Published in final edited form as:

Oper Res Health Care. 2019 March ; 20: 45–55. doi:10.1016/j.orhc.2018.10.002.

An efficient algorithm to enumerate sets with fallbacks in a kidney paired donation program

Wen Wang^{a,*}, Mathieu Bray^a, Peter X.K. Song^{a,b}, and John D. Kalbfleisch^{a,b}

^aDepartment of Biostatistics, University of Michigan, 1415 Washington Heights, Ann Arbor, MI, USA

^bUniversity of Michigan Kidney Epidemiology and Cost Center, University of Michigan, 1415 Washington Heights, Ann Arbor, MI, USA

Abstract

Kidney paired donation is a partial solution to overcoming biological incompatibility preventing kidney transplants. A kidney paired donation (KPD) program consists of altruistic or non-directed donors (NDDs) and pairs, each of which comprises a candidate in need of a kidney transplant and her/his willing but incompatible donor. Potential transplants from NDDs or donors in pairs to compatible candidates in other pairs are determined by computer assessment, though various situations involving either the donor, candidate, or proposed transplant may lead to a potential transplant failing to proceed. A KPD program can be viewed as a directed graph with NDDs and pairs as vertices and potential transplants as edges, where failure probabilities are associated with each vertex and edge. Transplants are carried out in the form of directed cycles among pairs and directed paths initiated by NDDs, which we refer to respectively as cycles and chains. Previous research shows that selecting disjoint subgraphs with a view to creating fallback options when failures occur generates more realized transplants than optimal selection of disjoint chains and cycles. In this paper, we define such subgraphs, which are called locally relevant (LR) subgraphs, and present an efficient algorithm to enumerate all LR subgraphs. Its computational efficiency is significantly better than the previous, more restrictive, algorithms.

Keywords

Kidney paired donation; Non-directed donor; Locally relevant subgraph; Fallback options; Breadth-first search

1. Introduction

Kidney transplants are often prevented by biological incompatibility, either due to ABO blood type mismatch or the transplant candidate's sensitivity to human leukocyte antigens (HLA) of the donor. A kidney paired donation (KPD) program provides a partial yet clinically important solution to such incompatibility [1–3]. A KPD program consists of

*Corresponding author. wangwen@umich.edu (W. Wang), braymath@umich.edu (M. Bray), pxsong@umich.edu (P.X.K. Song), jdkalbf@umich.edu (J.D. Kalbfleisch).

The data underlying the simulations in this paper and some descriptions are available at <https://github.com/jiangwenjiaowa/KPDLRS>.

pairs, each comprising a candidate in need of a kidney transplant and her/his willing but incompatible donor, and altruistic or non-directed donors (NDDs), each willing to donate, but with no associated candidate. Note that not all KPD programs involve NDDs. In a KPD program, transplants are carried out via cycles comprised of pairs and/or chains initiated by NDDs. A cycle consists of a sequence of distinct pairs such that the donor in each pair is compatible with the candidate in the next pair, with the donor in the final pair giving to the candidate in the first pair. A chain consists of an NDD followed by a sequence of distinct pairs such that the donor in each pair (or NDD, in the case of the initial transplant) is compatible with the candidate in the next pair. The donor in the last pair of a chain is uncommitted and may serve as an NDD in a subsequent set of exchanges. The length of a chain or a cycle is the number of pairs involved. See an example in Fig. 1 [4].

1.1. Preliminaries

A KPD program can be represented by a directed graph $\mathcal{G} = (V, E)$: the vertex set V is the union of the set of all pairs (V_p) and all NDDs (V_a); the edge set is

$$E = \{(u, v) \mid u \in V_a \cup V_p, v \in V_p, \text{donor in } u \text{ is compatible with candidate in } v\}.$$

Compatibilities among pairs and NDDs are assessed by virtual crossmatch, which is a computerized assessment of ABO blood type compatibility and candidate sensitivities to donor HLA antigens. In this paper, “vertex” and “pair/NDD” are referred to interchangeably, as are “edge” and “potential transplant”.

In practice, a KPD program is subject to uncertainty in that a potential transplant once selected may fail to proceed to transplantation. We consider two types of uncertainties: (i) uncertainty on the vertices, caused, for example, by sickness, pregnancy, business or family obligations, other transplant opportunities or death affecting either the donor or candidate [5]; and (ii) uncertainty on the edges, due to the fact that a compatible virtual crossmatch needs to be confirmed by a laboratory crossmatch before transplant or that a candidate may not agree to a proposed transplant. Length of chains and cycles are typically capped due to limited medical facilities and the high chance of failure of a long chain or cycle.

To manage a KPD program, match runs to select potential transplants are conducted on a regular basis. A typical match run for selecting chains and cycles includes Step (1a) to (4a) as shown in Table 1. Note that Step (3a) can be formulated as an integer programming (IP) problem [6]:

$$\begin{aligned} & \text{maximize} \quad \sum_{c \in C} w_c z_c, \\ & \text{subject to} \quad \sum_{c \in C: v \in V(c)} z_c \leq 1, \forall v \in V \text{ and } z_c \in \{0, 1\}, \forall c \in C, \end{aligned}$$

where C is the set of all chains and cycles enumerated in Step (1a), and w_c is the weight for the chain or cycle c in C calculated in Step (2a). For example, as shown in the hypothetical KPD program illustrated by Fig. 2, all cycles with length no larger than 3 are easily found as

(1, 3, 2), (3, 4) and (3, 6, 4). (Note that (1, 3, 2) refers to a cycle in which the donor in pair 1 gives to the candidate in pair 3, donor 3 gives to candidate 2, and donor 2 gives to candidate 1 to complete the cycle.) If we use the number of potential transplants to measure the value of each cycle and conduct a match run, then the solution would involve choosing either the cycle (1, 3, 2) or the cycle (3, 6, 4). With either choice we would realize three transplants if all vertices and edges are viable and otherwise, no transplant is realized. It is easy to see, however, that there is an advantage to the choice (3, 6, 4) if we consider the possibility of falling back to (3, 4) as follows: if (3, 6, 4) succeeds, then 3 transplants are realized; if (3, 6, 4) fails but (3, 4) succeeds, then the 2 transplants in (3, 4) are realized; if both (3, 6, 4) and (3, 4) fail, then no transplant is realized. Here (3, 4) serves as a fallback. This solution proposes two competing cycles together as a solution and can generate more realized transplants.

