Practical Algorithms for STV and Ranked Pairs with Parallel Universes Tiebreaking

JUN WANG, SUJOY SIKDAR, TYLER SHEPHERD, ZHIBING ZHAO, CHUNHENG JIANG, LIRONG XIA

STV and ranked pairs (RP) are two well-studied voting rules for group decision-making. They proceed in multiple rounds, and are affected by how ties are broken in each round. However, the literature is surprisingly vague about how ties should be broken. We propose the first algorithms for computing the set of alternatives that are winners under *some* tiebreaking mechanism under STV and RP, which is also known as parallel-universes tiebreaking (PUT). Unfortunately, *PUT-winners* are NP-complete to compute under STV and RP, and standard search algorithms from AI do not apply. We propose multiple DFS-based algorithms along with pruning strategies and heuristics to prioritize search direction to significantly improve the performance using machine learning. We also propose novel ILP formulations for PUT-winners under STV and RP, respectively. Experiments on synthetic and real-world data show that our algorithms are overall significantly faster than ILP, while there are a few cases where ILP is significantly faster for RP.

1 INTRODUCTION

The *Single Transferable Vote (STV)* rule[1] is among the most popular voting rules used in real-world elections. According to Wikipedia, STV is being used to elect senators in Australia, city councils in San Francisco (CA, USA) and Cambridge (MA, USA), and more [Wikipedia, 2018]. In each round of STV, the lowest preferred alternative is eliminated, in the end leaving only one alternative, the winner, remaining.

This raises the question: *when two or more alternatives are tied for last place, how should we break ties to eliminate an alternative?* The literature provides no clear answer. For example, O'Neill lists many different STV tiebreaking variants [O'Neill, 2011]. While the STV winner is unique and easy to compute for a fixed tiebreaking mechanism, it is NP-complete to compute all winners under *all* tiebreaking mechanisms. This way of defining winners is called parallel-universes tiebreaking (PUT) [Conitzer *et al.*, 2009], and we will therefore call them *PUT-winners* in this paper.

Ties do actually occur in real-world votes under STV. On Preflib data [Mattei and Walsh, 2013], 9.2% of profiles have more than one PUT-winner under STV. There are two main motivations for computing all PUT-winners. First, it is vital in a democracy that the outcome not be decided by an arbitrary or random tiebreaking rule, which will violate the *neutrality* of the system [Brill and Fischer, 2012]. Second, even for the case of a unique PUT-winner, it is important to prove that the winner is unique despite ambiguity in tiebreaking. In an election, we would prefer the results to be transparent about who all the winners could have been.

A similar problem occurs in the Ranked Pairs (RP) rule, which satisfies many desirable axiomatic properties in social choice [Schulze, 2011]. The RP procedure considers every pair of alternatives and builds a ranking by selecting the pairs with largest victory margins. This continues until every pair is evaluated, the winner being the candidate which is ranked above all others by this procedure [Tideman, 1987]. Like in STV, ties can occur, and the order in which pairs are evaluated can result in different winners. Unfortunately, like STV, it is NP-complete to compute all PUT-winners under RP [Brill and Fischer, 2012].

[0]Rensselaer Polytechnic Institute, {wangj38, sikdas, shepht2,zhaoz6,jiangc4}@rpi.edu, xial@cs.rpi.edu

[1]STV for choosing a winner is also known as *instant runoff voting, alternative vote*, or *ranked choice voting*.

Manuscript submitted for review to ACM Economics & Computation 2018 (EC '18).

To the best of our knowledge, no algorithm beyond brute-force search is known for computing PUT-winners under STV and RP. Given its importance as discussed above, the question we address in this paper is: *How can we design efficient, practical algorithms for computing PUT-winners under STV and RP?*

1.1 Our Contributions

Our main contributions are the first practical algorithms to compute the PUT-winners for STV and RP: search-based algorithms and integer linear programming (ILP) formulations.

In our search-based algorithms, the nodes in the search tree represent intermediate steps in the STV and RP procedures, each leaf node is labeled with a single winner, and each root-to-leaf path represents a way to break ties. The goal is to output the union set of winners on the leaves. See Figure 1 and Figure 2 for examples. To improve the efficiency of the algorithms, we develop the following techniques:

Pruning, which maintains a set of *known PUT-winners* during the search procedure and can then prune a branch if expanding a state can never lead to any new PUT-winner.

Machine-learning-based prioritization, which aims at building a large known winner set as soon as possible by prioritizing nodes that minimize the number of steps to discover a new PUT-winner.

Our main conceptual contribution is a new measure called *early discovery*, wherein we time how long it takes to compute a given proportion of all PUT-winners on average. This is particularly important for low stakes and anytime applications, where we want to discover as many PUT-winners as possible with limited resources and at any point during execution. In addition, we design ILP formulations for STV and RP.

The PUT problems are very challenging, mainly due to the exponential growth in the search space as the number of candidates increases (Section 6.5). Yet our algorithms prove practical as experiments on synthetic and real-world data demonstrate. Specifically we show the following in the efficiency of our algorithms in solving the PUT problem for STV and RP, hereby denoted PUT-STV and PUT-RP respectively:

• For both PUT-STV and PUT-RP, in the large majority of cases our DFS-based algorithms are orders of magnitude faster than solving Integer Linear Programming (ILP) formulations in terms of total runtime and time to discover PUT-winners (Section 7).

• For PUT-STV, our devised priority function using machine learning results in significant reduction in time for discovering PUT-winners (Section 6.2).

• For PUT-RP, (i) our proposed pruning conditions exploit the structure of the RP procedure to provide a significant improvement in runtime (Section 6.3), and (ii) our heuristic functions reduce both runtime and discovery time (Section 6.3).

• Most hard profiles have two or more PUT-winners in synthetic datasets, while most real world profiles have single winner. Results show that running time increases with number of PUT-winners (Section 6.4).

