arXiv:1507.05875v1 [cs.AI] 21 Jul 2015

Efficient Dodgson-Score Calculation Using Heuristics and Parallel Computing

Arne Recknagel and Tarek R. Besold

[1] Ludwig-Maximilian University Munich, arecknag@uos.de
[2] Institute of Cognitive Science, University of Osnabrück, tarek.besold@uos.de

Abstract. Conflict of interest is the permanent companion of any population of agents (computational or biological). For that reason, the ability to compromise is of paramount importance, making voting a key element of societal mechanisms. One of the voting procedures most often discussed in the literature and, due to its intuitiveness, also conceptually quite appealing is Charles Dodgson's scoring rule, basically using the respective closeness to being a Condorcet winner for evaluating competing alternatives. In this paper, we offer insights on the practical limits of algorithms computing the exact Dodgson scores from a number of votes. While the problem itself is theoretically intractable, this work proposes and analyses five different solutions which try distinct approaches to practically solve the issue in an effective manner. Additionally, three of the discussed procedures can be run in parallel which has the potential of drastically reducing the problem size.

1 Introduction

Voting is a common method for a group of individual agents, possibly having distinct goals, interests, and states of information, to coordinate and make a decision or express an opinion. 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 often serves as a means of evaluation whenever two rules disagree in their results.

One such rule is named after the French mathematician Marquis de Condorcet: When voting on different alternatives, a voter (v_i) is presented with all pairwise combinations of alternatives. For each pair, he then needs to decide which alternative is preferred over the other (resulting in a preference profile pp_{v_i}). If we collect all preferences from all possible voters, we declare that alternative to be the winner of the vote which wins all pairwise comparisons. Thus, if a winner is found by this method, this winner is preferred over any other alternative, and is called a 'Condorcet winner'. Unfortunately, a Condorcet winner does not always exist. Therefore, the notion of 'Condorcet-consistency' was introduced: The rule in question will select the Condorcet winner if it exists, but may also offer an alternative if it does not.

Charles Lutwidge Dodgson [1] proposed a refinement of Condorcet's rule, suggesting to select an alternative x which requires the minimal number of changes between adjacent alternatives in any of the voters' rankings in order for x to become a Condorcet winner (i.e., selecting the x which has minimal "Dodgson score", or sc_D). This approach is Condorcet-consistent: If an alternative 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 it to win — one act of convincing leads to the alternative being placed one position higher than it used to be (i.e., one transposition), etc. We choose the Condorcet winner if it exists and give a ranking if it does not, offering an account of how far an alternative is from being a Condorcet winner. Again, this offers a very intuitive interpretation of what the outcome means.

However, a significant weakness of Dodgson's rule is its computational complexity, as the number of possible swaps which can be made grows exponentially with the number of voters and alternatives. In this paper, the practical capacities of problems which can be solved with a Dodgson's rule implementation are explored[3], the results provide a basis for tests and comparisons of approximation algorithms like the ones proposed in [2,3,4]. Several search heuristics are analyzed and their performance is benchmarked. Some heuristics include traditional approaches to improving search space requirements and/or speed, others are specifically tailored for this problem.

The resulting algorithms and implementations, besides their relevance within computational social choice, are of special interest for multi-agent systems, which often employ voting rules to reach a decision. Therefore, 'voter' and 'agent' (irrespective of whether biological or computational) are used interchangeably.

2 The Baseline Scorer

Finding the Dodgson scores for all alternatives within a preference profile is a 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. In the Baseline — as in all other reported approaches — 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 to 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.

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. Additionally, another constraint is implemented: The search space will largely consist of solutions which even cannot be assessed via the Dodgson score measuring algorithm, as they are permutations with changes irrelevant or detrimental to the alternative in question. Since they cannot possibly be the solution, such preference profiles are filtered out during the search space construction.

The search space is generated and stored together with the Dodgson scores for each profile and a second algorithm checks for each of the profiles if it is a Condorcet winner. Once the whole search space has been evaluated, the solution with the minimal score becomes the final solution.