To exploit this idea of fallbacks, we introduce the notion of a locally relevant subgraph in Section 2 and enumerate the class L of all such subgraphs subject to some specified parameters in Step (1b). In Step (2b), associated with each such subgraph I , we can determine an expected number of transplants, w'_I , by taking account of all fallbacks within I . We then proceed to Step (3b) by finding the disjoint selection of locally relevant subgraphs that maximizes the sum of the expected numbers of transplants achieved by solving the following IP problem:

$$\begin{aligned} & \text{maximize } \sum_{I \in L} w'_I z_I \\ & \text{subject to } \sum_{I \in L: v \in V(I)} z_I \leq 1, \forall v \in V \text{ and } z_I \in \{0, 1\}, \forall I \in L. \end{aligned} \quad (1)$$

In the final Step (4b), we check the viability of vertices and edges within each selected locally relevant subgraph I to determine which vertices and edges are viable. This defines a subgraph of viable vertices and edges. Within this graph, with the uncertainties resolved, we can select disjoint chains and/or cycles to implement so as to maximize the number of transplants. This can typically be done by inspection.

1.2. Overview of prior research

There is substantial prior research into various aspects of KPD programs including issues of optimization formulation, optimization algorithms, in corporation of uncertainty as well as more clinical aspects such as inclusion of compatible pairs, ethical issues, global programs, and integration with deceased donor programs. We will concentrate our review, however, on a much less extensive literature having to do with studies related to incorporating uncertainties and fallbacks in KPD programs.

When managing a KPD program using cycles and chains, the two main types of optimization formulations are the cycle formulation, which has one binary decision variable for each cycle and chain of interest [7], and the edge formulation, which includes one decision variable for each edge [8]. The cycle formulation is related to the methods discussed in this paper. It generally involves many decision variables (exponentially large)

but has relatively simple constraints that increase in a linear manner with the number of vertices. This formulation generalizes to problems including uncertainties and fallbacks. Conversely, the number of decision variables in the edge formulation is linear whereas the number of constraints is exponential in terms of number of edges. In the extended edge formulation in [6], the numbers of decision variables and constraints are polynomial functions of the number of edges. Although the edge formulation is attractive from an optimization point of view, it appears to be much less flexible for incorporation of uncertainties and fallbacks.

As uncertainties on vertices and edges and fallbacks are introduced into KPD programs by [9–11], there are three different types of criteria considered in the associated match run. The first type is to select disjoint cycles and chains maximizing the expected number of transplants allowing fallbacks within each cycle and chain; this is referred to as “fallbacks” in [12,13], “expected vertex-disjoint cycle packing” in [10] and “internal recourse” in [14]. The second type is to select disjoint cycles and chains to maximize the expected number of transplants in the current and future match runs [15]. The third type is to select disjoint vertex induced subgraphs to maximize the expected number of transplants, allowing fallbacks within each subgraph; this is referred to as “locally relevant subgraph” in this paper, “extended fallbacks” in [13] and “subset recourse” in [14].

Some additional work on the third type can be found in [16,13,14]. When failure probability is moderate, it has been demonstrated that such locally relevant subgraphs can lead to greatly improved selection processes that yield many more realized transplants, by as much as, 8% compared to fallbacks within cycles and chains [13].

1.3. Purpose and outline

A typical match run to select locally relevant subgraphs and arrange transplants proceeds through steps (1b) to (4b) as shown in Table 1. Steps (1b), (2b) and (3b) are computationally much more extensive than Steps (1a), (2a) and (3a). To partially address these computational difficulties, [14] proposed algorithms to solve Steps (1b) and (2b) for a KPD program without NDDs. But this substantially restricts applications since most exchanges also have NDDs, who are playing an increasingly important role in selecting exchanges. The purpose of this paper is to develop a new more efficient algorithm to solve Step (1b) with both pairs and NDDs. The new algorithms enable us to handle relatively large KPD programs and larger locally relevant subgraphs.

We consider a set up that is the same as in [14] except the latter only focuses on cycles; although it can be easily extended to include chains. Thus, when applied in an actual KPD, either of these approaches would give rise to essentially equivalent solutions and the same optimum expected number of transplants. For this reason, a comparison of transplants obtained between the two approaches is not needed. Our aim in this paper is to develop an efficient algorithm to perform the calculations in Step (1b). In Section 5, we compare our approach to enumerating locally relevant subgraphs to the approach suggested in [14].

The rest of this article is organized as follows: Section 2 presents the definition of a locally relevant subgraph. Section 3 investigates the properties of locally relevant subgraphs and

presents algorithms to verify whether a graph is locally relevant. Section 4 concerns algorithms to enumerate locally relevant subgraphs under a given size. Section 5 examines efficiency of the proposed enumeration algorithms via simulation. Finally some concluding remarks are given in Section 6. Technical details are included in the appendix and supplementary materials.

2. Locally relevant subgraph

This section focuses on the definition of a locally relevant sub-graph. Motivated by KPD programs, we consider a simple directed graph $\mathcal{G} = (V, E)$, where V is the set of vertices and E is the set of edges. Note that we also use $V(G)$ and $E(G)$ to indicate the vertices and edges of the graph G . Vertices are of two types so that $V = V_a \cup V_p$ where V_p is the set of pairs and V_a is the set of NDDs with $V_a \cap V_p = \emptyset$. No edge points to any vertex in V_a . For convenience, the vertices are numbered from 1 to N where $|V| = N$, and the vertices in V_a , if any, are numbered first with values 1 to a , where $|V_a| = a \geq 0$. We refer to this as the default order and vertices are referred to using this numbering system. Cycles are simple circuits and chains are simple paths initiated by NDDs. The subgraph induced by S , a subset of V , is denoted by $G[S]$. The order of the subgraph is defined by $|S|$. The transpose graph, $\mathcal{G}^T$, of a directed graph $\mathcal{G} = (V, E)$ is defined by a directed graph whose vertex set is $V(\mathcal{G})$ and edge set is $\{(i, j) | (j, i) \in E(\mathcal{G})\}$.

Definition 1. The simple directed graph $\mathcal{G} = (V, E)$ is locally relevant (LR) with reference to cycles with length limit x ($x \in \{0, 2, 3, \dots, \infty\}$) and chains with length limit y ($y \in \{0, 1, 2, \dots, \infty\}$) ($\mathcal{G}$ is LR(x, y)) if and only if $|V| > 1$ and, for any partition dividing V into two disjoint nonempty parts, there exists a cycle of length $\leq x$ or a chain of length $\leq y$ that contains vertices in both parts.

Definition 2. We further define the class of all locally relevant subgraphs of size n or less as

$$\text{LRS}(x, y; n) = \{G[S] | S \subseteq V, |S| \leq n, G[S] \text{ is LR}(x, y)\},$$

where $x \in \{0, 2, 3, \dots, \infty\}$, $y \in \{0, 1, 2, \dots, \infty\}$ and $n \in \{2, 3, \dots\}$.

