Fast Optimal Clearing of Capped-Chain Barter Exchanges

Benjamin Plaut
Carnegie Mellon University
bplaut@andrew.cmu.edu

John P. Dickerson
Carnegie Mellon University
dickerson@cs.cmu.edu

Tuomas Sandholm
Carnegie Mellon University
sandholm@cs.cmu.edu

Abstract

Kidney exchange is a type of barter market where patients exchange willing but incompatible donors. These exchanges are conducted via cycles—where each incompatible patient-donor pair in the cycle both gives and receives a kidney—and chains, which are started by an altruist donor who does not need a kidney in return. Finding the best combination of cycles and chains is hard. The leading algorithms for this optimization problem use either *branch and price*—a combination of branch and bound and column generation—or *constraint generation*. We show a correctness error in the leading prior branch-and-price-based approach [Glorie et al. 2014]. We develop a provably correct fix to it, which also necessarily changes the algorithm's complexity, as well as other improvements to the search algorithm. Next, we compare our solver to the leading constraint-generation-based solver and to the best prior correct branch-and-price solver. We focus on the setting where chains have a length cap. A cap is desirable in practice since if even one edge in the chain fails, the rest of the chain fails: the cap precludes very long chains that are extremely unlikely to execute and instead causes the solution to have more parallel chains and cycles that are more likely to succeed. We work with the UNOS nationwide kidney exchange, which uses a chain cap. Algorithms from our group autonomously make the transplant plans for that exchange. On that real data and demographically-accurate generated data, our new solver scales significantly better than the prior leading approaches.

1 Introduction

Chronic kidney disease is a worldwide problem affecting, at various levels of severity, tens of millions of people at great societal burden (Neuen et al. 2013) and monetary cost (Saran et al. 2015). For those with end-stage kidney failure—of which there are over 100,000 in the US alone[1]—the procurement of a new healthy kidney is a life-saving necessity.

Cadaveric kidneys fulfill only a fraction of the demand for kidneys; indeed, the imbalance in supply and demand is growing. Living donation, where a willing donor with two healthy kidneys gives one organ to a patient with kidney failure, is even more desirable than deceased donation; grafts sourced in this manner generally last twice as long as cadaveric grafts in the recipient's body (HHS/HRSA/HSB/DOT 2011). Finding a feasible living donor is difficult due to medical compatibility and other logistical issues. Toward this end, *kidney exchange* (Rapaport 1986; Roth, Sönmez, and Ünver 2004) is a market where patients with willing but incompatible donors swap their paired donors, thus allowing participants to circumvent these compatibility issues.

In this paper, we address kidney exchange from a computational point of view. Specifically, given a set of incompatible pairs of patients and donors, we are interested in computing the "best" set of feasible organ trades, which take place in cycles or unpaired donor-initiated chains. This problem is both theoretically and empirically hard to solve (Abraham, Blum, and Sandholm 2007). Over the last decade, integer programming-based methods for solving different interpretations of the kidney exchange problem have been developed and then used in fielded exchanges. As kidney exchange matures, holes in the expressiveness and scaling capabilities of the current solvers are found, and improvements are made. We are actively involved in this feedback loop with the United Network for Organ Sharing (UNOS) US nationwide kidney exchange, and draw on that experience here.

The two leading kidney exchange clearing algorithms, due to Glorie et al. (2014) and Anderson et al. (2015b), address the optimization problem from complementary directions. We begin by identifying a bug in the correctness of the former algorithm, and give a provably correct fix that also necessarily changes its runtime complexity. We then incorporate the (corrected) idea of Glorie et al. (2014) into an improved version of the prior best branch-and-price-based solver, based on work by Abraham et al. (2007). On real data from the UNOS exchange and on demographically-accurate generated data, our new solver scales dramatically better than both prior approaches when a finite cap on the length of chains is imposed, as is the case in practice. Already on reasonably-sized instances, our method optimally clears markets that the prior methods cannot solve at all.

2 Preliminaries

Any barter exchange can be represented as a directed graph $G = (V, E)$, such that each participating agent is a vertex and directed edges between vertices represent potential trades from one agent to another. In the kidney exchange case, such a *compatibility graph* can be formed by constructing one vertex for each patient-donor pair in the pool (Roth, Sönmez, and Ünver 2004; 2005a; 2005b). Then, each directed edge e from vertex u to vertex v represents a potential medically-compatible transplant from the donor at u to the patient at v. The donor in u is willing to give her kidney if and only if her paired patient receives a kidney. Some potential transplants are more valuable than others. With this in mind, each edge $e = (u, v)$ is assigned a real-valued weight $w(e)$; we will also use the notation $w(u, v)$.

A cycle c of vertices in the compatibility graph G represents a possible kidney swap, with each vertex in the cycle obtaining the kidney of the previous vertex. In kidney ex-

[1] http://optn.transplant.hrsa.gov

change, cycles of length at most some small constant L are allowed—all transplants in a cycle must be performed simultaneously so that no donor backs out after his patient has received a kidney but before he has donated his kidney. In most fielded kidney exchanges—including at UNOS—only 2- and 3-cycles are allowed (i.e., $L = 3$).

Some donors in kidney exchange enter the pool without a paired patient. These non-directed donors (aka "altruist donors") trigger *chains* that start with that donor donating her kidney to a patient, whose paired donor donates his kidney to another patient, and so on (Montgomery et al. 2006; Roth et al. 2006; Rees et al. 2009). In recent years, chains have surpassed cycles as the primary matching mode in many fielded exchanges. The set of patient-donor pairs P and the set of altruist donors A partition the vertex set V.

Chains can be longer than cycles in practice because it is not necessary to carry out all the transplants in a chain simultaneously. Unlike in cycles, if a donor backs out of a chain after his paired patient receives a kidney, no pair in the remainder of the planned chain is strictly worse off; that is, no donor was "used up" before his or her paired patient receiving a kidney. Yet, within a single planning period, longer chains are generally less likely to execute than shorter chains,[2] and are less desirable in practice. Fielded kidney exchanges typically impose a single-period chain-length cap K to avoid very long chains that are extremely unlikely to execute in practice and instead causes the solution to have more parallel chains and cycles that are more likely to succeed. At UNOS, $K = 4$. Planned chains longer than 4 are very unlikely to execute because the success rate of every individual edge tends to be less than a third (Dickerson, Procaccia, and Sandholm 2013).[3]

Finally, a *matching* M is any collection of disjoint cycles and chains in the graph G. The cycles and chains must be disjoint because no donor can give more than one of her kidneys. Given the set of all legal matchings $\mathcal{M}$, the *clearing problem* is to find a matching M^* that maximizes some utility function $u : \mathcal{M} \to \mathbb{R}$. Common fielded utility functions are cardinality- or weight-based, while ongoing work explores incorporating other dimensions (Chen et al. 2012; Dickerson, Procaccia, and Sandholm 2013; Anderson 2014; Manlove and O'Malley 2014; Dickerson and Sandholm 2015; Glorie et al. 2015). For finite cycle cap $L > 2$ (even without chains), even the maximum cardinality problem is NP-hard (Abraham, Blum, and Sandholm 2007).