The size of the search space and the speed of the Condorcet winner finding algorithm are the two parameters relevant for assessing the size requirements and speed

[3] While a number of effective approximation methods exist in theory, to the best of our knowledge to date no broad practical evaluation and/or benchmark reference has become available of either an exact or an approximate computation of Dodgson Scores.

of this algorithm. However, if this algorithm is evaluated in relation to the heuristics-enriched improvements presented later, we only need to look at the search space size, as the Condorcet winner finding algorithm is identical for all approaches.

Searching for all solutions, with n voters and m alternatives, the size of the total space is $\Phi(\text{basic, worst case}) = \sum_{i=1}^{m} i^n$ and $\Phi(\text{basic, best case}) = m!^{\frac{n}{m}} m$.[4]

3 The Depth-First Approach

The search space is treated as 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 resides in needed disc space, as in a DFS every node is treated as a new solution and the winning conditions can be tested locally. Thus, while the Baseline needs memory in the range of $O(n^m)$, requirements for a DFS contain only the tree depth, i.e. $O(n)$.

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 [5], DFS is nearly 10 times faster (see Sect. 7). Still, this is most likely due to a technical particularity of an internal routine of the implementation system, as the overall size of the search space is still the same and, in a DFS, the entire search space has to be traversed in order to assure that a found solution indeed is optimal. Thus, also all calculations considering best/worst case scenarios are identical to the Baseline.

From a practical perspective, the implemented DFS 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. It can be argued that DFS forms an upper bound of sorts on the 'real' baseline for comparing Dodgson scores in this regard. Therefore, later implementations will also be compared to it.

4 The Uniform-Cost Approach

In finding the Dodgson score, obtaining a solution by itself is not sufficient unless it also is a minimal one. A classical DFS approach puts no useful preference over which path to chose and when to backtrack, as it only keeps track of local data. Therefore, a DFS needs to traverse the entire search space as any found solution is useless unless all others have also been seen. A BFS has larger space requirements because it stores more information, which can be exploited if the problem needs to fulfill a 'global' condition. That is the case here: We seek the globally smallest possible solution in the tree representing the search space.

---

[4] If n is divisible by m without a remainder, the given formula holds precisely. Otherwise, the calculation becomes more complex and the formula only serves as an approximation.

[5] [5] contains a more in-depth analysis of average runtimes with different values for n and m. We chose $n = 8$ and $m = 5$ because each run would almost never take more than two minutes, which was necessary to collect big enough sample sizes.

Uniform-cost search (UCS) is based on a breadth-first search (BFS). If the edges of the tree are weighted, a uniform cost search can be performed.

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. Still, 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. For that reason we are interested in reducing the space a preference profile requires.

Another problem is that preference profiles, when thought of in terms of change over time, are inefficient to work with. That is why 'swap profiles' are used here and in the following two implementations: The original profile is only stored once, together with the alternative under examination. 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 (i.e., number of agents) per node instead of the full profile. Also, the sum of all values in the vector is the Dodgson score, simplifying comparisons between nodes and facilitating sorting.

Computational Resources Required

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

1. 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 how that affects runtime. The answer to the first question is identical to the best and worst case profiles and the resulting search space size from Sect. 2. 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.

2. Worst case: For a worst-case alternative x to be the Condorcet winner, since the majority of each solution has to be worse than x, it needs to be at the top for more than half of the voting agents. The base is the position the alternative has in all profiles — index of x, or i_x — and the exponent is the majority of agents, or $\frac{n}{2}+1$, given integer division. Thus, $C(x) = i_x^{\frac{n}{2}+1}$, and therefore: $C(all) = \sum_{i=1}^{m} i^{\frac{n}{2}+1}$.

For comparison, for m ranging from 1 to 10 and $n = 5$, we experimentally obtain following values for the total search space $\Phi(n,m)$, the actually traversed space $C(n,m)$, the factor by which Φ is bigger than C, and $\frac{C}{\Phi}$ in percent.

For any n and m it can be said that in a worst case scenario — ceteris paribus — an algorithm traversing C instead of Φ is about $factor$ times faster.