2 RELATED WORK AND DISCUSSIONS

A previous version of this paper was presented at EXPLORE-17 workshop [Jiang *et al.*, 2017]. There is a large literature on the computational complexity of winner determination under commonly-studied voting rules. In particular, computing winners of the Kemeny rule has attracted much attention from researchers in AI and theory [Conitzer *et al.*, 2006, Kenyon-Mathieu and Schudy, 2007]. However, STV and ranked pairs have both been overlooked in the literature, despite their

popularity. We are not aware of previous work on practical algorithms for PUT-STV or PUT-RP. A recent work on computing winners of commonly-studied voting rules proved that computing STV is P-complete, but only with a fixed-order tiebreaking mechanism [Csar *et al.*, 2017]. Our paper focuses on finding all PUT-winners under all tiebreaking mechanisms. See [Freeman *et al.*, 2015] for more discussions on tiebreaking mechanisms in social choice.

Standard procedures to AI search problems unfortunately do not apply here. In a standard AI search problem, the goal is to find a path from the root to the goal state in the search space. However, for PUT problems, due to the unknown number of PUT-winners, we do not have a clear predetermined goal state. Other voting rules, such as Coombs and Baldwin, have similarly been found to be NP-complete to compute PUT winners [Mattei *et al.*, 2014]. The techniques we apply in this paper for STV and RP can be extended to these other rules, with slight modification based on details of the rule.

3 PRELIMINARIES

Let $\mathcal{A} = \{a_1, \cdots, a_m\}$ denote a set of m *alternatives* and let $\mathcal{L}(\mathcal{A})$ denote the set of all possible linear orders over $\mathcal{A}$. A *profile* of n voters is a collection $P = (V_i)_{i \le n}$ of votes where for each $i \le n$, $V_i \in \mathcal{L}(\mathcal{A})$. A voting rule takes as input a profile and outputs a non-empty set of winning alternatives.

Single Transferable Vote (STV) proceeds in $m-1$ rounds over alternatives $\mathcal{A}$ as follows. In each round, (1) an alternative with the lowest plurality score is eliminated, and (2) the votes over the remaining alternatives are determined. The last-remaining alternative is declared the winner.

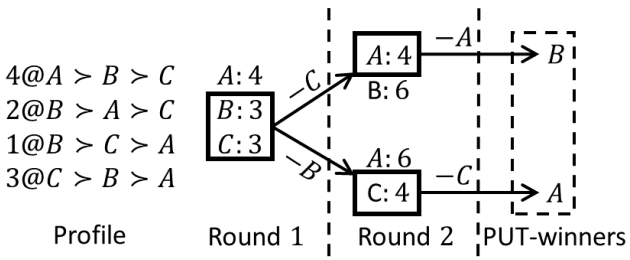


Fig. 1. An example of the STV procedure.

Example 1. Figure 1 shows an example of how the STV procedure can lead to different winners depending on the tiebreaking rule. In round 1, alternatives B and C are tied for last place. For any tiebreaking rule in which C is eliminated, this leads to B being declared the winner. Alternatively, if B were to be eliminated, then A is declared the winner.

Ranked Pairs (RP). For a given profile $P = (V_i)_{i \le n}$, we define the *weighted majority graph* (WMG) of P, denoted by $\text{wmg}(P)$, to be the weighted digraph $(\mathcal{A}, E)$ where the nodes are the alternatives, and for every pair of alternatives $a, b \in \mathcal{A}$, there is an edge (a, b) in E with weight $w_{(a,b)} = |\{V_i : a \succ_{V_i} b\}| - |\{V_i : b \succ_{V_i} a\}|$. We define the *nonnegative WMG* as $\text{wmg}_{\ge 0}(P) = (\mathcal{A}, \{(a, b) \in E : w_{(a,b)} \ge 0\})$. We partition the edges of $\text{wmg}_{\ge 0}(P)$ into tiers $T_1, ..., T_K$ of edges, each with distinct edge weight values, and indexed according to decreasing value. Every edge in a tier T_i has the same weight, and for any pair $i, j \le n$, if $i < j$, then $\forall e_1 \in T_i, e_2 \in T_j, w_{e_1} > w_{e_2}$.

Ranked pairs proceeds in K rounds: Start with an empty graph G whose vertices are $\mathcal{A}$. In each round $i \le K$, consider adding edges $e \in T_i$ to G one by one according to a tiebreaking mechanism,

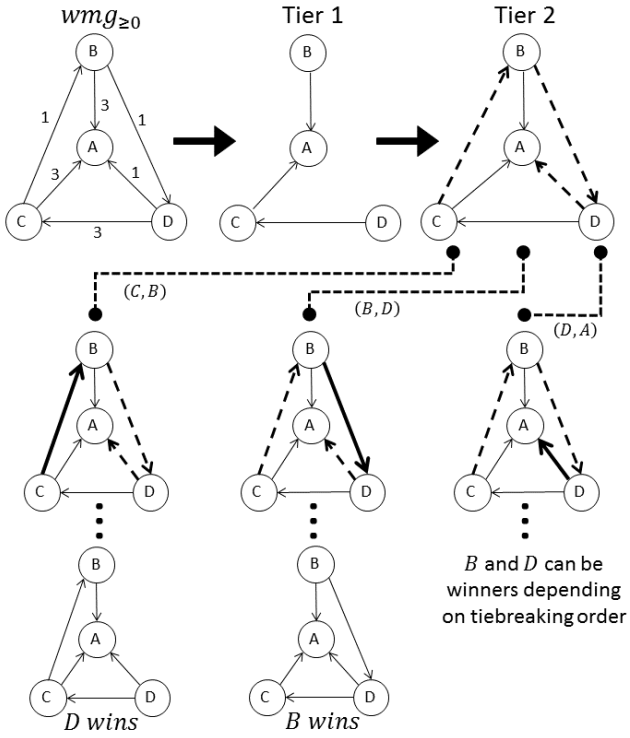


Fig. 2. An example of the RP procedure.

as long as it does not introduce a cycle. Finally, output the ranking corresponding to the topological ordering of G, with the winner being ranked at the top.