A natural question concerning Definition 1 is whether it is necessary to differentiate between NDDs and pairs in the formation of chains and cycles. One way to unify NDDs and pairs is by adding “back-edges” from all pairs to NDDs, effectively converting all chains into cycles. The equivalence between chains and cycles needs one extra condition: each “generalized cycle” can have at most one NDD. However in some graphs, back-edges may be misleading. For example, the graph in Fig. 3 is not LR(∞, ∞) by Definition 1. But if we treat NDDs as pairs and back-edges as regular edges, this subgraph would be recognized as a locally relevant subgraph of size 4. These extra complications motivates us to treat chains and cycles as separate entities.

The primary objective of this article is to develop an effective algorithm to enumerate the elements in $\text{LRS}(x, y; n)$ in G . This step is important enough to warrant a separate and careful treatment.

3. LR(x, y) verification

We begin with a brief overview of this section. We first verify that $LRS(\infty, \infty; n)$ is the disjoint union of $LRS(\infty, 0; n)$ and $LRS(0, \infty; n)$. We then develop two algorithms. The first verifies whether a subgraph is $LR(\infty, 0)$ and the second verifies whether a subgraph is $LR(0, \infty)$. Finally we develop necessary and sufficient conditions for a subgraph to be $LR(x, y)$ and an algorithm to verify whether a graph is $LR(x, y)$. In Section 4, these results are used for enumeration of LR subgraphs in a KPD program.

3.1. Preliminaries

In this discussion, the breadth first search (BFS) algorithm [17] is based on the default order; in other words, among all vertices eligible to be visited in the BFS, the first node in the default order is visited first. The root of a BFS tree is the earliest visited vertex by the BFS algorithm among vertices in the BFS tree. The class of all the BFS trees of a directed graph is called the BFS forest. Additionally, given a vector A , its i th entry is denoted by $A[i]$. Given a set A , $|A|$ denotes its cardinality and $\min A$ denotes its minimum element. If A is a vector of vertices or if A is a vector of vectors of vertices, then $\hat{A}$ represents the set of the distinct vertices in A . Given a matrix A , its (i, j) entry is denoted by $A[i, j]$. The adjacency matrix of G , with vertices arranged in the default order, is denoted by M , where $M[i, j] = I((i, j) \in E)$ for $i, j \in \{1, 2, \dots, N\}$ and $I(\cdot)$ is the indicator function. The vertex type vector, T , is a vector of length $|V|$, where $T[i] = I(i \in V_a)$ represents whether each vertex is an NDD or not. If $S = \{v_1, \dots, v_{|S|}\}$ is a nonempty subset of $V(\mathcal{G})$, the $|S| \times |S|$ matrix, $SP(S)$, has (i, j) entry $SP(S)[i, j]$, the length of the shortest simple path from v_i to v_j . Note that $SP(S)[i, i] = 0$ for all i and, if there is no path from v_i to v_j in $\mathcal{G}[S]$, then $SP(S)[i, j] = \infty$.

The following proposition states that $LRS(\infty, \infty; n)$ is a disjoint union of $LRS(\infty, 0; n)$ and $LRS(0, \infty; n)$.

Proposition 1.

1. $LRS(\infty, 0; n) = LRS(\infty, \infty; n) \cap \{\mathcal{G}[S] \mid S \subseteq V_p\};$
2. $LRS(0, \infty; n) = LRS(\infty, \infty; n) \cap \{\mathcal{G}[S] \mid S \subseteq V, S \cap V_a \neq \emptyset\};$
3. $LRS(\infty, \infty; n) = LRS(\infty, 0; n) \cup LRS(0, \infty; n), \quad LRS(\infty, 0; n) \cap LRS(0, \infty; n) = \emptyset.$

Proof. The first property is trivial and hence the proof is omitted.

For the second property, it can be seen that $LRS(0, \infty; n) \subseteq LRS(\infty, \infty; n) \cap \{\mathcal{G}[S] \mid S \subseteq V, S \cap V_a \neq \emptyset\}$. Suppose that $LRS(0, \infty; n) \not\subseteq LRS(\infty, \infty; n) \cap \{\mathcal{G}[S] \mid S \subseteq V, S \cap V_a \neq \emptyset\}$. Then, there exists a $\mathcal{G}[S] \in LRS(0, \infty; n)$ such that $S \cap V_a = \emptyset$ and $S = A \cup B$, $A \neq \emptyset, B \neq \emptyset$, $A \cap B = \emptyset$ and no chain in $\mathcal{G}[S]$ that connects A and B . Without loss of generality, we assume that $A \cap V_a = \emptyset$. Define $C = \{v \mid v \in S \text{ and } v \text{ is reachable from at least one vertex in } A \cap V_a \text{ in } \mathcal{G}[S]\}$. It is easy to see $C \subseteq A$. This implies that there exists a cycle connecting C and $B \cup \{A \setminus C\}$, thus creating a chain from A to B and a contradiction.

The third property follows immediately from the first two properties. $\square$

By Proposition 1, we can verify the $LR(\infty, \infty)$ property by verifying the $LR(\infty, 0)$ property and the $LR(0, \infty)$ property separately. Likewise, enumeration of elements in $LRS(x, y; n)$ may be carried out by enumerating elements in $LRS(\infty, 0; n)$ and $LRS(0, \infty; n)$ separately and further verifying whether they are in $LRS(x, y; n)$.

3.2. $LR(\infty, 0)$ verification

By [18], we have the following proposition:

Proposition 2. For any $S \subseteq V(\mathcal{G})$ such that $1 < |S| \leq n$, the following statements are equivalent:

1. $\mathcal{G}[S]$ is $LR(\infty, 0)$;
2. $\mathcal{G}[S]$ is strongly connected;
3. $\mathcal{G}[S]$ and $\mathcal{G}[S]^T$ each have a one-tree BFS forest.

Proposition 2 leads to Algorithm 1, which verifies whether $\mathcal{G}[S]$ is $LR(\infty, 0)$:

Algorithm 1 Algorithm to verify if $\mathcal{G}[S]$ is $LR(\infty, 0)$

Input: The adjacency matrix M of $\mathcal{G}$, the vertex type vector T of $\mathcal{G}$,
 S which is a subset of $V(\mathcal{G})$.
Output: A boolean variable indicating if $\mathcal{G}[S]$ is $LR(\infty, 0)$.

```

1: function  $LRSI_{NFOVERI}(M, T, S)$ 
2:   if  $|S| < 2$  then
3:     return false
4:   if  $S \cap V_a \neq \emptyset$  then
5:     return false
6:   if  $\mathcal{G}[S]$  does not have a one-tree BFS forest then
7:     return false
8:   if  $\mathcal{G}^T[S]$  does not have a one-tree BFS forest then
9:     return false
10:  return true

```

3.3. $LR(0, \infty)$ verification

To verify that $\mathcal{G}[S]$ is $LR(0, \infty)$, we have the following three propositions. Proposition 5 gives simple necessary and sufficient conditions.