3. Probability of the worst case: 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 between runs. The reason

n, m	$\Phi(n, m)$	$C(n, m)$	factor	diff
1, 5	1	1	1.0	100.0000%
2, 5	33	9	3.667	27.2727%
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 1. Exploration rate of the overall search space.

is that very few bad cases tend to dominate the whole outcome, for UCS leading to a very high standard deviation in terms of clock ticks (see Sect. 7). This implies that there were few but heavy outliers. Checking the values confirms that the outliers are the runs with unusually high duration. However, 'worst case scenarios' are not as sparse in a real-world setup due to a bias among alternatives: An alternative which is (dis)liked by one is often also (dis)liked by others. This, taken to its extreme of all agents having the exact same preferences, leads to the worst case. Given that the difference between worst and best case is significant, this tendency of realistic problems to shift towards the worst case scenario has to be considered quite unfortunate. Sect. 6 includes a proposition on how to address this.

4. Best case: We again estimate C in relation to Φ. Fulfilling the Condorcet condition is a more complex problem for the best case as the agents' preferences are not interchangeable. As we traverse 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 score $\leq x$ until we are done. The minimal Dodgson score for a best case scenario with n being a multiple of m (which, for simplicities' sake, are the only ones we consider) can be computed as: $sc_d min(x) = \frac{n}{m} \sum_{i=1}^{n/2} i$.

We then sort solutions by their Dodgson score and count how many of them have a smaller or equal and how many a bigger score than our minimal one. As illustration, we give the first 12 examples which compute the space given $n = m$. '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 2. 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 the amount of checks performed by the scorer, we have exact information on how many 'nodes' were explored in the search space tree.

5 The Smart Caching and Iterative Cost Raise Approaches

These two algorithms focus even more on not directly handling the preference profiles but using swap profiles instead. Consequently, we refer to the search space as 'swap space'. Both Smart Caching (SC) as well as Iterative Cost Raise (ICR) are alternative ways to semi-informed searches 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 elements: The initial preference profile with the currently examined alternative, the swap profiles (as opposed to preference profiles) which represent the possible solutions, and a position table used for translating swap profiles back into a preference profile.

The idea behind this algorithm class is that by treating the search space as a swap space, we can more easily sort potential solutions. The new core property is that it is possible to generate swap profiles with a specific Dodgson score, as the score is a 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 distributing a summand partition of the score among the n agents.

This defines the general core of these algorithms: 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 higher Dodgson score, etc. Thus the name for SC: We are caching a certain subspace of all possibilities in order to search a valid solution among them.

ICR is an improvement to SC in the way that DFS is to the baseline. On the surface, ICR behaves like an iterative deepening DFS — with the difference that no real depth first search is employed. 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, meaning no previous work needs to be repeated.

The immediate evaluation offered by ICR has, apart from the reduction in space requirements, another upside. Keeping track of the global minimum means that the the first encountered valid solution stops the summand partition algorithm — there is no need to generate more potential solutions. This leads to a gain in speed as well.[6] [7]

[6] The number of possible preference profiles for the next Dodgson Score is always roughly m times higher. As such, cutting the required time of the final traversal by x cuts the overall time by $x/(m/((m-1)/((m-2)/((m-3)...))))$, which is $\approx x$ for big m. x is given through the probability of finding a solution given n, m, and the currently examined Dodgson Score. The exact calculation is non-trivial, but generally it will grow with n and m.

[7] Anticipating the benchmarking results, ICR outperforms SC significantly in terms of speed. But given a model case with $n = 8, m = 8$, the measured reduction in time was nearly four

The position table is critical to the usefulness of these approaches as it makes sure that no impossible profile is generated. By knowing where the alternative in question is located at all times, generating impossible profiles is halted immediately. As currently a recursive generation algorithm is used, the gain from just one early stop can be immense, pruning all consecutive faulty builds away.

Smartly generating summand partitions of the Dodgson score 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 factorizations 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 in it, we increment the current score and generate the next set, check again, etc. Since we can assert that any profile which satisfies the Condorcet criterion in the current set is also a global minimum, we stop when a solution is found. The concept of the algorithm is presented in pseudo code for ICR:

Algorithm 1: The ICR algorithm

begin Search Code
  `ICRSearch(`$pp = v_{11}v_{12}\dots v_{ij}\dots v_{nm}, a \in \{a_1a_2\dots a_m\}$`)`
  Data: A preference profile and one of the alternatives present in it
  Result: The $DodgsonScore$ of the alternative in question

  sc$_D \leftarrow 0$;
  while *True* **do**
    create: Permutor(sc$_D$, pp, a);
    while *Permutor.hasNext()* **do**
      next $\leftarrow$ Permutor.next() $\times$ pp;
      if `CondorcetWinner(`*next, a*`)` **then**
        return sc_D;
      end
    end
    sc$_D \leftarrow$ sc$_D + 1$;
  end
end
begin Permutator description
  Iterates over all possible upward swaps given a maximum number of swaps (i.e. the current Dodgson score), a preference profile, and the alternative in question. The resulting list of integers will, when applied to the preference profile, produce its corresponding permutation.
end

Abstractly, the search space is no longer conceptualized as a tree being traversed in an ordered fashion, backtracking when necessary. Instead, it now is a layered set, with no connectivity between its elements whatsoever.

The size of the finally generated space depends on when a solution is found. The number of profiles measured against their scores in estimation behaves like a quadratic

times higher than the reduction in function calls. This suggests that, similar to the speedup from the baseline to DFS, inefficient storage access in SC also plays a role in the case of ICR.

function, causing finding an early solution to yield significant payoff. Partly similar to the baseline, caching the search space and searching for a solution is separated into different algorithms. Still, even in a bad case only a fraction of the total space is cached at any point in time, and due to the swap profiles less memory is needed for storing.

While we could stop the searching algorithm as soon as we find the first winner, the current implementation finishes only after checking the complete corresponding layer in order to collect all minimal solutions. This facilitates the comparison of solutions with the other scorers: All use different ways of looking for their winner(s), and while they all offer the same final Dodgson score, debugging is easier if all winning profiles are collected and not just the first one encountered — which can very well be different in several approaches. Additionally, it could also be important in some settings to find all solutions, and it does not dramatically raise the problem.[8]

6 Multi-processing/-threading

Sect. 4 mentioned a weakness of all current implementations in that the most likely cases in a real-world scenario are the most difficult ones to solve. This is counterintuitive, because for humans 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, as resolving a tournament usually requires ranking all alternatives, and not all rules allow to meaningfully rank an individual alternative while ignoring the ranking of 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:

1. Run all alternative searches simultaneously. 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.

2. 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 score and running those first. An easy heuristic is to sum up their distances to the top for all voters, i.e. sort them using a Borda count [6].

3. Do both.

Consider an $i \times i$ worst-case scenario with all preference orderings being $(a_1 < a_2 < \ldots < a_{i-1} < a_i)$ (i.e., a_1 being the most and a_i the least preferred alternative). Applying the heuristic, the scenario becomes a best case: BordaCount(a_1) returns the highest value of all alternatives, and running a_1 first produces an instant solution and halts the computation. The 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. The former best

[8] In fact, the computational resource requirements, in big-O notation, stay the same. They 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.

case is now the (comparatively manageable) worst case; at the cost of getting only the Dodgson winner performance significantly improves.

Computational Resources Required

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 [7]. The worst case, with each alternative being perfectly symmetric to all others, is in comparison to it a lot less likely, as it is a constraint for every single alternative: Only one of them breaking the pattern results in a much more favorable case.

In summary, the total speed solely depends on the easiest to compute alternative. Since alternatives do not exist independent of each other, we cannot find ourselves with only bad alternatives; at worst we can have many mediocre ones. If we cannot compute a solution, we halt after a set time and assert that no alternative is better than the last computed score.

7 Benchmarking & Results

Randomness Impartial culture[9] for all agents is assumed in benchmarking. While this makes most sense for testing, it needs to be considered that the runtime should be expected to differ in a more realistic setup. There, agents most likely will more often than not have similar preferences. For the standard variants of all algorithms, this approaches the worst case. For the threaded variants, it approaches the best case.

The preference profiles used for benchmarking are generated pseudo-randomly. When benchmarking several scorers, the initialization seed for the pseudo-random procedure is saved and re-used to guarantee comparability of the results.

Average Performance The first criterion by which the scorers were tested was average performance. A problem size was picked assuring that rounding errors or performance lows did not influence the outcome too much, whilst still allowing enough data to be collected in a reasonable time.

