11	21.20	29.60	31.80	60	1.30	8.60	6.80
12	12.40	17.00	16.80	61	0.00	5.40	14.40	12	13.60	17.00	22.60	61	0.10	13.20	10.40
13	18.00	26.20	28.40	62	0.00	4.60	8.60	13	23.80	29.80	25.20	62	0.05	9.00	6.00
14	13.60	15.60	13.40	63	0.00	6.70	13.20	14	13.00	16.60	19.60	63	0.11	15.20	5.90
15	16.80	20.60	23.60	64	0.00	5.20	8.20	15	19.60	25.20	23.80	64	0.08	10.80	4.30
16	9.80	15.00	15.00	65	0.00	5.20	10.40	16	12.20	16.20	15.00	65	0.04	11.20	7.70
17	16.40	17.60	22.80	66	0.00	4.00	8.20	17	14.40	25.80	26.80	66	0.12	10.40	4.00
18	12.20	14.20	15.40	67	0.00	5.70	7.60	18	13.40	15.20	15.60	67	0.08	6.30	6.30
19	13.60	23.80	26.20	68	0.00	5.50	9.20	19	15.40	21.60	23.80	68	0.09	0.80	5.40
20	10.40	15.00	12.80	69	0.00	5.80	7.80	20	12.40	16.40	15.00	69	0.05	1.10	4.50
21	14.60	21.60	19.80	70	0.00	4.40	8.80	21	15.00	22.40	19.80	70	0.05	0.73	5.30
22	11.00	12.20	13.40	71	0.00	5.50	8.80	22	10.60	15.20	14.00	71	0.03	1.27	4.90
23	11.60	16.80	20.80	72	0.00	4.40	7.40	23	13.40	24.00	20.40	72	0.00	1.15	3.30
24	9.40	13.80	13.40	73	0.00	4.70	7.80	24	10.80	12.40	15.60	73	0.00	1.35	4.60
25	10.60	15.00	16.20	74	0.00	3.60	6.60	25	17.60	18.80	15.00	74	0.00	0.75	4.20
26	10.20	9.80	13.00	75	0.00	5.20	7.60	26	11.20	14.20	12.00	75	0.00	1.10	3.40
27	15.00	13.60	19.00	76	0.00	3.00	6.80	27	14.00	16.60	19.00	76	0.00	0.75	4.60
28	8.80	12.80	14.40	77	0.00	5.60	7.80	28	9.80	11.00	11.20	77	0.00	1.00	4.20
29	10.80	18.20	20.00	78	0.00	2.80	7.80	29	12.20	17.20	19.00	78	0.00	0.70	3.60
30	7.60	12.20	13.00	79	0.00	2.80	7.80	30	8.00	12.20	15.20	79	0.00	1.18	4.40
31	11.40	15.20	16.60	80	0.00	1.80	6.40	31	12.00	15.80	19.80	80	0.00	0.75	4.80
32	9.20	10.60	12.60	81	0.00	1.40	7.00	32	8.40	12.60	12.60	81	0.00	1.02	6.60
33	10.40	16.60	19.60	82	0.00	1.60	4.20	33	13.00	19.00	14.00	82	0.00	0.55	3.20
34	7.60	11.60	12.00	83	0.00	2.00	6.20	34	4.30	12.00	12.20	83	0.00	0.73	4.80
35	11.00	15.00	13.80	84	0.00	1.40	3.60	35	6.40	13.40	15.80	84	0.00	0.48	3.80
36	8.60	10.40	12.80	85	0.00	2.20	5.00	36	5.20	11.60	10.80	85	0.00	0.08	4.80
37	11.00	17.40	15.80	86	0.00	1.40	6.00	37	5.90	12.80	17.40	86	0.00	0.08	3.20
38	6.80	13.00	10.80	87	0.00	1.80	7.20	38	5.80	12.40	10.00	87	0.00	0.11	2.20
39	7.20	16.00	12.00	88	0.00	1.40	4.20	39	6.50	10.20	13.20	88	0.00	0.07	2.40
40	8.40	10.00	8.00	89	0.00	1.80	3.60	40	4.40	9.60	11.00	89	0.00	0.05	0.50
41	8.00	12.00	17.60	90	0.00	1.20	3.80	41	4.70	15.60	11.20	90	0.00	0.04	0.55
42	10.00	9.80	8.40	91	0.00	2.00	5.60	42	4.10	7.60	9.80	91	0.00	0.04	0.75
43	8.80	16.80	13.00	92	0.00	2.60	4.20	43	5.40	12.20	9.60	92	0.00	0.08	0.35
44	5.20	10.00	9.40	93	0.00	1.60	2.07	44	5.00	11.40	10.60	93	0.00	0.07	0.50
45	6.00	13.00	14.20	94	0.00	1.40	2.47	45	5.20	17.40	13.40	94	0.00	0.06	0.45
46	4.00	9.80	11.60	95	0.00	1.60	2.00	46	5.30	10.80	9.40	95	0.00	0.06	0.45
47	1.50	14.80	14.20	96	0.00	1.20	2.20	47	3.80	15.80	11.20	96	0.00	0.08	0.50
48	0.95	9.20	10.00	97	0.00	2.40	1.40	48	4.20	11.60	9.00	97	0.00	0.06	0.40
49	0.37	14.00	12.80	98	0.00	2.00	1.60	49	5.50	12.60	12.20	98	0.00	0.08	0.40
50	0.22	9.40	8.20	99	0.00	1.80	1.60	50	3.40	9.60	8.80	99	0.00	0.06	0.45
51	0.20	12.60	12.00	100	0.00	1.80	1.60	51	5.70	11.20	9.20	100	0.00	0.06	0.45