Proposition 3. If $\mathcal{G}[S]$ is $LR(0, \infty)$, then the following properties hold:

1. the number of trees in the BFS forest of $\mathcal{G}[S]$ is $|S \cap V_a|$; and
2. the roots of trees in the BFS forest of $\mathcal{G}[S]$ are NDDs.

If there is one and only one NDD in each BFS tree of $\mathcal{G}[S]$, we can construct a directed graph, $\widehat{\mathcal{G}}[S] = (\tilde{V}, \tilde{E})$ as follows: $\tilde{V}$ is the set of all trees in the BFS forest of $\mathcal{G}[S]$ and the default order of each NDD in $\mathcal{G}[S]$ is used to label each tree in $\tilde{V}$; $\tilde{E} = \{(i, j) \mid i, j \in \tilde{V}, i \neq j, \exists a, b \in S \text{ s.t. } a \in V(i), b \in V(j) \text{ and } (a, b) \in E(\mathcal{G}[S])\}$ and

$(a, b) \in E(\mathcal{G}[S])\}$. We further construct a graph, denoted by $\overline{\mathcal{G}[S]} = (\overline{V}, \overline{E})$, based on $\mathcal{G}[S]$ by setting $\overline{V} = \tilde{V}$ and $\overline{E} = \{(i, j) | i, j \in \tilde{V}, i < j \text{ and } (j, i) \in \tilde{E}\}$.

We establish the following properties for $\overline{\mathcal{G}[S]}$ and $\mathcal{G}[S]$:

Proposition 4. If $\mathcal{G}[S]$ is $\text{LR}(0, \infty)$, then we have:

1. $\forall i, j \in (\overline{\mathcal{G}[S]})$ such that $i < j$, then $(i, j) \notin E(\overline{\mathcal{G}[S]})$; and
2. $\overline{\mathcal{G}[S]}$ is connected.

Figs. 4(a), 4(b) and 4(c) present examples of $\mathcal{G}[S]$ (which is $\text{LR}(0, \infty)$), $\overline{\mathcal{G}[S]}$ and $\mathcal{G}[S]$, respectively. Fig. 4(d) gives the one-treeBFS forest of $\overline{\mathcal{G}[S]}$.

Proposition 5. For $S \subseteq V(\mathcal{G})$ with $|S| > 1$, $\mathcal{G}[S]$ is $\text{LR}(0, \infty)$ if and only if all of the following conditions are satisfied:

1. the number of trees in the BFS forest equals $|S \cap V_a|$;
2. $\overline{\mathcal{G}[S]}$ is connected.

The necessary and sufficient conditions given in Proposition 5 lead to Algorithm 2 below.

Algorithm 2 Algorithm to verify if $\mathcal{G}[S]$ is $\text{LR}(0, \infty)$

Input: The adjacency matrix M of $\mathcal{G}[S]$, the vertex type vector T of $\mathcal{G}[S]$, S which is a subset of $V(\mathcal{G})$.

Output: A boolean variable indicating whether $\mathcal{G}[S]$ is $\text{LR}(0, \infty)$.

```

1: function  $\text{LRS}_{\text{OINFVERI}}(M, T, S)$ 
2:   if  $|S| < 2$  then
3:     return false
4:   build BFS forest  $\mathcal{F}$  of  $\mathcal{G}[S]$  according to the default order
5:   if number of trees in  $\mathcal{F} \neq |S \cap V_a|$  then
6:     return false
7:   if  $\overline{\mathcal{G}[S]}$  is not connected then
8:     return false
9:   return true

```

3.4. LR(x, y) verification

Given a directed graph $\mathcal{G} = (V, E)$, it is useful to define a graph $\widehat{\mathcal{G}}_{x,y} = (\widehat{V}, \widehat{E}_{x,y})$ with the vertex set $\widehat{V} = V$ and the edge set

$$\begin{aligned} \widehat{E}_{x,y} = & \{(v_1, v_2) \mid v_1 \in V_a, v_2 \in V_p, SP(V)[v_1, v_2] < y + 1\} \cup \\ & \{(v_1, v_2) \mid v_1, v_2 \in V_p, v_1 < v_2, SP(V)[v_1, v_2] \\ & + SP(V)[v_2, v_1] < x + 1\}. \end{aligned}$$

Proposition 6. For $S \subseteq V(\mathcal{G})$ with $|S| > 1$, $\mathcal{G}[S]$ is LR(x, y) if and only if $\widehat{\mathcal{G}[S]}_{x,y}$ is connected.

See Figs. 4(a) and 5 for an example of an LR(3, 2) directed graph $\mathcal{G}$ and $\widehat{\mathcal{G}}_{3,2}$. From Proposition 6, we obtain the following:

Algorithm 3 Algorithm to verify if $\mathcal{G}[S]$ is LR(x, y)

Input: the adjacency matrix M of $\mathcal{G}$, the vertex type vector T of $\mathcal{G}$,
a subset S of V , cycle length limit x , chain length limit y .
Output: a boolean variable indicating whether $\mathcal{G}[S]$ is LR(x, y).

```

1: function LRSINFOVERI( $M, T, S, x, y$ )
2:   if  $|S| < 2$  then
3:     return false
4:   calculate shortest path length matrix  $SP(S)$  for  $\mathcal{G}[S]$ 
5:   from  $\widehat{\mathcal{G}[S]}_{x,y}$  from  $SP(S)$ 
6:   return whether  $\widehat{\mathcal{G}[S]}_{x,y}$  is connected

```

4. Enumeration of elements in LRS($x, y; n$)

Our strategy to enumerate the elements in LRS($x, y; n$) is to first enumerate the elements in LRS($\infty, \infty; n$) and then determine whether they belong to LRS($x, y; n$). Proposition 1 indicates that the first step of enumerating elements in LRS($\infty, \infty; n$) can be separately carried out by enumerating elements in LRS($\infty, 0; n$) and LRS($0, \infty; n$). Algorithm 3 can be utilized to check for LRS($x, y; n$).

4.1. Enumeration of elements in LRS($\infty, 0; n$)

Proposition 2 suggests that elements in LRS($\infty, 0; n$) can be enumerated in two steps: (i) enumerate subsets S of V_p such that $1 < |S| \leq n$ and $\mathcal{G}[S]$ has a one-tree BFS forest; (ii) verify if the $\mathcal{G}[S]^T$ has a one-tree BFS forest.