In this paper, we build a fast clearing engine to optimally solve the maximum-cardinality and maximum-weighted clearing problems on realistic kidney exchange graphs. The first serious computational approach to solving the kidney exchange problem built a specialized branch-and-price-based (Barnhart et al. 1998) integer program solver (Abraham, Blum, and Sandholm 2007); we discuss that method in Section 3, and build on it. Section 3 also discusses the leading non-branch-and-price-based solver, due to Anderson et al. (2015b); that uses a sophisticated recursive traveling-salesman-inspired constraint generation process. The current fastest branch-and-price-based technique is due to Glorie et al. (2014); we discuss that method, identify a bug in its correctness, and propose and prove the correctness of a fix in Section 4. (We discuss solver implementation details in Appendix C.) Finally, in Section 5, we provide extensive experimental results comparing the original branch-and-price-based solver (Abraham, Blum, and Sandholm 2007), our new solver that incorporates the (now correct) ideas of Glorie et al. (2014) and other improvements, and the leading constraint-generation-based solver (Anderson et al. 2015b). We show on both real data from the UNOS kidney exchange and on demographically-accurate data that our solver scales dramatically better than the prior best solvers for realistic values of K and L; indeed, already on moderately-sized compatibility graphs, our solver provides optimal clearing results while the other solvers provide no solution due to excessive run time.

3 Optimally Clearing Large Barter Markets

In this section, we briefly overview the two leading approaches to solving integer program (IP) models of the kidney exchange clearing problem.[4] Models solved by *branch and price* use one binary decision variable for each legal cycle and chain, while those solved by *constraint generation* use a combination of binary decision variables representing edges and cycles—but not chains. In Section 5, we compare two branch-and-price-based solvers and one constraint-generation-based solver; we define their basic structure here.

Branch and price

Given a set of vertices $V = P \cup A$, the number of cycles of length at most L is $O(|P|^L)$, the number of uncapped chains is exponential in $|P|$ if $A \neq \emptyset$, and the number of capped chains of length at most K is $O(|A||P|^{K-1})$. Let $\mathcal{C}(L, K)$ represent the set of cycles of length at most L and chains of length at most K. With one decision variable per cycle and chain $c \in \mathcal{C}(L, K)$, an integer program *model* cannot even be *written* to main memory—much less solved—for even moderately-sized graphs. Indeed, Abraham et al. (2007) could not write down the full model for instances as small as 1000 patient-donor pairs for $\mathcal{C}(3, 0)$, while Dickerson et al. (2012b) could not write down the full model for instances as small as 256 pairs with just 10 altruists for $\mathcal{C}(3, 4)$. Thus, any solver must maintain at most a reduced model (i.e., subset of columns and rows in the constraint matrix) in memory.

Branch and price is a combination of standard branch and bound with column generation that searches for and proves the optimality of a solution to an integer program while maintaining only a reduced model in memory (Barnhart et

[2]For an overview based on real UNOS data of edge, cycle, and chain failure rates and reasons, see §7 of Dickerson et al. (2013).

[3]At the end of a chain is a donor that has not donated yet, and that donor can be used as an altruist in the next batch match (Rees et al. 2009) (e.g., at UNOS, there are two batches per week). This way chains can be continued from batch to batch, and the chains become long that way. In the US, kidney exchange chains have sometimes grown to be 60 long. Note that this in no way contradicts the motivation for the *within-batch* chain-length cap.

[4]For an in-depth survey of integer programming approaches to the kidney exchange problem, see Mak-Hau (2015).

al. 1998). For kidney exchange, the idea is as follows (Abraham, Blum, and Sandholm 2007). (We will loosely refer to cycles and chains only as cycles, because they are represented as decision variables in the model, and because a chain is equivalent to a cycle with an additional "dummy" zero-weight back-edge to an altruist donor.) First, start with some relatively small number of, or no, "seed" cycle variables in the model, and solve the linear program (LP) relaxation of this reduced model. Next, generate *positive price* cycles—variables that might improve the solution when brought into the model. For the maximum-weight clearing problem, the price of a cycle c is given by $\sum_{(u,v)\in c} w_{(u,v)} - \delta_u$, where δ_u is the dual value of vertex u in the LP.

The pricing problem is to generate one or more positive price cycles to bring into the model, or prove that none exist. While any positive price cycles exist at the current node in the branch and bound search tree, optimality has not been proven for the LP. Solving the pricing problem can be expensive in its own right, as we discuss in Section 4. Once there are no more positive price cycles, if the LP solution is integral, optimality is proved at that node in the search tree. However, if the LP is fractional, branching occurs. Abraham et al. (2007) branched on individual cycles c, creating one subtree that includes c in the final solution and a second subtree that explicitly does not, and recursing in this way. Our solver necessarily uses more complex branching, as described later. These branches are then explored in depth-first order until a provably optimal solution is found.

Constraint generation

Constraint-generation-based approaches to kidney exchange have all variables of the appropriate model in memory from the start, but bring in the constraints of the model incrementally. A basic constraint generation form of the kidney exchange problem uses a decision variable for each edge (i.e., only $O(|V|^2)$ variables) in the compatibility graph and solves a flow problem such that unit flow into a vertex exists if and only if unit flow out of that vertex also exists (Abraham, Blum, and Sandholm 2007). This relaxed form of the full problem with only a polynomial number of constraints will not obey cycle or chain caps, so constraints of that form are added until an optimal solution to the relaxed problem is also feasible with respect to cycle and chain caps.

Anderson et al. (2015b) built the leading constraint-generation-based IP solver for the kidney exchange problem. Their solver builds on the prize-collecting traveling salesperson problem (Balas 1989), where the problem is to visit each city (patient-donor pair) exactly once, but with the additional option to pay some penalty to skip a city. They maintain decision variables for all cycles of length at most L, but build chains in the final solution from decision variables associated with individual edges. Then, an exponential number of constraints is required to prevent the solver from including chains of length greater than K; these are generated incrementally until optimality is proved.