Example 2. Figure 2 shows the ranked pairs procedure applied to the WMG resulting from a profile over $m = 4$ alternatives (a profile with such a WMG always exists) [McGarvey, 1953]. We focus on the addition of edges in tier 2, where $\{(C, B), (B, D), (D, A)\}$ are to be added. Note that D is the winner if (C, B) is added first , while B is the winner if (B, D) is added first.

4 ALGORITHMS FOR PUT-STV

We propose Algorithm 1 to compute PUT-STV. For the most part, Algorithm 1 follows a depth-first search (DFS) procedure, except that we include a *pruning* condition whenever all alternatives remaining in the procedure are known to be winners, and the algorithm uses a heuristic *priority* to order exploration of children.

Early Discovery and Heuristic Function. One advantage of Algorithm 1 is its *any-time* property, which means that, if terminated at any time, it can output the known PUT-winners as an approximation to all PUT-winners. Such time constraint is realistic in low-stakes, everyday voting systems such as Pnyx [Brandt and Geist, 2015], and it is desirable that an algorithm outputs as many PUT-winners as early as possible. To measure this, we introduce *early discovery* for PUT-winner algorithms. For any PUT-winner algorithm and any number $0 \leq \alpha \leq 1$, the α-discovery value is the average runtime for the algorithm to compute α fraction of PUT-winners. We note that 100%-discovery value can be much smaller than the total runtime of the algorithm, because the

ALGORITHM 1: PUT-STV(P)

Input: A profile P.
Output: All PUT-STV winners W.
Initialize a stack F with the state $\mathcal{A}$; $W = \emptyset$;
while *F is not empty* **do**
  Pop a state S from F to explore;
  if *S has a single remaining alternative* **then**
    add it to W;
  end
  if *S already visited or $S \subseteq W$ (pruned)* **then**
    skip state;
  end
  For every remaining lowest plurality score alternative c, in order of *priority* add $(S \setminus c)$ to F;
end
return W;

algorithm may continue exploring remaining nodes after 100% discovery to verify that no new PUT-winner exists.

This is why we focus on DFS-based algorithms, as opposed to, for example, BFS—the former reaches leaf nodes faster. To achieve early discovery through Algorithm 1, we prioritize nodes whose state contains more candidate PUT-winners that have not been discovered. In this sense, we design a heuristic function for a state S with known PUT-winners W, $Priority(S) = \sum_{c \in (S-W)} \pi(c)$. Here $\pi(c)$ is the machine learning model probability of c to be a PUT-winner. Details of machine learning setup can be found in Section 6. It is important to note we do *not* use the machine learning model to directly predict PUT-winners. Instead, in the searching process, if we are able to estimate the probability of a branch to have new PUT-winners, we can actively choose which branch to explore first. So under the circumstance without our knowing which branch is promising, machine learning can serve as our guidance to prioritize a better branch with higher probability to find PUT-winners.

5 ALGORITHMS FOR PUT-RANKED PAIRS

At a high level, our algorithm takes a profile P as input and solves PUT-RP using DFS. It is described as the *PUT-RP*(P) procedure in Algorithm 2. Each node has a state (G, E), where E is the set of edges that have not been considered yet and G is a graph whose edges are pairs that have been "locked in" by the RP procedure according to some tiebreaking mechanism. The root node is $(G = (\mathcal{A}, \emptyset), E_0)$, where E_0 is the set of edges in $\text{wmg}_{\geq 0}(P)$. Exploring a node (G, E) at depth t involves finding all maximal ways of adding edges from T_t to G without causing a cycle, which is done by the *MaxChildren*() procedure shown in Algorithm 3. *MaxChildren*() takes a graph G and a set of edges T as input, and follows a DFS-like addition of edges one at a time. Within the algorithm, each node (H, S) at depth d corresponds to the addition of d edges from T to H according to some tiebreaking mechanism. $S \subseteq T$ is the set of edges not considered yet.

Definition 1. *Given a directed acyclic graph $G = (V, E)$, and a set of edges T, a graph $C = (V, E \cup T')$ where $T' \subseteq T$ is a* maximal child *of (G, T) if and only if $\forall e \in T \setminus T'$, adding e to the edges of C creates a cyclic graph.*

Pruning. For a graph G and a tier of edges T, we implement the following conditions to check if we can terminate exploration of a branch of DFS early: (i) If every alternative that is not a known

ALGORITHM 2: PUT-RP(P)

Input: A profile P.
Output: All PUT-RP winners W.
Compute $(\mathcal{A}, E_0) = \text{wmg}_{\geq 0}(P)$;
Initialize a stack F with $((\mathcal{A}, \emptyset), E_0)$ for DFS; $W = \emptyset$;
while *F is not empty* **do**
  Pop a state (G, E) from F to explore;
  if *E is empty or this state can be pruned* **then**
    Add all topologically top vertices of G to W and skip state;
  end
  $T \leftarrow$ highest tier edges of E;
  for *C in MaxChildren*(G, T) **do**
    Add $(C, E \setminus T)$ to F;
  end
end
return W;

ALGORITHM 3: MaxChildren(G, T)

Input: A graph $G = (\mathcal{A}, E)$, and a set of edges T.
Output: Set C of all maximal children of G, T.
Initialize a stack I with (G, T) for DFS; $C = \emptyset$;
while *I is not empty* **do**
  Pop $((\mathcal{A}, E'), S)$ from I;
  if *E' already visited or state can be pruned* **then**
    skip state;
  end
  The successor states are $Q_e = (G_e, S \setminus e)$ for each edge e in S, where graph $G_e = (\mathcal{A}, E' + e)$;
  Discard states where G_e is cyclic;
  if *in all successor states G_e is cyclic* **then**
    We have found a max child; add $(\mathcal{A}, E')$ to C;
  else
    Add states Q_e to I in order of *local priority*;
  end
end
return C;

winner has one or more incoming edges or (ii) If all but one vertices in G have indegree > 0, the remaining alternative is a PUT-winner. For example, in Figure 2, we can prune the right-most branch after having explored the two branches to its left.