To implement step (i) above, a fundamental requirement is a convenient data structure to represent the BFS tree of a subgraph $\mathcal{G}[S]$, where $S \subseteq V_p$ and $\mathcal{G}[S]$ has a one-tree BFS forest. We propose to use two vectors of non-negative integers to represent a BFS tree: the “vertex vector” (VV) and the “depth vector” (DV). The VV stores default orders of all vertices in S in ascending order according to $\leq_{\mathcal{G}[S]}$, which is defined as follows.

Definition 3. Suppose that $\mathcal{G}[S]$ has a one-tree BFS forest. Then $\forall i, j \in S, i \leq_{\mathcal{G}[S]} j$ if and only if one of the following conditions is satisfied:

1. the depth of i in the BFS tree is strictly less than that of j ;
2. the depth of i in the BFS tree equals that of j and $i < j$.

Remark 1. Note that Definition 3 applies to any graph with a one-tree BFS forest when the vertices of the graph are labeled by integers.

The DV stores depths of the corresponding vertices listed in the VV . For the example given in Fig. 4(a), for the subgraph $\mathcal{G}[\{1, 5, 6, 7\}]$, we can see that the corresponding VV is (1, 6, 7, 5) and DV is (0, 1, 1, 2). Note that VV and DV characterize the BFS tree of $\mathcal{G}[S]$, but do not uniquely characterize the subgraph $\mathcal{G}[S]$ itself.

The following proposition provides a set of conditions that we use in Algorithm 4 to enumerate elements in $LRS(\infty, 0; n)$:

Proposition 7. Let $\mathcal{G}$ be a directed graph, A be a vector of vertices and B be a vector of integers. $\mathcal{G}[\overset{\circ}{A}]$ has a one-tree BFS forest and vectors A and B are the VV and DV of $\mathcal{G}[\overset{\circ}{A}]$ respectively if and only if the following conditions are satisfied:

1. $|\overset{\circ}{A}| > 0$ and, A and B are $|\overset{\circ}{A}|$ -vectors;
2. $B[1] = 0$ and if $|\overset{\circ}{A}| \geq 2$, then $B[2] = 1$;
3. $\forall 1 \leq k < |\overset{\circ}{A}|, B[k+1] - B[k]$ is either 0 or 1;
4. $A[1] = \min \overset{\circ}{A}$;
5. $\forall 1 \leq k \leq |\overset{\circ}{A}|$ such that $B[k] > 0$, then:
 - (a) $\exists 1 \leq l < k$ such that $B[l] = B[k] - 1$ and $(A[l], A[k]) \in E(\mathcal{G}[\overset{\circ}{A}])$ and
 - (b) $\nexists 1 \leq l < k$ such that $B[l] < B[k] - 1$ and $(A[l], A[k]) \in E(\mathcal{G}[\overset{\circ}{A}])$;
6. $\forall 1 \leq k \leq |\overset{\circ}{A}|$ such that $B[k+1] = B[k]$, we have $A[k+1] > A[k]$.

We will refer to the six conditions above as $Cond(\mathcal{G}, A, B)$ and Condition 5. for a given k as $Cond_5(\mathcal{G}, A, B, k)$.

Remark 2. If $\mathcal{G}$ is an undirected graph, the definition of VV and DV can be generalized for undirected graphs, and Proposition 7 remains true.

This leads to the Algorithm 4. (Note that terms used in the algorithm are defined in Appendix B.)

Algorithm 4 Enumeration of Elements in $\text{LRS}(\infty, 0; n)$

Input: The adjacency matrix M of $\mathcal{G}$, the vertex type vector T of $\mathcal{G}$, a non-negative integer n indicating size limit of subgraphs.

Output: A vector of vectors of vertices LRSList indicating an enumeration of elements in $\text{LRS}(\infty, 0; n)$.

```

1: function  $\text{LRSLIST\_ENUM}(M, T, n)$ 
2:   Set  $\text{LRSList}$ ,  $VV$  and  $DV$  as empty vectors
3:   for each  $i \in V_p$  do
4:      $VV = (i)$ 
5:      $DV = (0)$ 
6:      $\text{BFST\_TREE\_GEN}(M, n, VV, DV, \&\text{LRSList})$ 
7:   return  $\text{LRSList}$ 
8: function  $\text{BFST\_TREE\_GEN}(M, n, VV, DV, \&\text{LRSList})$ 
9:    $A = \{j\}$ 
10:  1.  $j \in V, j > VV[1], j \notin VV$  and;
11:  2.  $\forall 1 \leq k \leq \text{LENGTH}(VV)$  such that  $DV[k] < \text{BACK}(DV)$ , then  $M[VV[k], j] = 0$  and;
12:  3.  $\exists 1 \leq k \leq \text{LENGTH}(VV)$  such that  $DV[k] = \text{BACK}(DV), M[VV[k], j] = 1.$ 
13:  for each  $B \in \{B | B \subseteq A, 0 < |B| \leq n - \text{LENGTH}(VV), \text{elements in } B \text{ are in ascending order.}\}$  do
14:     $VVTemp = \text{CAT}(VV, B)$ 
15:     $DVTemp = \text{CAT}(DV, \text{REP}(\text{BACK}(DV) + 1, |B|))$ 
16:    if  $\mathcal{G}^T[VVTemp]$  has one-tree BFS forest then
17:       $\text{PUSH}(\text{LRSList}, VVTemp)$ 
18:  if  $\text{LENGTH}(VVTemp) < n$  then
19:     $\text{BFST\_TREE\_GEN}(M, n, VVTemp, DVTemp, \&\text{LRSList})$ 

```

Proof. The detailed proof of Algorithm 4 is given in Appendix C.

Here, we outline how Algorithm 4 works and give an example. Algorithm 4 is based on $\text{LRS}(\infty, 0; n) = \cup_{i=1}^n \text{LRS}(\infty, 0; n, i)$, where $\text{LRS}(x, y; n, i) = \{\mathcal{G}[S] \in \text{LRS}(x, y; n) | \min S = i\}$. Line 6 puts $\text{LRS}(\infty, 0; n, i)$ into LRSList by finding VV and DV of each vertex-induced subgraph with a one-tree BFS forest and then checking whether $\mathcal{G}[VV]^T$ has a one-tree BFS forest according to Proposition 2. In each call of BFST_TREE_GEN , there are four main steps: collection of eligible vertices, extension of VV , verification and recursion. More specifically, first, the set A in Line 9 collects all eligible vertices as the vertices with depth $\text{Back}(DV) + 1$ of the BFS tree described by VV and DV . Second, in Line 11, VV is extended by each nonempty subset of A to get $VVTemp$. Third, Line 13 checks whether $(\mathcal{G}[VVTemp])^T$ has a one-tree BFS forest before it is pushed into LRSList . Finally, in Line 16, BFST_TREE_GEN is called to further extend $VVTemp$.