In this paper, we focus on three instantiations of kidney exchange clearing engines: BNP-DFS, the initial branch-and-price-based solver due to Abraham et al. (2007); CG-TSP, the leading constraint-generation-based approach due to Anderson et al. (2015b); and BNP-POLY, a new solver we built that combines the (now corrected by us) methodology of Glorie et al. (2014) with other improvements. The next section discusses this new solver.

4 Efficiently Solving the Pricing Problem

In the branch-and-price approach, solving the pricing problem—that is, finding a positive price cycle or set of cycles, or proving that none exist—is performed at every node in the branch-and-bound search tree. Thus speedups in pricing can result in dramatic overall runtime gains. In this section, we discuss pricing methods for the kidney exchange problem. We show that the current leading pricing algorithm is incorrect, and describe our fix for that problem.

Exponential-time pricing

The first branch-and-price-based IP solver for the kidney exchange problem solved the pricing problem by exhaustively considering all feasible cycles and chains, relative to the current partial solution represented by the search tree (Abraham, Blum, and Sandholm 2007). At each node, an exhaustive depth-first-search (DFS) in the compatibility graph computes the price for all cycles until up to a user-specified maximum number of positive cycles are found, or until the search proves that no positive price cycles exist. That proof of nonexistence necessarily sometimes explores all cycles and chains (of capped length) in G which, as discussed in Section 3, is untenably slow. Indeed, for long chains in pools with many non-directed donors, the pricing problem cripples the BNP-DFS performance, as we show in Section 5.

Polynomial-time pricing

We discuss a recent polynomial-time pricing algorithm due to Glorie et al. (2014), find a problem with it, and then propose a fix and prove its correctness.

Method of Glorie et al. (2014). Glorie et al. (2014) give an algorithm that solves the pricing problem (for many kidney exchange functions) in polynomial time. We show that the method is incorrect, after briefly describing the idea behind it. Fortunately, the *idea*—once corrected—is excellent, as we show experimentally in Section 5.

Glorie et al. (2014) reduce the problem of generating positive price cycles to finding negative weight cycles in a directed graph. They construct a "reduced" graph with the same vertices and edges, but with different weights on the edges. If $e = (u, v)$ is an edge in the original graph with weight w_e, and δ_u is the dual value of vertex u, its weight r_e in the reduced graph is given by $r_e = \delta_u - w_e$. Thus, a cycle is positive price in the original graph if and only if it is a negative cycle in the reduced graph.

The next step is to efficiently find *negative* cycles of length at most L for cycles, or K for chains. We will use parentheses to denote a path, and angular brackets to denote a cycle. For example, $(v_1, v_2 ... v_n)$ is a path from vertex v_1 to v_n, while $\langle v_1, v_2 ... v_n\rangle$ is a cycle containing the above path, plus the edge (v_n, v_1). Glorie et al. note the following: suppose there is a path $(v_1, v_2, \ldots, v_n)$ of reduced weight r_1, and an edge $e = (v_n, v_1)$ with reduced weight r_2. Then if $r_1 + r_2 < 0$, $\langle v_1, v_2, \ldots, v_n \rangle$ is a negative cycle.

Thus, efficiently finding short paths of length at most L or K in the reduced graph also finds positive price cycles in the compatibility graph. Hereafter, we use "short" and "long" to refer to the weight of path, not its edge count. In general, the shortest path in a graph with negative edge weights is undefined due to the ability to repeat a negative weight cycle multiple times in a single path. Since a path in our context is not valid if it reuses edges, the problem is well-defined. Yet, finding the shortest path is NP-hard via reduction from the Hamiltonian path problem: set all edge weights to -1 and ask if the shortest path from a source u to any neighbor v such that $(u, v) \in E$ is of weight $1 - |V|$. However, the pricing procedure need only find *some*—not necessarily the shortest—negative weight cycle or prove nonexistence.

The Bellman-Ford algorithm[5] is ideally suited to this. As Glorie et al. (2014) note, the ith step of Bellman-Ford computes shortest paths using at most i edges; however, some edges in those paths may be reused by way of reusing negative sub-cycles in the path. To prevent confusion between the kidney exchange cycles and these sub-cycles in the reduced graph, we refer to sub-cycles as "loops." In Glorie et al., nothing is done to prevent the creation of loops. Internal loops can be removed to recover a valid path, but this may increase the weight of the path above zero. While pursuing that path, the Bellman-Ford algorithm might have ignored a different path that was less promising at the time, but had no internal loops, and would have ended up being a valid negative chain or cycle. This leads to cases where the algorithm returns no negative cycles even though they exist, as demonstrated in Counterexample 1. This causes the overall branch-and-price algorithm to sometimes fail to find an optimal solution, and instead report a suboptimal one as optimal.

Counterexample 1. *Consider the graph with reduced weights in Figure 1, and let the cycle cap $L = 3$ and chain cap $K = 6$. Vertex a is the only altruist, while vertices $p_1, \ldots, p_8$ are patient-donor pairs. The only valid negative cycle or chain in the above graph is the chain $\langle a, p_5, p_6, p_7, p_8, p_1 \rangle$. Since there are no cycles of length at most $L = 3$, no negative cycles will be found on any run of Bellman-Ford where vertex a is not the source. Thus, we only consider the case where a is the source.*

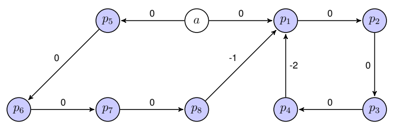


Figure 1: Counterexample to Glorie et al. pricing method.

Let $d(u)$ be the distance from vertex a to vertex u. After four steps, $d(p_4) = 0$ via the path (a, p_1, p_2, p_3), and $d(p_8) = 0$ via (a, p_5, p_6, p_7). On the fifth and final (because $K = 6$) step, $d(p_1)$ updates via p_4 through an internal loop, as $d(p_4) + w(p_4, p_1) = -2 < -1 = d(p_8) + w(p_8, p_1)$. Thus, the path $(a, p_5, p_6, p_7, p_8, p_1)$ is ignored.

At the termination of Bellman-Ford, $d(p_1) = -2$, with path $(a, p_1, p_2, p_3, p_4, p_1)$ stored as its list of predecessors. Because this is negative, Bellman-Ford tries to generate the corresponding negative chain (equivalent to a positive price chain in the compatibility graph) by following its predecessors. After removal of the internal loop at vertex p_1, the chain weight is no longer negative. However, the path $(a, p_5, p_6, p_7, p_8, p_1)$ was ignored in favor of the path to vertex p_1 by way of p_4. That path corresponds to a positive price chain in the compatibility graph but is not returned.