Prioritization. To aid in early discovery and faster pruning, we devised and tested three algorithms for heuristic functions. We will use A to refer to the set of candidate PUT-winners (vertices with 0 indegree), and W to refer to the set of known PUT-winners (previously discovered by the search).

- LP $= |A - W|$: Local priority; orders the exploration of children by the value of $|A - W|$, the number of potentially unknown PUT-winners.
- LPout $= \sum_{a \in A-W} \text{outdegree}(a)$: Local priority with outdegree.
- LPML $= \sum_{a \in A-W} \pi(a)$: Local priority with machine learning model π.

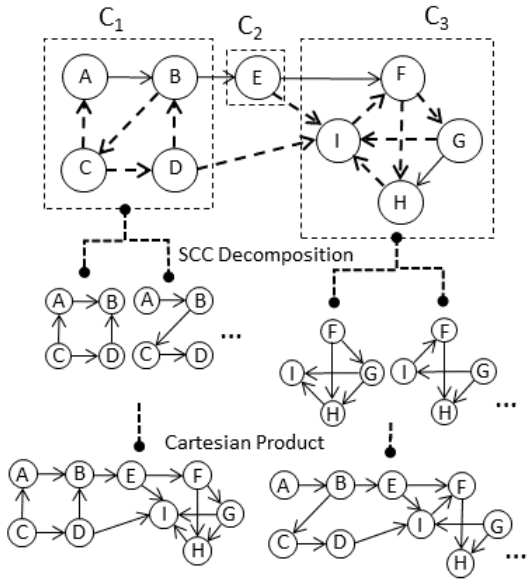


Fig. 3. Example of SCC Decomposition.

SCC Decomposition. We further improve Algorithm 3 by computing strongly connected components (SCCs). For a digraph, an SCC is a maximal subgraph of the digraph where for each ordered pair u, v of vertices, there is a path from u to v . Every edge in an SCC is part of some cycle. The edges not in an SCC, therefore not part of any cycle, are called the bridge edges [Kleinberg and Tardos, 2005, p. 98-99]. Given a graph G and a set of edges T, finding the maximal children will be simpler if we can split it into multiple SCCs. We find the maximal children of each SCC, then combine them in the Cartesian product with the maximal children of every other SCC. Finally, we add the bridge edges. Figure 3 shows an example of SCC Decomposition in which edges in G are solid and edges in T are dashed. Note this is only an example, and does not show all maximal children. In the unfortunate case when there is only one SCC we cannot apply SCC decomposition.

The following Theorem 1 is related to SCC decomposition used in solving the Feedback Arc Set problem, where finding minimal feedback arc sets is very similar to finding our maximal children. The minimal feedback arc set problem is to find, given a directed graph $G = (V, E)$, a minimal subset of edges $E' \subseteq E$ such that $G' = (V, E \setminus E')$ is acyclic. That is, adding any edge $e \in E'$ back to G' creates a cycle [Baharev *et al.*, 2015]. Our maximal children problem has the additional constraint that only edges in the tier T can be removed from the edges of the graph.

Theorem 1. *For any directed graph H, C is a maximal child of H if and only if C contains exactly (i) all bridge edges of H and (ii) the union of the maximal children of all SCCs in H.*

6 EXPERIMENT RESULTS

6.1 Datasets

We use both real-world preference profiles from Preflib and synthetic datasets with m alternatives and n voters to test our algorithms' performance. The synthetic datasets were generated based on impartial culture with n independent and identically distributed rankings uniformly at random over m alternatives for each profile. From the randomly generated profiles, we only test on *hard*

cases where the algorithm encounters a tie that cannot be solved through simple pruning. All the following experiments are completed on an office machine with Intel i5-7400 CPU and 8GB of RAM running Python 3.5.

Synthetic Data. For PUT-STV, we used 50,000 $m = n = 30$ synthetic profiles as our main dataset for the experiments in the paper. For PUT-RP, we generated 40,000 $m = n = 10$ synthetic profiles at random, and picked out 14,875 hard profiles according to our definition in paragraph 1 of Section 6. In the method of SCC(LPML), as we stated in Section 6, we learned a neural network using tenfold cross validation on 10,000 hard profiles, and finally tested our algorithms on another 1,000 hard profiles. We chose the $m = n$ profiles as our dataset for both voting rules, because from Figure 4 we can see that the running time reaches its peak when m and n are close. So in order to obtain relatively good performance on hard cases, we simply chose the $m = n$ profiles for testing.

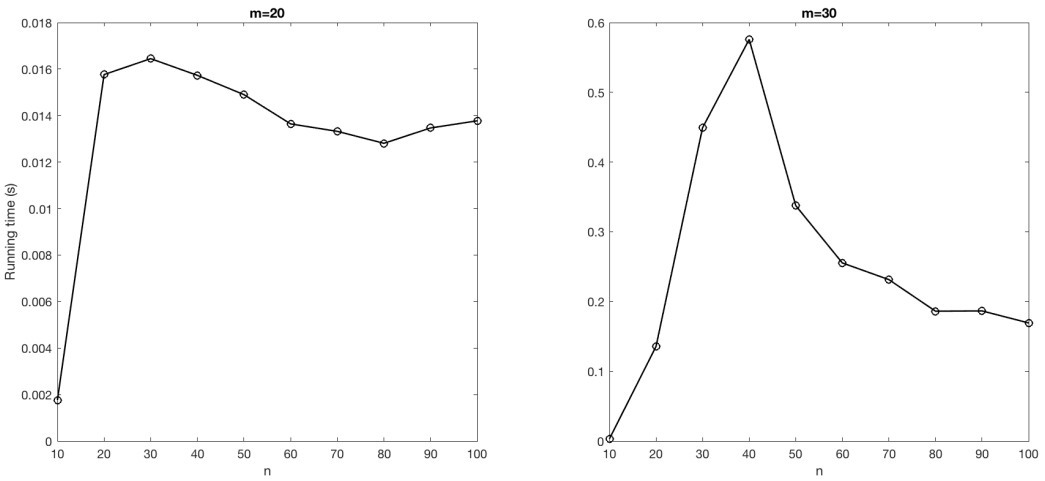


Fig. 4. Running time of DFS for PUT-STV for different number of voters n, for profiles with $m = 20$ and $m = 30$ candidates.