Table 12.3: Multi-processed algorithm runs

12.2 Uniform Cost Search

Basic UCS

The following section contains a full explanation of the basic UCS algorithm and will be explained using pseudo-code resembling C++. The numbers inside the text refer to parts in the code.

First, a structure of some kind needs to be supplied which serves as Node [1]. Each Node has two fields: its specific Node data, and a list of all its weighted edges. These edges direct to the actual child-Nodes, usually by a reference to them.

The heart of the algorithm is the priority queue [2], which not only holds all Nodes, but also dictates the order in which they are explored. It is initated with the root-Node, which is the only parameter the basic search needs in order to be run. After this initialization, the loop begins [3] which will traverse the search tree by expanding Node after Node until it finds a solution. The first task in the loop is always to extract the current minimum from the priority queue, together with the cost it was associated with [4]. After the extraction, the entry can be tossed away. Next, the win-condition is checked for the Node data [5], and if satisfied, the Node's data and its associated cost are returned and the algorithm can be stopped. If the win-condition was not fulfilled, the Node needs to be expanded. This is done by examining the Node's edges [6] and sorting them into the priority queue according to their costs [1]. The cost from each entry in both edge list as well as priority queue is extracted by consulting its 'first' value, and in case of the edge list we also have to add the current cost to it. If all Nodes were explored, the priority queue is empty. We then have failed in finding a solution [7], and actions depending on the problem need to be taken.

This concludes the basic UCS algorithm.

[1] Note, that it is assumed that the edge costs do not sink, which enables us to sort all new Nodes into the queue one after the other. That way only one traversal of the queue is necessary for each new Node expansion, instead of once for each new Node. Depending on the problem this has the potential to save a lot of time.

```
1  // 1
2  Node {
3    List edges<int, Node>;
4    Data d;
5  }
6
7
8  /* This algorithm explores the search tree offered by the root and, *
9   * using winCondition, finds a solution.                            *
10  * Return value is the minimal found cost and the associated data,  *
11  * or 0 if no solution was found.                                   */
12 Basic_UCS (Node root) {
13
14   // 2
15   List priorityQueue<int, Node>;
16   priorityQueue.add( 0, root );
17
18   // 3
19   while(not priorityQueue.empty()) {
20
21     // 4
22     int currentCost, Node current = priorityQueue.smallest();
23     priorityQueue.deleteSmallest();
24
25     // 5
26     if(winCondition( current.d ))
27       return currentCost, current.d;
28
29     // 6
30     int queuePos = 0;
31     for(entry : current.edges) {
32
33       while(priorityQueue[queuePos].first <= (entry.first + currentCost)) {
34         queuePos++;
35       }
36       priorityQueue.insertAt(queuePos, entry);
37
38     }
39   }
40
41   // 7
42   return 0;
43 }
```

Dodgson UCS

The following section contains a full explanation of the Dodgson's rule optimized UCS algorithm and will be explained using pseudo-code resembling C++. The numbers inside the text refer to parts in the code.