[5]Cormen et al. (2009) overview the Bellman-Ford algorithm.

Corrected polynomial-time pricing. Counterexample 1 breaks the correctness of the solver presented in Glorie et al. (2014), but is amenable to a simple fix: *prevent looping during the Bellman-Ford iterations, not as a post-process afterwards*. To prevent looping, before updating the distance to some vertex v via the edge (u, v), we perform an additional check through the predecessors of u. If v already occurs in the path to u, this would create a loop; if this occurs, we do not update the distance to v.

Assuming $K > L$, the complexity of the algorithm given by Glorie et al. is $O(|V||E|K)$: Bellman-Ford runs from each vertex for K or L steps and examines $O(|E|)$ edges at each step. Our modification adds an extra factor of K, since on each update, we now have to examine up to $O(K)$ predecessors. This yields an overall complexity of $O(|V||E|K^2)$.

Pseudocode for the full method is given in Appendix B, and implementation details are given in Appendix C. Proofs of correctness follow as Theorem 1 (proof in Appendix A) and Theorem 2.

Theorem 1. *Any cycle returned by the algorithm has negative weight (i.e., has a positive price).*

Theorem 2. *If there is a negative cycle in the graph, the algorithm will return at least one negative cycle.*

Proof. We will show that if there is a negative cycle c that we do not find, there must exist a negative cycle with strictly fewer vertices. Thus, for any negative cycle c that we do not return, there must exist a negative cycle p^*q^* with fewer vertices. So, there exists a negative cycle with no negative cycles smaller than it, which our algorithm finds and returns.

Say $c = \langle v_1, v_2, \ldots, v_n \rangle$ is that negative cycle that we do not return. Without loss of generality, assume that c contains the shortest path from v_1 to v_n; if it does not, then that cycle containing the shortest path is also a negative cycle.

Consider running the modified Bellman-Ford method with v_1 as the source. Since by assumption the algorithm does not find c, it must compute a different path from v_1 to v_n than the one in c. We know that the computed path is not shorter, since c contains the shortest path to v_n. Without loss of generality, assume it is strictly longer; were it equal in length, we would be done (as this is a negative cycle that is found by the algorithm as well).

The only way our modified Bellman-Ford method does not compute the shortest path to v_n is if there exists some vertex v_{split}, where $v_{split} \in c$, but the shortest path to v_{split} is not in c. This can occur due to the modification that prevents loops in shortest paths. Let p be the shorter path from v_1 to v_{split}, and let p_c be the path from v_1 to v_{split} in c. Let q be the path from v_{split} to v_n in c, plus the edge (v_n, v_1). This is shown in Figure 2.

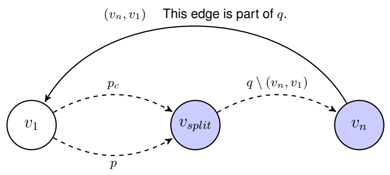


Figure 2: Widget with a negative cycle and existence of a shorter negative cycle. Dotted arrows are paths that contain zero or more vertices (and thus one or more edges).

Then $c = p_c v_{split} q$. Also, since the weights on the paths are $w(p) < w(p_c)$, we have $w(pq) < w(p_c q) = w(c) < 0$.

For any path ρ, let $|\rho|$ represent the number of vertices in that path. We know that $c = p_c q$ satisfies the cycle size cap, since it is valid by assumption.

Claim: $|p| \leq |p_c|$
Proof: By way of contradiction, assume $|p_c| < |p|$. Then, the sequence of updates along p_c will reach v_{split} before p does—which means we will have computed p_c. Even though we may compute p later, we will still be able to make updates with p_c as the base path. Therefore we can go on to compute the full $p_c q$. We will still be able to make updates with p_c as the base path; for more information, see Appendix C.

It is possible that we might encounter this issue again when computing a path to v_n with p_c as the base; in the process of computing q with p_c as the base, there may exist some vertex v'_{split} that causes the same issue as v_{split}. In that case, our logic can be applied recursively until no such vertex like v'_{split} exists. Therefore, $|p| \leq |p_c|$. ■

We can ignore considerations regarding the cycle cap for the rest of the proof, since all cycles discussed will have size at most $|pq| \leq |p_c q| = |c|$, which is legal by assumption.

At this point, we have p and q such that pq is a *circuit* (i.e., a path that starts and ends at the same vertex but which might not be a cycle because it might visit some vertices more than once), and $w(pq) < 0$. We now introduce a tool that we will use to finish the proof of the theorem by using the tool repeatedly.

Claim: In a directed graph, if there exists a circuit ℓ that is not a cycle and $w(\ell) < 0$, then there exists a circuit ℓ' where $w(\ell') < 0$ and $|\ell'| < |\ell|$.
Proof: One can split ℓ into two non-empty paths, α and β, where neither path intersects itself. Because ℓ is a circuit but not a cycle, α and β intersect. Thus there exists $v_\cap \in \alpha$ where $v_\cap \in \beta$. If there are multiple such vertices, let $v_\cap$ be the one occurring earliest in α. Then $\alpha = \alpha_1 v_\cap \alpha_2$ and $\beta = \beta_1 v_\cap \beta_2$, where α_1, α_2, β_1, and β_2 are nonempty. Since $v_\cap$ is the earliest vertex in α that intersects with β, we have that α_1 and β are disjoint; in particular, α_1 and β_2 are disjoint.

We know that α_1 is a path from some start vertex u to $v_\cap$ and that β_2 is a path from $v_\cap$ back to u. Since α_1 and β_2 are disjoint, $\alpha_1\beta_2$ is a cycle, and $|\alpha_1\beta_2| < |\ell|$.

Case I: $w(\alpha_1\beta_2) < 0$. This trivially satisfies the claim.

Case II: $w(\alpha_1\beta_2) \geq 0$. Because $w(\alpha_1\beta_2) \geq 0$ and $w(\ell) < 0$, we must have $w(\alpha_2\beta_1) < 0$. Since α_2 is a path from $v_\cap$ to some vertex $u' \neq u$, and thus β_1 is a path from some vertex u' to $v_\cap$, $\alpha_2\beta_1$ is a circuit such that $w(\alpha_2\beta_1) < 0$. Since neither α_2 nor β_1 contain u, $|\alpha_2\beta_1| < \ell$. ■

We now return to the proof of the theorem. Recall that we have p and q such that pq is a circuit, and $w(pq) < 0$.