Preflib Data. For the real world data, we use all available datasets on Preflib suitable for our experiments on both rules. Specifically, 315 profiles from Strict Order-Complete Lists (SOC), and 275 profiles from Strict Order-Incomplete Lists (SOI). They represent several real world settings including political elections, movies and sports competitions. For political elections, the number of candidates is often not more than 30. For example, 76.1% of 136 SOI election profiles on Preflib has no more than 10 candidates, and 98.5% have no more than 30 candidates.

6.2 PUT-STV

We have the following observations.

Local Priority with Machine Learning Significantly Improves Early Discovery. As shown in Figure 5, for $m = n = 30$, the algorithm of LPML which delivers a significant 10.39% improvement over

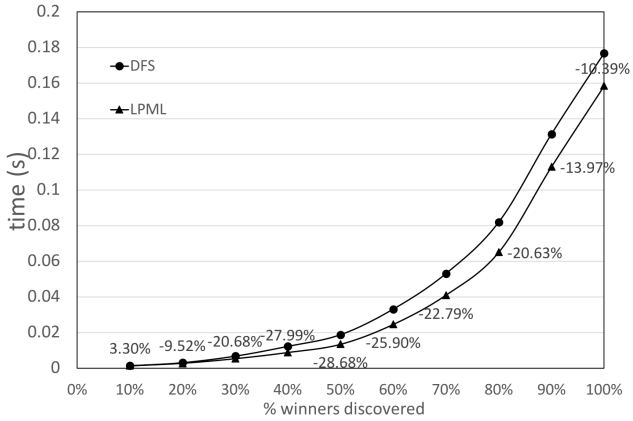


Fig. 5. PUT-STV early discovery.

	DFS	LP	SCC(LP)	SCC(LP+outdeg)	SCC(LPML)
Avg. runtime (s)	22.5045	20.6086	20.4445	21.5017	21.2949
Avg. 100%-discovery time (s)	16.3914	13.8476	13.7339	14.4154	17.0183
Avg. # states	52630.102	47967.451	47955.530	47959.605	48399.779
Avg. # prunes	22434.844	20485.753	20474.685	20476.115	20655.721

Table 1. Experiment results of different algorithms for finding maximal children in ranked pairs.

the baseline of an already optimal, manually designed DFS. [2] Further, the algorithm has 28.68% reduction in 50%-discovery. Results are similar for other datasets with different m. The early discovery figure is computed by averaging the time to compute a given percentage p of PUT-winners. For example, for a profile with 2 PUT-winners which are discovered at time t_1 and t_2, we set the 10%-50% discovery time as t_1 and the 60%-100% discovery time as t_2.

For the machine learning in the local priority function, we learn a neural network model π to predict the m-dimensional vector, where each component indicates whether the corresponding alternative is a PUT-winner. We trained the models on 50,000 $m = n = 30$ hard profiles using tenfold cross validation, with mean squared error 0.086. We also tried other methods like SVC, kernel ridge regression and logistic regression.

Pruning has Small Improvement. When evaluating the performance of pruning in PUT-STV, we see only a small improvement in the running time: on average, pruning brings only 0.33% reduction in running time for $m = n = 10$ profiles, 2.26% for $m = n = 20$ profiles, and 4.51% for $m = n = 30$ profiles.

DFS is Practical on Real-World Data. Our experimental results on Preflib data show that on 315 complete-order real world profiles, the maximum observed running time is only 0.06 seconds and the average is 0.335 ms.

[2]The baseline algorithm is already the best DFS algorithm without using machine learning, and is itself an important contribution of our work.

6.3 PUT-RP

We run different algorithms to find maximal children. Specifically, we evaluate four algorithms that use DFS with different improvements: (i) standard DFS (DFS in Figure 6), (ii) local priority based on # of candidate PUT-winners (LP), (iii) local priority based on total outdegree of candidate PUT-winners (LPout), and (iv) local priority based on machine learning predictions (LPML). We also evaluate the SCC based variants (denoted as SCC(x) where x is the original algorithm). Our experimental results are summarized in Table 1. We observe the following.

Pruning is Vital. Pruning plays a prominent part in the reduction of running time. From Table 1, we see that hitting the early stopping conditions always accounts for a large proportion (about 40%) of the total number of states in the subfunction of finding maximal children. This means our pruning techniques avoid exploring many more states (than the number itself) under the eliminated branches. Our contrasting experiment further justifies this argument: on a dataset of 531 profiles, DFS without pruning takes 125.31 s in both running time and 100%-discovery time on average, while DFS with pruning takes only 2.23 s and 2.18 s respectively with a surprising reduction of 98%.

Local Priority Improves Performance. Our main conclusion is that SCC(LP) is the optimal algorithm for PUT-RP, as we see in Figure 6. LP, i.e. local priority based on number of candidate PUT-winners, significantly reduces both average total running time and average time to discover all PUT-winners compared to standard DFS. SCC-based algorithms SCC(x) always perform slightly better than the corresponding algorithm x, due to the advantage in handling multi-SCC cases. In Figure 6, we compute the time-percentage relation for the 4 algorithms like in PUT-STV and plot the early discovery curves. Specifically, we show the reduction number of discovery time for SCC(LP) compared to DFS. We observe that SCC(LP) has the largest reduction in time; in particular it spends 24.45% less time compared to standard DFS when 50% of PUT-winners are found. LP and SCC(LPout) are slightly worse, whereas SCC(LPML) does not help as much. For LPML, we learn a neural network model π using tenfold cross validation on a dataset of 10,000 $m = n = 10$ profiles, with the objective of minimizing the $L1$-distance between the prediction vector and the target true winner vector. Our mean squared error was 0.0833 on a test set of 1,000 profiles.