Again, we need to invent a structure to represent Nodes [1]. But this time we can skip the edge-list, as the possible children can be generated from the Node's data — the swap history, which will be explained here with an example:

Both the Node's preference profile as well as all children's preference profiles can be constructed using a swap history. First, we will examine how a swap history represents a preference profile of some Node, given the initial preference profile and one of its alternatives as the alternative in question.

v_1	v_2	v_3
a	b	c
b	c	a
c	a	b

Table 12.4: original preference profile

a

Figure 12.1: examinated alternative

v_1	v_2	v_3
a	a	a
b	b	c
c	c	b

Table 12.5: exemplary new preference profile

0	2	1

Table 12.6: corresponding swap history

The swap history represents the new preference profile in so far, as for each agent v_i, the value in the swap history at position i memorizes how many times the examinated alternative was swapped in that agent's preference list. As a consequence, the sum of all values in a swap history is the Dodgson score [2]. Keeping in mind that all children of some Node can be obtained by collecting all copies which are incremented at one place, we can see that we don't need preference profiles for child-generation. The children of the exemplary Node would then be

1	2	1

(a) incremented for v_1

0	3	1

(b) incremented for v_2

0	2	2

(c) incremented for v_3

Table 12.7

These potential children must then be tested for validity, i.e. if they are actually possible permutations of the initial preference profile. In our case none of them are, since the last Node already lead to a being placed on top of all preference lists. We can check this using a position table, which remembers the position of a in the initial profile. The value from any position i reflects how many swaps can maximally be performed on agent v_i's preference list. Doing this check at generation level is crucial, as removing impossible Nodes at later stages is cumbersome. Also, halting a generation early will stop all its impossible children from being generated as well, which will in most cases save a lot of time. When checking for the winning condition, a simple algorithm [3] can transform a swap history back into the corresponding preference profile, using the history, the original profile, the examined alternative, and said position-table as input.

We start the algorithm by storing the maximum number of swap-processes possible per profile, which is just the number of agents [4]. Next, we generate the position-table [5], followed by the initialization of the priority queue [2] [6] with the empty root. Again, the first step in the loop is to collect the data of the current minimal entry, and tossing it afterwards [7]. In all implementations which base the queue on a vector instead, the minimal entry should be located at the end, as only there deletion can be done in constant time. After that, we can check if the current profile has the necessary amount of swaps [8] and if it also fulfills the winning condition [9]. If it does, we can stop computation, as the global condition of minimality is also satisfied due to the generation order followed by this algorithm. If it does not, checking the priority queue's front container and deleting it if it is empty enables us to always check the smallest Nodes with `priorityQueue[0]`, and also makes insertion of new Nodes simpler[10]. If however the Node failed the first condition, we know that the Node still needs to be expanded [11], which is the last step in this algorithm. Failure [12] is actually impossible, since every alternative has a Dodgson score.