Table 2 presents updates of $VVTemp$, $DVTemp$ and A along the course of $LRS_{INFOENUM}(M, T, 4)$, in which M and T are, respectively, the adjacency matrix and the vertex type vector of the graph in Fig. 2.

4.2. Enumeration of elements in $LRS(0, \infty; n)$

4.2.1. Enumeration using Algorithm 4—If there are relatively few NDDs in the $\mathcal{G}$, then we can add back-edges in the graph from every vertex in V_p to every vertex in V_a and use Algorithm 4 to enumerate elements of $LRS(\infty, 0; n)$ in the new graph that treats NDDs as pairs. For each obtained subgraph that includes more than one NDD, we verify whether it is $LR(0, \infty)$ in the original graph $\mathcal{G}$. Essentially, we need to remove structures similar to those in Fig. 3. In Section 5, we provide numerical comparisons between this procedure and the direct enumeration algorithm in Section 4.2.2.

4.2.2. Direct enumeration—We can enumerate the elements in $LRS(0, \infty; n)$ by enumerating subsets S of V that satisfy $|S| \leq n$ and the conditions in Proposition 5. We use three vectors to represent the BFS forest of the subgraph induced by such a subset S : the first is “vertex vector 2” ($VV2$), which is a vector of vectors and stores the VV s of BFS trees of $\mathcal{G}[S]$ in ascending order of $\leq_{\mathcal{G}[S]}$; the second is “depth vector 2” ($DV2$), which is a vector of vectors and stores the DV s of the BFS trees of $\mathcal{G}[S]$ in ascending order of $\leq_{\mathcal{G}[S]}$; the third is “tree depth vector” (TDV), which is the DV of the BFS tree of $\overline{\mathcal{G}[S]}$. For example, for $\mathcal{G}$ as in Fig. 4(a), $VV2$ is $((1, 6, 7, 5), (2, 4), (3))$, and $DV2$ is $((0, 1, 1, 2), (0, 1), (0))$, and TDV is $(0, 1, 1)$. Again, it should be noted that $VV2$, $DV2$ and TDV do not uniquely characterize the subgraph $\mathcal{G}[S]$.

Proposition 8. Let A be a vector of vectors of vertices in V , B be a vector of vectors of integers and C be a vector of integers. Then $\mathcal{G}[A] \in LRS(0, \infty; n)$ and A , B and C are the $VV2$, $DV2$ and TDV of $\mathcal{G}[A]$, respectively, if and only if all of the following conditions are satisfied:

1. $1 < |A| \leq n$;
2. A , B and C are of the same length;
3. $A[1][1], A[2][1], \dots, A[|A|][1]$ are distinct NDDs;
4. $\forall 1 \leq k \leq |A|, \text{Cond}(\mathcal{G}, A[k], B[k]);$ (Cond is defined in Proposition 7.)
5. If a and a' are vectors in A and $a[1] < a'[1]$, then no edges in $\mathcal{G}$ point from any vertex in a to any vertex in a' ;
6. $\overline{\text{Cond}(\mathcal{G}[A], (A[1][1], A[2][1], \dots, A[|A|][1]), C)}$.

This leads to Algorithm 5.

Proof. Section 7 in supplementary material presents a detailed version of this algorithm and its proof. $\square$

Here, we outline how Algorithm 5 works and provide an example. As before, we have $LRS(0, \infty; n) = \cup_{i=1}^a LRS(0, \infty; n, i)$. Line 7 puts $LRS(0, \infty; n, i)$ into $LRSList$ by finding $VV2$, $DV2$ and TDV of each element in $LRS(0, \infty; n, i)$. In each call of $BFSTreeGen1$, $VV2$ is extended in two ways: one is extending $Back(VV2)$ in Line 10 to Line 18; the other is adding a new vector of vertices into $VV2$ in Line 19 to Line 30. Similar to Algorithm 4, there are four steps to extend $Back(VV2)$ and four steps to add a new vector of vertices into $VV2$. To extend $Back(VV2)$, Line 10 collects eligible vertices in A , Line 12 extends $VV2$, Line 15 verifies whether $\mathcal{G}[VV2Temp]$ is connected before it is pushed into $LRSList$, and finally $BFSTreeGen2$ is called in Line 18 to further extend $VV2Temp$. To add a new vector into 2, Line 21 collects eligible vertices, Line 23 extends $VV2$, Line 27 verifies whether $\mathcal{G}[VV2Temp]$ is connected before it is pushed into $LRSList$, and finally $BFSTreeGen2$ is called in Line 30 to further extend $VV2Temp$.

See Table 3 for an example of the changes in $VV2Temp$, $DV2Temp$, $TDV(Temp)$ and A over the process of $BFSTreeGen2(M, T, 4, ((1)), ((0)), (0), true, LRSList)$ in Line 7 of Algorithm 5, in which M is the adjacency matrix and T is the vertex type vector of the graph in Fig. 4(a).

4.3. Enumeration of elements in $LRS(x, y; n)$

As mentioned earlier, we can enumerate the elements in $LRS(x, y; n)$ by enumerating elements in $LRS(\infty, \infty; n)$ using Algorithms 4 and 5, and then using Algorithm 3 to check whether these elements belong to $LRS(x, y; n)$.