Algorithms Perform Well on Real-World Data. Using Preflib data, we find that our optimal algorithm SCC(LP) performs significantly better than standard DFS. We compare the two algorithms on 161 profiles with partial order. For SCC(LP), the average running time and 100% discovery time are 1.33s and 1.26s, which have 46.0% and 47.3% reduction respectively compared to standard DFS. On 307 complete order profiles, the average running time and 100% discovery time of SCC(LP) are both around 0.0179s with only a small reduction of 0.7%, which is due to most profiles being easy cases without ties. In both experiments, we omit profiles with thousands of alternatives but very few votes which cause our machines to run out of memory.

6.4 Distribution of PUT-winners and Running Time

Majority of hard profiles have two or more PUT-winners in synthetic datasets. As we show in Figure 7(a), for PUT-RP with $m = 10, n = 10$, $> 99\%$ of the 14,875 hard synthetic profiles have two or more PUT-winners. Similarly, Figures 8(a), and 8(c) show the histogram of the number of PUT-winners in all synthetic profiles used in our experiments for PUT-STV with $m = 20, n = 20$, and PUT-STV with $m = 30, n = 30$ respectively. We find that greater than two-thirds of the profiles have two or more PUT-winners in these experiments.

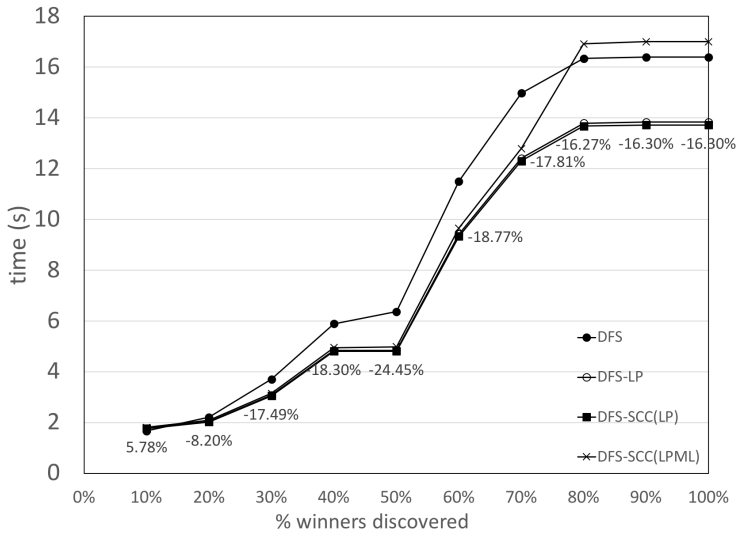
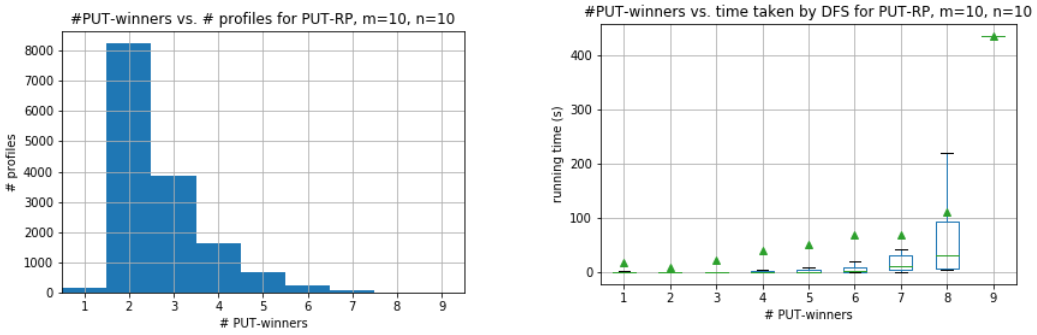


Fig. 6. PUT-RP early discovery.

Most real world profiles have single winner. 90.8% out of 315 SOC profiles have single winner under PUT-STV; 93.2% out of 307 non-timeout SOC profiles, and 89.4% out of 161 non-timeout SOI profiles have single PUT-RP winner.

Running time increases with number of PUT-winners. As we show in Figure 7(b), for PUT-RP with $m = 10, n = 10$, the running time grows with the number of PUT-winners. We make the same observation for PUT-STV (see Figure 8(b) for $m = 20, n = 20$, and Figure 8(d) for $m = 30, n = 30$).



(a) (b)

Fig. 7. Results for 14,875 hard profiles for PUT-RP with $m = 10, n = 10$. (a) shows the histogram of # PUT-winners in hard synthetic datasets for PUT-RP with $m = 10, n = 10$. (b) shows the running time of DFS for PUT-RP vs. # PUT-winners for profiles with $m = 10, n = 10$. Green arrows show the mean. Green horizontal line shows the median (second quartile). Box shows first and third quartiles. Whiskers show minimum and maximum.