By the claim above, the presence of a negative circuit pq implies that either p and q do not intersect, or that there exists a negative circuit $p'q'$ that has fewer vertices.

If p and q were not intersecting, pq would be a shorter path than $p_c q$, which violates the assumption that c contains the shortest path. Thus, p and q do intersect. Therefore, there exists a negative circuit $p'q'$ that has fewer vertices.

Since we can only shrink pq, $p'q'$, and so on in this fashion a finite number of times, there must exist some negative circuit p^*q^* where p^* and q^* do not intersect; so, the negative circuit is a cycle. □

5 Experiments

We experimentally compare our new branch-and-price-based solver BNP-POLY (which has the modified, corrected pricer of Glorie et al. (2014)) against the prior state-of-the-art branch-and-price-based solver, BNP-DFS, due to Abraham et al (2007), and the current state-of-the-art constraint-generation-based solver, CG-TSP, due to Anderson et al. (2015b). On each problem instance, each solver was given access to 28GB of RAM, 4 cores, and 60 minutes of wall time. (Timeouts are counted—conservatively against our solver as will become clear—as 60 minutes toward runtime averages.)

The cap on cycle length was set to 3, as is almost ubiquitous in practice (also at UNOS). We varied the chain cap.

Real UNOS match runs. We first test on real data from the United Network for Organ Sharing (UNOS) nationwide kidney exchange, which now contains 143 transplant centers, that is, 60% of all transplant centers in the US. The 164 match runs on which we test range from October 2010 to November 2014, during which the exchange grew from around 70 patient-donor pairs and altruists to almost 200.

Figure 3 shows mean time to completion for each of the three solvers. All three easily solve instances for chain caps at or below 5; however CG-TSP begins to struggle at chain caps above 5, and even times out on one instance. BNP-DFS remains competitive with BNP-POLY until a chain cap of 9, at which point its exhaustive DFS to solve the pricing problem begins to add substantial runtime cost. BNP-POLY solves all instances extremely quickly.

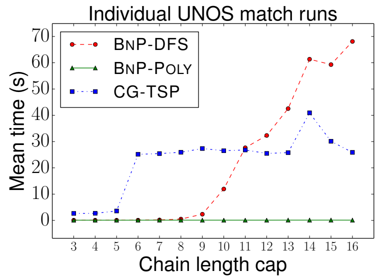


Figure 3: Mean runtime for BNP-DFS, BNP-POLY, and CG-TSP on the first 164 UNOS exchange match runs.

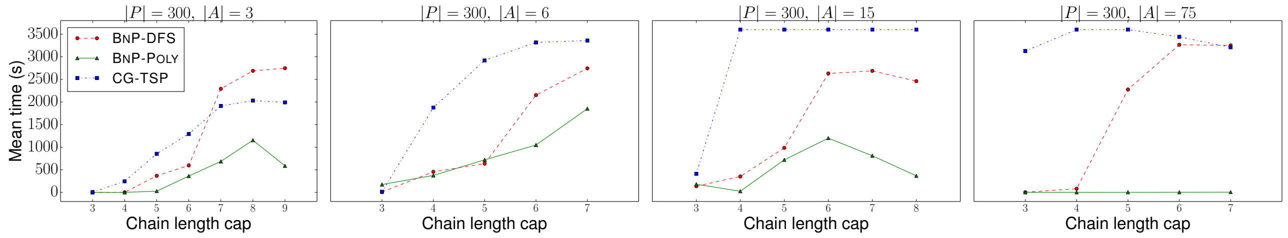


Figure 4: Run time as the number of altruist donors $|A| \in \{3, 6, 15, 75\}$ increases (left to right), for varying finite chain caps.

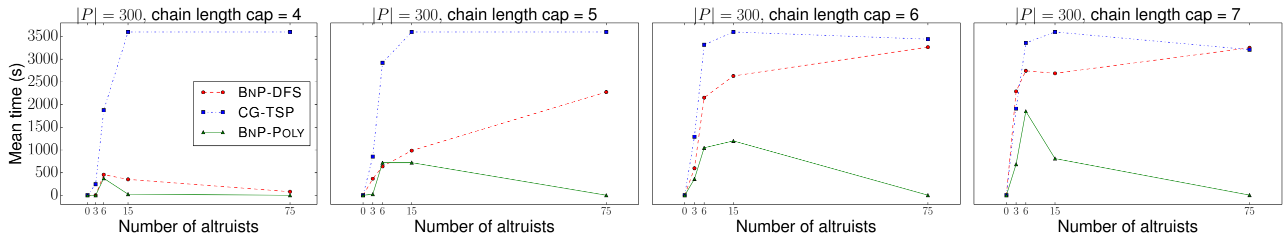


Figure 5: Run time as the chain cap $K \in \{4, 5, 6, 7\}$ increases (left to right), for varying numbers of altruist donors.

Generated UNOS data. At the time of writing, the two largest kidney exchanges—UNOS and the National Kidney Registry[6]—each contain around 300 patient-donor pairs and altruist donors. To test on large numbers of instances with 300 vertices, we generated demographically-accurate problem instances by sampling the set of all pairs and altruists who had entered the UNOS exchange by Nov. 2014. In the following figures, each data point is the average over 20 instances. Each algorithm was run on the same instances.

Figure 4 shows run time for increasing $|A|$ and chain caps. In general, higher chain caps tend to increase problem difficulty for all solvers (although for much smaller graphs we observed an interior hardness peek as a function of chain cap for CG-TSP). BNP-DFS and CG-TSP timed out on instances with just three altruists and a chain cap of 5. BNP-POLY beat both of those prior state-of-the-art solvers (both with respect to timeouts and average runtime).

Figure 5 again shows that BNP-POLY is clearly faster and has fewer timeouts than BNP-DFS and CG-TSP. With an interior number of altruists in the pool, all algorithms take non-negligible time. For very large $|A|$, BNP-POLY solves instances more quickly than for a medium number. We conjecture that this is because good upper bounds are reached quickly in the branch-and-bound tree, since with large $|A|$ the best feasible solution matches all pairs and thus meets the upper bound that is computed without cycle or chain caps.[7] BNP-DFS will have these same bounds, but the exponential-time pricing problem takes substantially longer due to a (potentially necessary) crawl of a large number of chains. With large numbers of altruists, BNP-POLY is fast while the other solvers time out on essentially all instances.

When there is no chain cap, CG-TSP tends to significantly outperform the other solvers. However, as explained earlier in the paper, the very long chains that it generates in that setting typically fail to execute in practice.