Algorithm 5 Enumeration of Elements in $LRS(0, \infty; n)$

Input: The adjacency matrix M of $\mathcal{G}$, the vertex type vector T of $\mathcal{G}$, a non-negative integer n limiting the size of subgraphs.*Output:* A vector of vectors of vertices $LRSList$ indicating elements in $LRS(0, \infty; n)$.

```

1: function  $LRS_{OINENUM}(M, T, n)$ 
2:   Set  $LRSList, VV2, DV2$  and  $TDV$  as empty vectors
3:   for each  $i \in V_a$  do
4:      $VV2 = ((i))$ 
5:      $DV2 = ((0))$ 
6:      $DV = (0)$ 
7:    $BFST_{REEGEN2}(M, T, n, VV2, DV2, TDV, \text{true}, \&LRSList)$ 
8:   return  $LRSList$ 
9: function  $BFST_{REEGEN2}(M, T, n, VV2, DV2, TDV, flag, \&LRSList)$ 
10:   $A = \{j \mid C_{AT2}(VV2, j), C_{AT2}(DV2, B_{ACK}(B_{ACK}(DV2)) + 1) \text{ and } TDV \text{ satisfy conditions in Proposition 8 including Condition 6 (given in Proposition 7) except possibly Condition 5a}\}$ 
11:  for each  $B \in \{B \mid B \subseteq A, 0 < |B| \leq n - L_{ENGTH2}(VV2), B \text{ elements in ascending order.}\}$  do
12:     $VV2Temp = C_{AT2}(VV2, B)$ 
13:     $DV2Temp = C_{AT2}(DV2, R_{EP}(B_{ACK}(B_{ACK}(DV2)) + 1, |B|))$ 
14:     $flagTemp = \text{whether } VV2Temp, DV2Temp \text{ and } TDV \text{ satisfy conditions in Proposition 8}$ 
15:    if  $flagTemp = \text{true}$  then
16:       $P_{USH}(LRSList, VV2Temp)$ 
17:      if  $L_{ENGTH2}(VV2Temp) < n$  then
18:         $BFST_{REEGEN2}(M, T, n, VV2Temp, DV2Temp, TDV, flagTemp, \&LRSList)$ 
19:        if  $flag = \text{true}$  and  $1 < L_{ENGTH2}(VV2) < n$  then
20:          for each  $depth \in \{B_{ACK}(TDV), BACK(TDV) + 1\} \setminus \{0\}$  do
21:             $A = \{j \mid C_{AT}(VV2, (j)), C_{AT}(DV2, (0)) \text{ and } C_{AT}(TDV, depth) \text{ satisfy conditions in Proposition 8 including Condition 6 (given in Proposition 7) except possibly Condition 5a}\}$ 
22:            for each  $a \in A$  do
23:               $VV2Temp = C_{AT}(VV2, (a))$ 
24:               $DV2Temp = C_{AT}(DV2, (0))$ 
25:               $TDVTemp = C_{AT}(TDV, depth)$ 
26:               $flagTemp = \text{whether } VV2Temp, DV2Temp \text{ and } TDVTemp \text{ satisfy conditions in Proposition 8}$ 
27:              if  $flagTemp = \text{true}$  then
28:                 $P_{USH}(LRSList, VV2Temp)$ 
29:                if  $L_{ENGTH2}(VV2Temp) < n$  then
30:                   $BFST_{REEGEN2}(M, T, n, VV2Temp, DV2Temp, TDVTemp, flagTemp, \&LRSList)$ 

```

Algorithms 4 and 5, and then using Algorithm 3 to check whether these elements belong to $LRS(x, y; n)$.

5. Algorithm efficiency and simulations

5.1. Comparisons of approaches

In this section, we want to compare the efficiency of the following approaches to enumerate the elements in $LRS(x, y, n)$:

Approach I	The approach proposed in Section 4.3.
Approach II	A naive combinatorial approach which enumerates all subsets of V with size no larger than n and then uses Algorithm 3 to verify whether they belong to $LRS(x, y, n)$.
Approach III	An extension of the approach proposed in [14] which enumerates all cycles with length no larger than x and chains with length no larger than y , and then combines chains and cycles with common vertices to get LR subsets.

Approach I enumerates elements in $LRS(\infty, \infty, n)$ first and then checks whether they belongs to $LRS(x, y, n)$ using Algorithm 3. Because the computational complexity of Algorithm 3 does not depend on x and y , the computational complexity of Approach I does not depend on x and y . For the same reason, the computational complexity of Approach II is also independent of x and y . Approach III is a combinatorial method based on cycles and chains, as opposed to vertices in Approach II, and so the algorithmic complexity of Approach III increases as x and y increase. The smallest x and y used in practice are 3 and 2 respectively. If Approach I is more efficient than Approach II and Approach III when $x = 3$ and $y = 2$, then it follows that Approach I performs the best among the three for any $x \geq 3$ and $y \geq 2$.

Simulations were conducted to compare the efficiency of the three Approaches. We generated 500 KPD programs, where 10% of vertices are NDDs, by randomly drawing NDDs and pairs with replacement from a combined database of pairs and NDDs from the Alliance for Paired Donation (APD) and the University of Michigan (UM) KPD programs. There are 585 pairs and 56 NDDs in the APD data and 281 pairs and 7 NDDs in the University of Michigan KPD program. In both data sources, blood type and anti-HLA antibodies of each candidate, and blood type and HLA antigens of each donor are recorded. Those characteristics are used to generate edges for the simulated KPD programs. We find that 18.4% of all donor– candidate combinations correspond to potential transplants. The adjacency matrix and the vertex type vector for all NDDs and pairs in the database used in the simulation are given in the online appendix. The three Approaches were applied to search for elements in $LRS(3, 2; n)$ for various values of n and N , where N is the size of the KPD pool.

Approach II and Approach III were slow and terminated if the runtime exceeded 2 h. No time limit was imposed on Approach I. Fig. 6 shows that Approach I is the most efficient among the three approaches. Approach II considers too many unnecessary cases and Approach III suffers from an excessive number of cycles and chains for which duplication must be eliminated. Fig. 7 illustrates that the number of cycles and chains in a KPD is typically much greater than the size of the KPD.

5.2. Algorithms using back-edges

As noted earlier, an alternative approach to enumerating $LRS(x, y; n)$ is to include back-edges from each candidate pair to each altruistic donor and then enumerate $LRS(\infty, 0; n)$ in the resulting graph, which can be performed using only Algorithm 4. If there is only one NDD, this would give all cycles and chains automatically. More generally, however, it will include structures like those in Fig. 3 and more complex ones. We can proceed by determining whether each subgraph obtained after dropping the back-edges is $LR(x, y)$ using Algorithm 3. Although there are many more sub-graphs to check this way, it avoids the more complicated Algorithm 5 that enumerates $LRS(0, \infty; n)$. In fact, this approach appears to be quite competitive with the combined use of Algorithms 4 and 5 to enumerate $LRS(\infty, \infty; n)$. Table 4 compares the runtime to complete enumeration of $LRS(x, y; n)$ for these two approaches in KPDs of moderate to large sizes and the differences tend to be relatively small, at least in the cases considered.

6. Concluding remarks

In this paper, we propose and evaluate an efficient algorithm to enumerate the elements of $LRS(x, y; n)$. The comparison of our new algorithm to two existing algorithms indicates considerable computational efficiency gain. It is possible to further reduce the runtime of the algorithm by executing in parallel the content of each iteration in the for loops in Line 3 of Algorithm 4 and Line 3 of Algorithm 5.

As noted, the use of LR subgraphs yields considerable advantage over selecting cycles and chains when the size n of the LRS is restricted to 3 or 4. Further, as reported by [13] it is found that $LRS(x, y; 4)$ yields many more transplants than $LRS(x, y; 3)$. We are currently investigating the extent to which larger values of $n > 4$ would further increase the number of realized transplants in moderate sized KPD programs.

It is the presence of uncertainty that leads to the advantages of selecting subsets and taking into account possible fallbacks (or recourse), and in formulation of KPD programs in this paper, we described a probability structure for the viability of potential transplants. Typically we have relatively little information upon which to estimate these probabilities and only rough estimates are available. In simulations we have done, we have noted that the solution obtained under one probability model are nearly optimal under other models that are not too different from that assumed. This robustness suggests that simply recognizing the uncertainty and building it into the problem, even if not completely accurate, gives advantage in the solutions obtained. Note that whatever the probability model, the enumeration problem is the same.