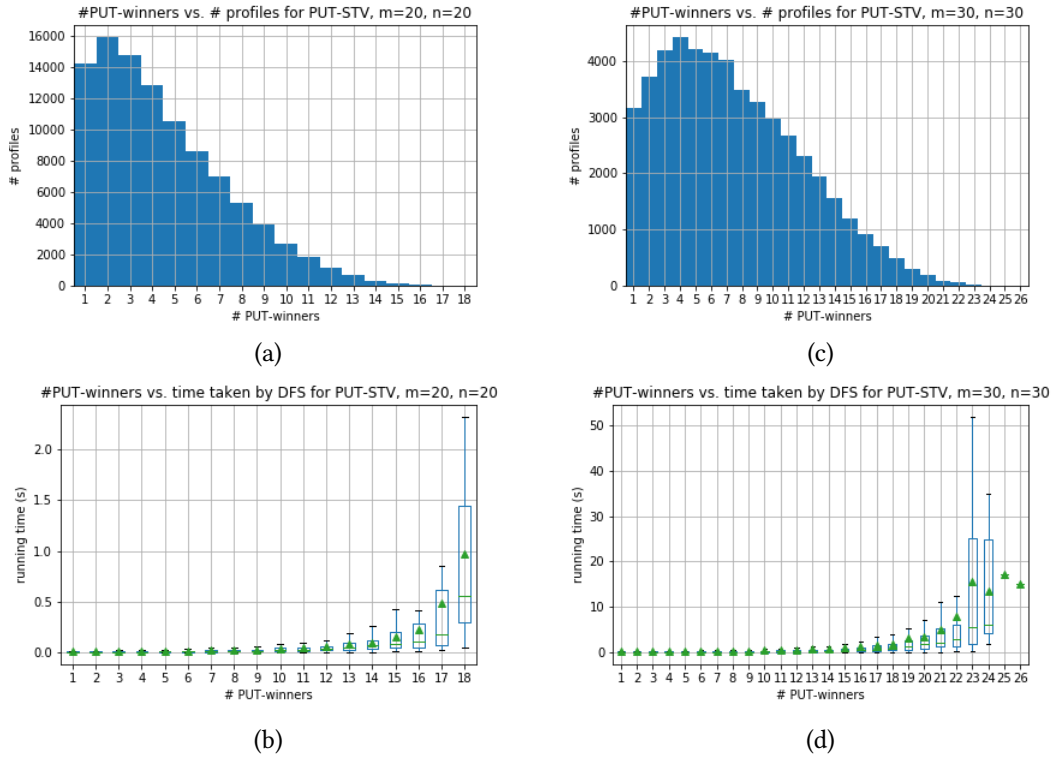


Fig. 8. Results for PUT-STV with 100,000 hard synthetic profiles for $m = 20, n = 20$, and 50,000 hard synthetic profiles for $m = 30, n = 30$. (a) and (c) show the histogram of number of PUT-winners for $m = 20, n = 20$, and $m = 30, n = 30$ respectively. (b) and (d) show the running time of DFS for PUT-STV vs. number of PUT-winners for profiles with $m = 20, n = 20$, and $m = 30, n = 30$ respectively. Green arrows show the mean. Green horizontal line shows the median (second quartile). Box shows first and third quartiles. Whiskers show minimum and maximum values.

6.5 The impact of the size of datasets on the algorithms

The sizes of m and n have different effects on searching space. Our algorithms can deal with larger numbers of voters (n) without any problem. In fact, increasing n reduces the likelihood of ties, which makes the computation easier.

But for larger m, the issue of memory constraint which comes from using cache to store visited states, becomes crucial. Without using cache, DFS becomes orders of magnitude slower. Our algorithm for PUT-STV with $m > 30$ terminates with memory errors due to the exponential growth in state space, and our algorithm for PUT-RP is in a similar situation. Even with as few as $m = 10$ alternatives, the search space grows large. There are $3^{\binom{m}{2}} = 3^{\frac{m(m-1)}{2}}$ possible states of the graph. For $m = 10$, this is 2.95×10^{21} states. As such, due to memory constraints, currently we are only able to run our algorithms on profiles of size $m = n = 10$ for PUT-RP.

7 INTEGER LINEAR PROGRAMMING

ILP for PUT-STV and Results. The solutions correspond to the elimination of a single alternative in each of $m - 1$ rounds and we test whether a given alternative is the PUT-winner by checking if

there is a feasible solution when we enforce the constraint that the given alternative is not eliminated in any of the rounds. We omit the details due to the space constraint. Table 2 summarizes the experimental results obtained using Gurobi's ILP solver. Clearly, the ILP solver takes far more time than even our most basic search algorithms without improvements.

m	10	20	30
n	10	20	30
# Profiles	1000	2363	21
Avg. Runtime(s)	1.14155	155.1874	12877.2792

Table 2. ILP for STV rule.

ILP for PUT-RP. We develop a novel ILP based on the characterization by Zavist and Tideman (Theorem 2). Let the *induced weight* (IW) between two vertices a and b be the maximum path weight over all paths from a to b in the graph. The path weight is defined as the minimum edge weight of a given path. An edge (u, v) is *consistent* with a ranking R if u is preferred to v by R. G_R is a graph whose vertices are $\mathcal{A}$ and whose edges are exactly every edge in $\text{wmg}_{\geq 0}(P)$ consistent with a ranking R. Thus there is a topological ordering of G_R that is exactly R.

Example 3. In Figure 2, consider the induced weight from D to A in the bottom left graph. There are three distinct paths: $P_1 = \{D \to A\}$, $P_2 = \{D \to C \to A\}$, and $P_3 = \{D \to C \to B \to A\}$. The weight of P_1, or $W(P_1) = 1$, $W(P_2) = 3$ and $W(P_3) = 1$. Thus, $\text{IW}(D, A) = 3$, and note that $\text{IW}(D, A) \geq w_{(A,D)} = -1$.

Theorem 2. *[Zavist and Tideman, 1989] For any profile P and for any strict ranking R, the ranking R is the outcome of the ranked pairs procedure if and only if G_R satisfies the following property for all candidates $i, j \in \mathcal{A}$: $\forall i \succ_R j$, $IW(i, j) \geq w_{(j,i)}$.*

Based on Theorem 2, we provide a novel ILP formulation of the PUT-RP problem. We can test whether a given alternative i^* is a PUT-RP winner if there is a solution subject to the constraint that there is no path from any other alternative to i^*. The variables are: (i) A binary indicator variable $X^t_{i,j}$ of whether there is an $i \to j$ path using locked in edges from $\bigcup T_{i \leq t}$, for each $i, j \leq m, t \leq K$. (ii) A binary indicator variable $Y^t_{i,j,k}$ of whether there is an $i \to k$ path involving node j using locked in edges from tiers $\bigcup T_{i \leq t}$, for each $i, j, k \leq m, t \leq K$.