The results of 1000 runs using the standard algorithms on a 8×5 preference profile are given in Table 3. The unit of measure is clock ticks, with 1 clock tick taking one millisecond on the used system. Best performance is bolded, σ indicates the standard deviation.

The multiprocessing variants of the last three algorithms were also tested on their average performance on the same problem. Again, an 8×5 profile with 1000 runs was chosen, and Baseline and DFS were run as before. The results are given in Table 4.

Maximum Range In order to assess the impact of exponential growth, each algorithm was run with increasingly complex problems. n is constant and m is incremented each

[9] Impartial culture means, that no bias in the agents' votes exists — one agent's voting option b highest gives no information what so ever about any other agent's opinion of b

scorer	*min*	*median*	*max*	*mean*	σ	*avg.calls*
Base	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 3. Average performance of normal algorithms.

scorer	*min*	*median*	*max*	*mean*	σ	*avg.calls*
Base	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 4. Average performance of threaded algorithms.

time a solution is found. When exceeding a preset time window, the final value of m is stored and the process is repeated with $n+1$. The process is stopped for the current scorer if $m \leq 4$. The averaged results for 5 iterations are depicted in Fig. 1.[10]

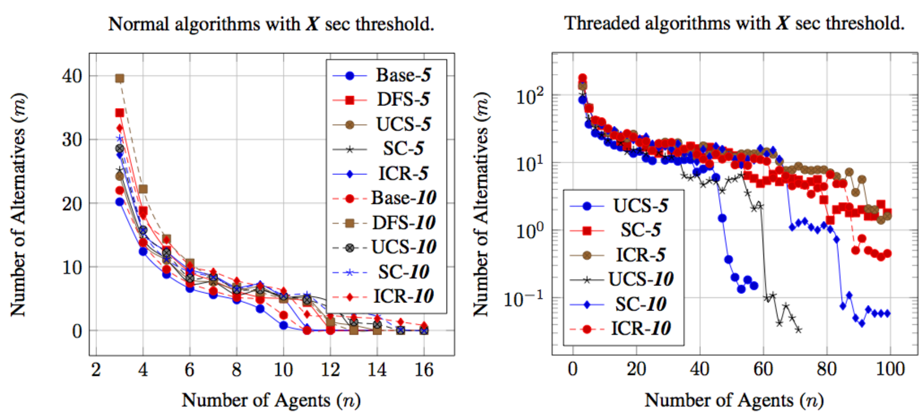

Fig. 1. Experimental range comparison between standard and threaded algorithms.

Sequential Algorithm Results Theoretically, the amount of saved time should be significant with the scorers using more sophisticated algorithms. However, if we look at the averaged algorithm performances (Table 3), the actual gain seems rather small. A speedup by the factor 3 to 17 turns out to be negligible, considering that the gain in size of solvable problems is marginal. In an exemplary maximum range testing, the Baseline can only solve problems up to sizes 3×22 to 9×5 reliably. Most notably, UCS and SC do not beat a simple DFS in the average performance test. ICR does, but by far less than

[10] Similar to [7], even values for n were omitted from the data as the performance of multiprocessing algorithms is heavily influenced by the probability of there being a Condorcet winner. Even numbers of agents make this significantly less likely due to ties, resulting in the graph spiking after every other value.

what could be expected when analyzing the algorithms on paper. All three advanced algorithms have significantly higher max-values than DFS with bad cases occurring often enough to cripple the overall outcome.

Reasons for the unexpectedly low performance could be:
1. While UCS and SC are as fast as or faster than DFS in most cases, their average performance is dragged down by very time-consuming bad cases, as reflected by the high σ.
2. Best cases are rare. If a single alternative from the whole profile leans towards a higher Dodgson score, the whole process slows down since the amount of computational resources required grows exponentially with the highest score of the preference profile. Good cases, however, are solved very quickly as indicated by the low minima.
3. In UCS, even if the amount of explored leaves is considerably low, the hidden overhead are all the paths which lead to them. In any tree with depth n and branching factor m the number of total nodes is $\sum_{i=1}^{n} m^i$, so m^n leafs can have $\sum_{i=1}^{n-1} m^i$ hidden nodes. These were not present in DFS: only changes in the actual profile were explored. The overhead for SC and ICR might be even bigger, as after each layer the intermediate swap profiles need to be re-generated.
4. The overhead for using swap profiles impairs performance more than assumed. Evaluation becomes more time consuming, since we cannot simply pass node data to the Condorcet checker. DFS can skip the necessary transformation as it directly handles preference profiles as node data.