[6]`http://www.kidneyregistry.org`

[7]This can be solved in polynomial time using maximum-weighted matching (Abraham, Blum, and Sandholm 2007)

6 Conclusions & Future Research

In this paper, we built a fast clearing engine to optimally solve the maximum-cardinality and maximum-weight kidney exchange problems. First, we identified a bug in the state-of-the-art algorithm, proposed a fix, and proved its correctness. We incorporated this fixed method and other performance improvements into a prior branch-and-price-based integer program solver. Motivated by our experience with the United Network for Organ Sharing (UNOS) kidney exchange which, like other exchanges, uses cycles and chains with finite caps, we then tested our solver against the leading constraint-generation-based solver and a prior state-of-the-art branch-and-price solver. On both real match run data from the UNOS exchange and realistic simulated data, for realistic cycle and chain caps, our solver significantly outperforms both prior state-of-the-art solvers—often optimally clearing instances that the other solvers cannot.

Beyond being able to support the growing practical pools and desired chain caps, faster clearing algorithms enable more expressive, and thus more realistic, models of kidney exchange to be solved—and deployed in practice. Faster static solvers slot into dynamic kidney exchange frameworks, all of which struggle with computational complexity even at very small exchange sizes (Awasthi and Sandholm 2009; Dickerson, Procaccia, and Sandholm 2012a; Dickerson and Sandholm 2015); taking the dynamics of exchange (vertex entrance/departure, edge failure, and so on) into account would result in substantial gains in theory and practice, making this a promising future research direction (Ünver 2010; Blum et al. 2013; Akbarpour, Li, and Gharan 2014; Blum et al. 2015; Anderson et al. 2015a). Adaptations of our solver could also be used to clear exchanges with different logistical constraints, e.g., lung (Ergin, Sönmez, and Ünver 2014; Luo and Tang 2015), liver, and cross-organ exchanges (Dickerson and Sandholm 2014).

7 Acknowledgments

This material was funded by NSF grants IIS-1320620 and IIS-1546752, by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program, and by a Facebook Fellowship. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575; specifically, it used XSEDE computing resources provided by the Pittsburgh Supercomputing Center. We thank Kristiaan Glorie and Ross Anderson for helpful discussion and comments.

References

Abraham, D.; Blum, A.; and Sandholm, T. 2007. Clearing algorithms for barter exchange markets: Enabling nationwide kidney exchanges. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, 295–304.

Akbarpour, M.; Li, S.; and Gharan, S. O. 2014. Dynamic matching market design. In *Proceedings of the ACM Conference on Economics and Computation (EC)*, 355.

Anderson, R.; Ashlagi, I.; Gamarnik, D.; and Kanoria, Y. 2015a. A dynamic model of barter exchange. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

Anderson, R.; Ashlagi, I.; Gamarnik, D.; and Roth, A. E. 2015b. Finding long chains in kidney exchange using the traveling salesman problem. *Proceedings of the National Academy of Sciences* 112(3):663–668.

Anderson, R. 2014. *Stochastic models and data driven simulations for healthcare operations*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Awasthi, P., and Sandholm, T. 2009. Online stochastic optimization in the large: Application to kidney exchange. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, 405–411.

Balas, E. 1989. The prize collecting traveling salesman problem. *Networks* 19(6):621–636.

Barnhart, C.; Johnson, E. L.; Nemhauser, G. L.; Savelsbergh, M. W. P.; and Vance, P. H. 1998. Branch-and-price: Column generation for solving huge integer programs. *Operations Research* 46(3):316–329.

Blum, A.; Gupta, A.; Procaccia, A. D.; and Sharma, A. 2013. Harnessing the power of two crossmatches. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, 123–140.

Blum, A.; Dickerson, J. P.; Haghtalab, N.; Procaccia, A. D.; Sandholm, T.; and Sharma, A. 2015. Ignorance is almost bliss: Near-optimal stochastic matching with few queries. In *Proceedings of the ACM Conference on Economics and Computation (EC)*.

Chen, Y.; Li, Y.; Kalbfleisch, J. D.; Zhou, Y.; Leichtman, A.; and Song, P. X.-K. 2012. Graph-based optimization algorithm and software on kidney exchanges. *IEEE Transactions on Biomedical Engineering* 59:1985–1991.

Cormen, T.; Leiserson, C.; Rivest, R.; and Stein, C. 2009. *Introduction to Algorithms*. Cambridge, MA: MIT Press, third edition.

Dickerson, J. P., and Sandholm, T. 2014. Multi-organ exchange: The whole is greater than the sum of its parts. In *AAAI Conference on Artificial Intelligence (AAAI)*, 1412–1418.

Dickerson, J. P., and Sandholm, T. 2015. FutureMatch: Combining human value judgments and machine learning to match in dynamic environments. In *AAAI Conference on Artificial Intelligence (AAAI)*.

Dickerson, J. P.; Procaccia, A. D.; and Sandholm, T. 2012a. Dynamic matching via weighted myopia with application to kidney exchange. In *AAAI Conference on Artificial Intelligence (AAAI)*, 1340–1346.

Dickerson, J. P.; Procaccia, A. D.; and Sandholm, T. 2012b. Optimizing kidney exchange with transplant chains: Theory and reality. In *International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 711–718.

Dickerson, J. P.; Procaccia, A. D.; and Sandholm, T. 2013. Failure-aware kidney exchange. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, 323–340.

Ergin, H.; Sönmez, T.; and Ünver, M. U. 2014. Lung exchange. Working paper.

Glorie, K.; Carvalho, M.; Constantino, M.; Bouman, P.; and Viana, A. 2015. Robust models for the kidney exchange problem. Working paper.

Glorie, K. M.; van de Klundert, J. J.; and Wagelmans, A. P. M. 2014. Kidney exchange with long chains: An efficient pricing algorithm for clearing barter exchanges with branch-and-price. *Manufacturing & Service Operations Management (MSOM)* 16(4):498–512.

HHS/HRSA/HSB/DOT. 2011. OPTN/SRTR annual data report.

Luo, S., and Tang, P. 2015. Mechanism design and implementation for lung exchange. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Mak-Hau, V. 2015. On the kidney exchange problem: cardinality constrained cycle and chain problems on directed graphs: a survey of integer programming approaches. *Journal of Combinatorial Optimization* 1–25.

Manlove, D., and O'Malley, G. 2014. Paired and altruistic kidney donation in the UK: Algorithms and experimentation. *ACM Journal of Experimental Algorithmics* 19(1).