We can determine all PUT-winners by selecting every alternative $i^* \leq m$, adding the constraint $\sum_{j \leq m, j \neq i^*} X^K_{j,i^*} = 0$, and checking the feasibility with the following constraints:

- To enforce Theorem 2, for every pair $i, j \leq m$, such that $(j, i) \in T_t$, we add the constraint $X^t_{i,j} \geq X^K_{i,j}$.
- In addition, we have constraints to ensure that (i) locked in edges from $\bigcup_{t \leq K} T_t$ induce a total order over $\mathcal{A}$ by enforcing asymmetry and transitivity constraints on $X^K_{i,j}$ variables, and (ii) enforcing that if $X^t_{i,j} = 1$, then $X^{\hat{t} > t}_{i,j} = 1$.
- The constraints to ensure that maximum weight paths are selected are detailed in Figure 9.

$$\left.\begin{array}{l}\forall i,j,k \le m, t \le K,\\ Y^t_{i,j,k} \ge X^t_{i,j} + X^t_{j,k} - 1\\ Y^t_{i,j,k} \le \frac{X^t_{i,j}+X^t_{j,k}}{2}\end{array}\right\} i \to j \to k$$

$$\left.\begin{array}{l}\forall i,k \le m, t \le K,\\ \text{if } (i,k) \in E^{\hat{t}\le t}, X^t_{i,k} \ge X^K_{i,k}\end{array}\right\} (i,k)$$

$$\left.\begin{array}{ll} & \forall j \le m,\\ & X^t_{i,k} \ge Y^t_{i,j,k},\\ \text{if } (i,k) \in T_{\hat{t}>t}, & X^t_{i,k} \le \sum_{j\le m} Y^t_{i,j,k},\\ \text{if } (i,k) \in T_{\hat{t}\le t}, & X^t_{i,k} \le \sum_{j\le m} Y^t_{i,j,k} + X^K_{i,k}\end{array}\right\} i \to k$$

Fig. 9. Maximum weight path constraints.

Results. Out of 1000 hard profiles, the RP ILP ran faster than DFS on 16 profiles. On these 16 profiles, the ILP took only 41.097% of the time of the DFS to compute all PUT-winners on average. However over all 1000 hard profiles, DFS is significantly faster on average: 29.131 times faster. We propose that on profiles where DFS fails to compute all PUT-winners, or for elections with a large number of candidates, we can fall back on the ILP to solve PUT-RP.

8 FUTURE WORK

There are many other strategies we wish to explore. In the local priority method, we implemented multiple priority functions, but none of them are significantly better than the number of potential PUT-winners. So one future work is to find a better priority function to encourage early discovery of new winners. Further machine learning techniques or potentially reinforcement learning could prove useful here. For PUT-RP, we want to specifically test the performance of our SCC-based algorithm on large profiles with many SCCs, since currently our dataset contains a low proportion of multi-SCC profiles. Also, we want to extend our search algorithm to multi-winner voting rules like the Chamberlin–Courant rule, which is known to be NP-hard to compute an optimal committee for general preferences [Procaccia *et al.*, 2007].

REFERENCES

Ali Baharev, Hermann Schichl, Arnold Neumaier, and TOBIAS Achterberg. An exact method for the minimum feedback arc set problem. *University of Vienna*, 10:35–60, 2015.

Felix Brandt and Guillaume Chabinand Christian Geist. Pnyx: A Powerful and User-friendly Tool for Preference Aggregation. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1915–1916, 2015.

Markus Brill and Felix Fischer. The Price of Neutrality for the Ranked Pairs Method. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1299–1305, Toronto, Canada, 2012.

Vincent Conitzer, Andrew Davenport, and Jayant Kalagnanam. Improved bounds for computing Kemeny rankings. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 620–626, Boston, MA, USA, 2006.

Vincent Conitzer, Matthew Rognlie, and Lirong Xia. Preference functions that score rankings and maximum likelihood estimation. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, pages 109–115, Pasadena, CA, USA, 2009.

Theresa Csar, Martin Lackner, Reinhard Pichler, and Emanuel Sallinger. Winner Determination in Huge Elections with MapReduce. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

Rupert Freeman, Markus Brill, and Vincent Conitzer. General Tiebreaking Schemes for Computational Social Choice. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1401–1409, 2015.

Chunheng Jiang, Sujoy Sikdar, Jun Wang, Lirong Xia, and Zhibing Zhao. Practical algorithms for computing stv and other multi-round voting rules. In *EXPLORE-2017: The 4th Workshop on Exploring Beyond the Worst Case in Computational Social Choice*, 2017.

Claire Kenyon-Mathieu and Warren Schudy. How to Rank with Few Errors: A PTAS for Weighted Feedback Arc Set on Tournaments. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, pages 95–103, San Diego, California, USA, 2007.

Jon Kleinberg and Eva Tardos. *Algorithm Design*. Pearson, 2005.

Nicholas Mattei and Toby Walsh. PrefLib: A Library of Preference Data. In *Proceedings of Third International Conference on Algorithmic Decision Theory (ADT 2013)*, Lecture Notes in Artificial Intelligence, 2013.

Nicholas Mattei, Nina Narodytska, and Toby Walsh. How hard is it to control an election by breaking ties? In *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, pages 1067–1068, 2014.

David C. McGarvey. A theorem on the construction of voting paradoxes. *Econometrica*, 21(4):608–610, 1953.

Jeff O'Neill. https://www.opavote.com/methods/single-transferable-vote, 2011.

Ariel D Procaccia, Jeffrey S Rosenschein, and Aviv Zohar. Multi-winner elections: Complexity of manipulation, control and winner-determination. In *IJCAI*, volume 7, pages 1476–1481, 2007.

Markus Schulze. A new monotonic, clone-independent, reversal symmetric, and Condorcet-consistent single-winner election method. *Social Choice and Welfare*, 36(2):267—303, 2011.

T. Nicolaus Tideman. Independence of clones as a criterion for voting rules. *Social Choice and Welfare*, 4(3):185–206, 1987.

Wikipedia. Single transferable vote — Wikipedia, the free encyclopedia, 2018. [Online; accessed 30-Jan-2018].

T. M. Zavist and T. N. Tideman. Complete independence of clones in the ranked pairs rule. *Social Choice and Welfare*, 6(2):167–173, Apr 1989.