Comparing loop iteration counts, it turns out that UCS needs on average about 1.6 times longer per checked node than DFS, SC on the other hand only 0.7 times and ICR even only 0.16 times. The number of core function calls vastly differs: UCS only has little more than the Baseline and DFS, but as SC and ICR use intermediate profiles the 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 lower than $n-1$ times the expected amount, but still can be quite high.

Multiprocessing Algorithm Results Weakening the winning condition to only searching the Dodgson winner and having multiple threads search simultaneously lowers the worst case to the former best case. Scorers employing those techniques are around 1000 to 4000 times faster than the Baseline in the average performance tests, with this discrepancy being bound to grow with the problem size. The maximum range of solvable problems reflects this: Compared to the Baseline with 3×22 to 9×5, the best multithreading solves preference profiles up to sizes of 3×172 to 92×4. The reason for the speedup is obvious: The algorithms themselves stay untouched, but the number of loop iterations for finding a solution is drastically reduced.

8 Conclusion & Future Work

The initial goal was to provide algorithms giving an idea of how complex problems can be solved with Dodgson's rule. Besides being applicable by themselves, if the obtained problem size was big enough, the algorithms could then be used to help in the development and evaluation of rules approximating the Dodgson score. Both goals were met: Especially if only the Dodgson winner is sought, the threaded implementations offer a wide range of possible problem sizes solvable in reasonable time. Also, we managed to effectively reduce the problem size for the constrained case.[11]

Ultimately, the time needed for finding the Dodgson winner might still be too high. However, approximations to the Dodgson score are justifiable, 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. This might be especially interesting for multi-agent systems, which could then decide to resort to other means of resolving the conflict and taking a decision, since casting a vote yielded no obvious solution.

A starting point for future work is the analysis of the amount of computational resources required by the currently best algorithms. Also, new implementation ideas can be explored since the source codes of the described project version are available for download from `https://sourceforge.net/projects/dodgsonscoring/`. Finally, an analysis of the distribution of good and bad scenarios is important. Sect. 4 and 5 contain initial steps towards an answer: the probability of Condorcet winners [7] and counts for a number of small model cases. However, a complete analysis of distributions for variable problem sizes is lacking. On a conceptual level, 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, countering criticism of its usefulness as a voting rule [8].

References

1. Dodgson, C.: A method of taking votes on more than two issues. In McLean, I., Urken, A., eds.: Classics of social choice, The University of Michigan Press (1995) Originally published 1876.
2. McCabe-Dansted, J., Pritchard, G., Slinko, A.: Approximability of Dodgson's Rule. Social Choice and Welfare **31** (2008) 311–330
3. Caragiannis, I., Kaklamanis, C., Karanikolas, N., Procaccia, A.: Socially Desirable Approximations for Dodgson's Voting Rule. In: EC'10 Proceedings of the 11th ACM Conference on Electronic Commerce. (2010) 253–262
4. Tideman, T.N.: Independence of Clones as a Criterion for Voting Rules. Social Choice and Welfare **4** (1987) 185–206
5. Recknagel, A.: An Approach to Efficiently Calculating Dodgson-Scores Using Heuristics and Parallel Computing. Volume 01-2015 of Publications of the Institute of Cognitive Science (PICS). Institute of Cognitive Science, Osnabrück (2015)
6. Ratliff, T.C.: A comparison of Dodgson's method and the Borda count. Economic Theory **20**(2) (2002) 357–372
7. Gehrlein, W.V.: Approximating the Probability that a Condorcet Winner Exists. In: Proceedings of the National Decision Sciences Institute. (1999) 626–628
8. Brandt, F.: Some Remarks on Dodgson's Voting Rule. Mathematical Logic Quarterly **55**(4) (2009) 460–463

[11] This stands in tension with [3]'s result that finding the Dodgson score of one alternative is harder than finding the Dodgson winner. Trivially, finding the winner is as hard as finding the Dodgson score of the alternative with the lowest one.