[2]The priority queue of UCS for Dodgson scoring works slightly differently from the basic one. Since we know that Dodgson scores tend to clump (data on this can be found at 8.4 (Analysis of the Search Space)), it is a lot faster and more convenient to prepare buckets for the incoming preference-profiles. That way, we don't have to traverse the search space at all for sorting in new profiles. Instead, we traverse the buckets, of which there are $\frac{n \cdot m}{2}$, which is a lot less then m^n, the theoretical upper limit of the search space. By doing that we can initialize our queue with the right amount of buckets straight away.

```
// 1
struct Node {
  List swapHistory<int>;

  Node(Node father, int newSwap) {
    swapHistory = copy( father.swapHistory );
    swapHistory.add( newSwap );
  }

  // 2
  int scD () {
    return sum( swapHistory );
  }
}


// 3
PreferenceProfile transform (List<int> swapHistory, List<int> posTable,
  PreferenceProfile original, int alternative) {
  PreferenceProfile newProfile = copy( original );

  for(int i = 0 .. swapProfile.size) {
    for(int j = 0 .. swapProfile[i]) {
      newProfile[i][posTable[i]-(j)] = newProfile[i][posTable[i]-(j+1)];
      newProfile[i][posTable[i]-(j+1)] = alternative;
    }
  }
  return newProfile;
}


/* This algorithms explores the search space and finds a minimal       *
 * winning preference profile, which is returned. We need the original *
 * profile for that, as well as the alternative we want to get the     *
 * Dodgson score of. Note that we don't need a root node, since we     *
 * dynamically generate the search space.                              */
PreferenceProfile dodgson_UCS (PreferenceProfile original, int alternative){

  // 4
  int maxDepth = original.n;

  // 5
  List<int> posTable;
  for(int i = 0 .. original.n) {
    for(int j = 0 .. original.m) {
      if(original[i][j] == alternative) {
        posTable.add( j );
      }
    }
  }


```

```
54    // 6
55    List priorityQueue<List<Node>>( original.n * original.m / 2 );
56    priorityQueue[0].add( new Node() );
57
58    while(not priorityQueue.empty()) {
59
60      // 7
61      Node current = priorityQueue[0].back();
62      priorityQueue[0].popBack();
63
64      // 8
65      if(current.swapHistory.size() == maxDepth) {
66
67        // 9
68        PreferenceProfile prefProf = transform(current.swapHistory, posTable,
69        original, alternative);
70        if(isCondorcetWinner(alternative, prefProf)){
71          return prefProf;
72        } else {
73
74            //10
75            if(priorityQueue[0].empty())
76                priorityQueue.popFront();
77        }
78      } else {
79
80        // 11
81        int queuePos = 0;
82        for(int i = 0 .. posTable[current.swapHistory.size()]){
83          priorityQueue[queuePos].add( new Node(current, i) );
84          queuePos++;
85        }
86      }
87    }
88
89    // 12
90    return 0;
91  }
```

Bibliography

[1] Brandt, Felix: *Some Remarks on Dodgson's Voting Rule*. online. 2009. doi: 10.1002/malq.200810017.

[2] Caragiannis, Ioannis, Kaklamanis, Christos, Karanikolas, Nikos and Procaccia, Ariel: " Socially Desirable Approximations for Dodgson's Voting Rule". In: *EC '10 Proceedings of the 11th ACM conference on Electronic commerce* (2010), pp. 253-262.

[3] De Donder, Philippe, Le Breton, Michel and Truchon, Michel. "Choosing from a weighted tournament". In: *Mathematical Social Sciences* 40 (2000), pp. 85-109.

[4] Fellows, Michael, Jansen, Bart, Lokshtanov, Daniel, Rosamond, Frances A. and Saurabh, Saket: *Determining the Winner of a Dodgson Election is Hard*. online. 2009. doi: 10.4230/LIPIcs.FSTTCS.2010.459.

[5] Fishburn, Peter C.: "Condorcet Social Choice Functions". In: *SIAM Journal on Applied Mathematics* 33 (1977), pp. 469-489.

[6] Gehrlein, William V.: "Approximating the Probability that a Condorcet Winner Exists". In: *Proceedings of National Decision Sciences Institute* (1999), pp. 626-628.

[7] McCabe-Dansted, John C., Pritchard, Geoffrey and Slinko, Arkadii: "Approximability of Dodgson's Rule". In: *Social Choice and Welfare* 31 (2008), pp. 311-330.

[8] Risse, Mathias: "Why the count de Borda cannot beat the Marquis de Condorcet". In: *Soc Choice Welfare* 25 (2003), pp. 95-113.

[9] Tideman, T. N.: "Independence of Clones as a Criterion for Voting Rules". In: *Social Choice and Welfare*, 4 (1987), pp. 185-206.