Montgomery, R.; Gentry, S.; Marks, W. H.; Warren, D. S.; Hiller, J.; Houp, J.; Zachary, A. A.; Melancon, J. K.; Maley, W. R.; Rabb, H.; Simpkins, C.; and Segev, D. L. 2006. Domino paired kidney donation: a strategy to make best use of live non-directed donation. *The Lancet* 368(9533):419–421.

Neuen, B. L.; Taylor, G. E.; Demaio, A. R.; and Perkovic, V. 2013. Global kidney disease. *The Lancet* 382(9900):1243.

Rapaport, F. T. 1986. The case for a living emotionally related international kidney donor exchange registry. *Transplantation Proceedings* 18:5–9.

Rees, M.; Kopke, J.; Pelletier, R.; Segev, D.; Rutter, M.; Fabrega, A.; Rogers, J.; Pankewycz, O.; Hiller, J.; Roth, A.; Sandholm, T.; Ünver, U.; and Montgomery, R. 2009. A non-simultaneous, extended, altruistic-donor chain. *New England Journal of Medicine* 360(11):1096–1101.

Roth, A.; Sönmez, T.; Ünver, U.; Delmonico, F.; and Saidman, S. L. 2006. Utilizing list exchange and nondirected donation through 'chain' paired kidney donations. *American Journal of Transplantation* 6:2694–2705.

Roth, A.; Sönmez, T.; and Ünver, U. 2004. Kidney exchange. *Quarterly Journal of Economics* 119(2):457–488.

Roth, A.; Sönmez, T.; and Ünver, U. 2005a. A kidney exchange clearinghouse in New England. *American Economic Review* 95(2):376–380.

Roth, A.; Sönmez, T.; and Ünver, U. 2005b. Pairwise kidney exchange. *Journal of Economic Theory* 125(2):151–188.

Saran, R.; Li, Y.; Robinson, B.; Ayanian, J.; Balkrishnan, R.; Bragg-Gresham, J.; Chen, J.; Cope, E.; Gipson, D.; He, K.; et al. 2015. US renal data system 2014 annual data report: Epidemiology of kidney disease in the United States. *American Journal of Kidney Diseases* 65(6 Suppl 1):A7.

Ünver, U. 2010. Dynamic kidney exchange. *Review of Economic Studies* 77(1):372–414.

A Extra Proofs

Proof of Theorem 1. Suppose the algorithm returns some cycle $c = \langle v_1, v_2, \ldots, v_n \rangle$. Then $(v_n, v_1) \in E$, and there is a path p from v_1 to v_n with weight $w(p)$, where $w(p) + w(v_n, v_1) < 0$. Running Bellman-Ford for L (resp. K) steps ensures that the cycle does not exceed the exogeneous cycle (resp. chain) cap. Furthermore, our modification only update the distances to a vertex v_i along p if v_i does not occur in p yet, so any computed path uses each vertex at most once. Also note that the edge (v_n, v_1) cannot be part of that path, since we never update the distance to the source. Thus, cycle c uses each vertex at most once, and has negative weight. □

B Pseudocode for the Corrected Polynomial Pricing Scheme

In this section, we provide as Algorithm 1 the full pseudocode for our adapted version of the polynomial pricing algorithm provided by Glorie et al. (2014). Algorithm 1 serves as the cycle and chain pricing engine used for the BNP-POLY experiments in Section 5.

In Algorithm 1, for a fixed source, let $d_i(v)$ represent the computed distance from that source to v after the ith step of the algorithm, where $d_0(v)$ represents the distances before any steps are performed. Distance is defined as the sum of the edge weights in the computed path. Let L and K be the maximum allowable cycle and chain lengths, respectively. Finally, let A be the set of altruist donors and let P be the set of donor-patient pairs. The function GETNEGATIVECYCLES is called with the reduced graph $G = (V, E)$, cycle cap L, and chain cap K.

C Other Solver Improvements and Implementation Details

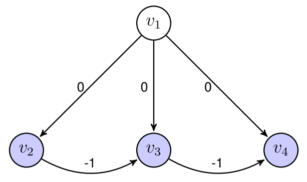
In this section, we discuss improvements we made to the base solver of Abraham et al. (2007), as well as additional implementation details regarding the implementation of the Glorie et al (2014) methodology and our fixed version of it.

The adapted Bellman-Ford pricing method is more complicated than normal Bellman-Ford

We now describe how it is necessary in the implementation of the adapted Bellman-Ford method of Algorithm 1 to maintain the entire 2-dimensional predecessor array for vertices in the pricing graph, whereas a 1-dimensional array suffices in typical Bellman-Ford (see, e.g., Cormen et al. (2009)). This difference arises from the fact that we need to limit the number of edges in a path, or else the cycles we generate may exceed the permissible length. If we only use a 1-dimensional predecessor array, running Bellman-Ford for k steps does not guarantee paths of length at most k.

The intuition for this requirement is as follows: say we would like to compute paths of length at most k, so we run k steps of Bellman-Ford. Suppose that after $k - 1$ steps, the path to vertex u has $k-1$ edges, and that on the last step, the distance to a neighboring vertex v is updated via vertex u. If nothing else is updated, the path to vertex v would have k edges, which is valid. However, suppose the path to vertex u also gets updated, and that this updated path also contains k edges. Since at this final step vertex v has u as its predecessor (denoted `pred`$(v) = u$), the path to vertex v is the path to u plus the edge (u, v), so the path to v is now $k + 1$ edges long, which is invalid.

It is also not viable to simply exclude those paths that end up with more than k edges, since the algorithm may have forgotten a different path to v that was less promising at the time, but—under the additional constraint that paths of length greater than k are invalid—would have ended up only using k edges. If that other path were to represent the only positive-price cycle, Algorithm 1 would mistakenly return that there are no positive price cycles, breaking the correctness of the branch-and-price solver. Figure 6 illustrates such a situation.



$$\begin{tikzcd} & v_1 \arrow[dl, "0"'] \arrow[d, "0"] \arrow[dr, "0"] & \\ v_2 \arrow[r, bend right, "-1"'] & v_3 \arrow[r, bend right, "-1"'] & v_4 \end{tikzcd}$$

Figure 6: Example pricing graph where a 2-dimensional predecessor array is used for correctness (when $L = 3$).

In the pricing graph of Figure 6, suppose we used the standard 1-dimensional predecessor array and ran Bellman-Ford for two steps, using vertex v_1 as the source. Then, Table 1 shows the computed predecessors for each of the three non-source vertices.

If only the last predecessor array row is examined, the path to vertex v_4 that is extracted by following the

Algorithm 1 Corrected polynomial-time Bellman-Ford search for negative weight cycles.

```
function GetNegativeCycles($G = (V, E), L, K$)
    $\mathcal{C} \leftarrow \emptyset$                                  ▷ Accumulator set for negative weight cycles
    for each $s \in V$ do
        $N \leftarrow s \in A \,?\, K - 1 : L - 1$                      ▷ Set maximum step number based on chain or cycle cap
        $pred_0(v) = \emptyset \;\; \forall v \in V$
        $d_0(s) = 0$                                                    ▷ Distance from source to source is zero
        $d_0(v) = \infty \;\; \forall v \neq s \in V$                   ▷ Distance at step 0 to other vertices is infinite
        for $i \in \{1, \ldots, N\}$ do
            $d_i(v) = d_{i-1}(v) \;\; \forall v \neq s \in V$
            $pred_i(v) = pred_{i-1}(v) \;\; \forall v \neq s \in V$
            for each $(u, v) \in E$ do
                if $v \notin$ TraversePreds($u, pred, i - 1$) then      ▷ Avoid loops in path
                    if $d_{i-1}(u) + w(u, v) < d_i(v)$ then             ▷ If this step decreases the distance to node
                        $d_i(v) \leftarrow d_{i-1}(u) + w(u, v)$        ▷ Update to shorter distance
                        $pred_i(v) \leftarrow (u, i - 1)$               ▷ Store correct predecessor
        for each $v \neq s \in V$ do                                    ▷ Find negative weight cycles with $s$ as the source
            if $d_N(v) + w(v, s) < 0$ then
                $\mathcal{C} \leftarrow \mathcal{C} \cup$ TraversePreds($v, pred, N$)
    return $\mathcal{C}$
function TraversePreds($v, pred, n$)
    $c \leftarrow []$                                                   ▷ Start with an empty list (representing a cycle or chain)
    $curr \leftarrow v$
    while $curr \neq \emptyset$ do                                      ▷ Until we reach the source node ...
        $c \leftarrow curr + c$                                         ▷ Add predecessor to path
        $(u, i) \leftarrow pred_n(curr)$                                ▷ Get predecessor of predecessor
        $curr \leftarrow u; \;\; n \leftarrow i$
    return $c$
```

Step #	pred(v_2)	pred(v_3)	pred(v_4)
0	–	–	–
1	v_1	v_1	v_1
2	v_1	v_2	v_3

Table 1: Predecessor table computed for the graph of Figure 6 with vertex source v, for $k = 2$ steps.

pred mapping will be (v_1, v_2, v_3, v_4)—which contains three edges, even though we only ran Bellman-Ford for $k = 2$ steps.

It is even possible to form paths of arbitrary length after two steps. Suppose there also existed $v_4 \ldots v_n$. Add an edge (v_1, v_i) with weight 0, and an edge (v_i, v_{i+1}) with weight -1 for all i. Then after two steps, a 1D predecessor array would implicitly hold a path of length n.

We solve this issue as follows. When we update the distance to vertex v by way of neighboring vertex u, we cannot simply replace the new path to vertex v by the path to vertex u plus the edge (u, v), as would be done in typical Bellman-Ford. Instead, the new path to vertex v should be the path to vertex u *at the time of the update* plus the edge (u, v). In the above example, when we update the distance to vertex v_4 on the second step, the path to vertex v_3 is (v_1, v_3), so the overall path to vertex v_4 should be (v_1, v_3, v_4).

To handle this, whenever we make an update we must not only store the predecessor, but also the time of the update (the step number). Then when extracting a path to vertex, we can jump to the predecessor of that vertex at the time of the update. However, this process requires storing the entire 2-dimensional array of all (updated) predecessors on each step.

Note that every time the algorithm jumps to a vertex's predecessor, it moves at least one time step backwards in the 2-dimensional predecessor array. Since the path creation process ends when the time step reaches 0, and there are a total of k steps (rows in the array), any extracted paths are guaranteed to have at most k edges. Also note that this process does not change the sequence of updates in the algorithm; instead, it ensures that the paths extracted in the end accurately reflect the sequence of updates.

Finally, note that when we store paths at the time of update, the final path to a vertex v may contain a vertex u but not the final path to vertex u. This is explains in the proof of Theorem 2 how we are able to continue to make updates to p_c, even after p is computed later.

A simple optimization

A standard implementation optimization of Bellman-Ford that we use is that on each step, we only check out-edges from nodes whose distances we just updated. This is especially useful in our context, because when L and K are small, there will often only be a few nodes whose distances were just updated.

Edge branching scheme and heuristics

During the branch-and-bound (and thus also branch-and-price) search, when the LP relaxation at a given node is non-integral, and the upper and lower bounds do not fathom that subtree, a decision variable (or set of decision variables) must be chosen on which to branch. For example, the original branch-and-price-based solver BNP-DFS chooses a single variable x_c corresponding to a cycle or chain c whose value is non-integral (i.e., for a binary variable x_c in the IP, the relaxed value in the LP $x_c \in (0, 1)$) to branch on (Abraham, Blum, and Sandholm 2007). If there is more than one such non-integral variable, the one with value closest to 0.5 is chosen, and the subtree with $x_c = 1$ is explored first in depth-first order.

As discussed by Glorie et al. (2014), their polynomial pricing algorithm (and also the fixed version we present in this paper) is incompatible with branching on cycles. Consequently, we use an edge branching scheme in BNP-POLY: when the LP solution at a node is fractional, a non-integral *edge* is chosen to branch on. Glorie et al. (2014) showed how to branch on edge variables, while still in a cycle formulation model. That is the scheme we use in this paper, as described below.

Let x_c be the relaxed decision variable for cycle c in the LP. Then, the value for any edge e in the compatibility graph is $e = \sum_{c:e\in c} x_c$, the sum of relaxed values for $x_c \in [0, 1]$ over all cycles c that contain edge e. It is not immediately clear that the presence of a fractional cycle in the LP implies the presence of a fractional edge, but Glorie et al. (2014) show that this is in fact the case.

At a given node in the search tree, there may be many edges that are fractional under this definition. In our experiments, we choose an edge e with value closest to 0.5. We branch first to the subtree that forces the inclusion of this edge. That forced inclusion is implemented by an additional constraint stating $\sum_{c:e\in c} x_c \geq 1$ (i.e., at least one cycle containing edge e must be contained in the final solution). The alternate direction is implemented by an additional constraint $\sum_{c:e\in c} x_c \leq 0$.