As shown in Table 1, there are four steps to arrange transplants in the setting of LR subsets. With this new approach presented in this paper, Step (1b) can be effectively implemented with larger KPD programs and larger LR subset size limits. There still are, however, substantial computational issues in Step (2b) and Step (3b). In [16], we have investigated efficient methods to evaluate the expected utilities required in Step (2b). The IP problem that corresponds to Step (3b) becomes more complex as the number of potential sets to select increases. It is of interest in future research to develop ways to achieve near optimal

solutions to Step (3b); for example, one might strategically divide the vertices into separate groups, each of which can be managed independently. Here the aim is to define the groups so that the separate optimizations yield a nearly optimal overall solution.

When interest centers on cycles of length x and chains of length y , we have restricted our attention to the class $LRS(x, y, n)$. More generally, we could have considered the whole class of all subgraphs $\mathcal{G}[S]$ with $|S| = n$ and solved the corresponding IP problem to find the optimal selection of disjoint subsets in this larger class. But there is no advantage to doing this. This follows since any subgraph $\mathcal{G}[S] \notin LRS(x, y, n)$ can be divided into two or more disjoint parts each of which is in $LRS(x, y, n)$ but without losing any cycles or chains of interest. Thus the more general formulation with all vertex-induced subgraphs cannot increase the total expected utility obtained, and it is sufficient, and considerably more efficient, to restrict attention to the LR subgraphs considered here.

The reference [14] makes distinction between internal recourse and subset recourse, and a similar distinction is made in [13] between fallbacks and extended fallbacks. Subset recourse (or extended fallbacks) is essentially the same as locally relevant subsets whereas internal recourse (or fallbacks) refers to an approach that considers selecting cycles and chains but evaluating their expected utility by taking account of any recourse (or fallback options) available within them. Compared to internal recourse, subset recourse was found to have substantial advantage in the number of transplants achieved in [13].

Supplementary Material

Refer to Web version on PubMed Central for supplementary material.

Acknowledgments

The authors thank the Associate Editor and an anonymous reviewer for constructive comments on a previous draft of this paper that helped to improve the presentation. This research was supported by the National Institutes of Diabetes and Digestive and Kidney Diseases (NIDDK), USA through grant number 1R01-DK093513. We thank Dr Michael A. Rees and Alan B. Leichtman for access to the APD and UM KPD data respectively.

Appendix A.

Notation

The following notation is used in the article:

- $\mathcal{G} = (V, E)$ is a directed graph with vertex set V and edge set E ;
- $V = V_a \cup V_p$, where V_a is NDD set and V_p is pair set;
- $V(\mathcal{G})$ is the vertex set for the graph $\mathcal{G}$;
- $E(\mathcal{G})$ is the edge set for the graph $\mathcal{G}$;
- $\mathcal{G}[S]$ is the subgraph of $\mathcal{G}$ induced by $S \subseteq V(\mathcal{G})$;
- $\mathcal{G}^T$ is the transpose graph of $\mathcal{G}$;

- $LR(x, y)$ means a graph $\mathcal{G}$ is locally relevant with reference to cycles with length limit x and chains with length limit y ;
- $LRS(x, y, n)$ is the class of all vertex induced subgraphs which are $LR(x, y)$ and with size no larger than n ;
- BFS means breadth first search;
- $A[i]$ is the i th entry of vector A ;
- $\overset{\circ}{A}$ is the set of unique vertices in A , where A is a vector of vertices or a vector of vectors of vertices;
- $A[i, j]$ is the (i, j) entry of matrix A ;
- $\mathbb{I}(\cdot)$ is the indicator function;
- M is the adjacency matrix of graph $\mathcal{G}$, where $M[i, j] = \mathbb{I}((i, j) \in E(\mathcal{G}))$;
- T is the vertex type vector of graph $\mathcal{G}$, where $T[i] = \mathbb{I}(i \in V_a)$;
- $SP(S)$ is the matrix of the length of the shortest path in $\mathcal{G}[S]$;
- $\overline{\mathcal{G}[S]}$ and $\overline{\mathcal{G}}[S]$ are graphs derived from $\mathcal{G}[S]$, refer to Proposition 3 for details.

Appendix B.

Algorithm format

The following is standard notation:

- “ $()$ ” notation: a vector is represented by $(a_1, \dots, a_d)$;
- “ $[]$ ” notation: for a vector V , $V[k](k \in \mathbb{Z}^+)$ is the k th element in V ; for a matrix M , $M[k, l](k, l \in \mathbb{Z}^+)$ is the element in the k th row and the l th column;
- “ $\&$ ” notation: if a function is declared as “function Name ($arg1, \&arg2$)”, then the changes made to $arg2$ remain whereas those made to $arg1$ in function Name do not remain after function Name terminates.
- Function Max: input is a finite set of integers, output is the maximum element in the set;
- Function Rep: input is a variable a and a positive integer i , output is a vector of length i whose elements are all a ;
- Function Length: input is a vector, output is the length of the vector;
- Function Length2: input is a vector of vectors, output is the sum of the lengths of elements in the vector;
- Function Push: input is a vector and a variable of the same type as the elements in the vector, no output but add the variable to the end of the vector;
- Function Pop: input is a vector, no output but the last entry of the vector, if it exists, is removed from the vector;

- Function Back: input is a vector, output is the last element of the vector;
- Function Cat: input is a vector and a variable of the same type as the elements in the vector, output is a vector whose elements are elements in the input vector followed by the variable;
- Function Cat2: input is a nonempty vector of vectors V and a variable a of the same type as the elements of the vectors in V , output is identical to V except that the last vector element is $\text{Cat}(\text{Back}(V), a)$;
- Function Continue: when called inside a loop, the next iteration of the loop is forced to take place immediately.

Appendix C.

Proof of Algorithm 4

Proof. We show that the function LRSInfoEnum enumerates all elements in $\text{LRS}(\infty, 0; n)$, by proving the following statements:

- The function LRSInfoEnum terminates after a finite number of steps.
- All elements in LRSList belong to $\text{LRS}(\infty, 0; n)$.
- All elements in $\text{LRS}(\infty, 0; n)$ belong to LRSList .
- There is no duplication in LRSList .

Proof of (a): It is enough to prove BFSTreeGen terminates after a finite number of steps. We use induction on $\text{Length}(VV)$. If $\text{Length}(VV) = n$, then the set in Line 10 is empty and BFSTreeGen terminates after a finite number of steps. Assume BFSTreeGen terminates after a finite number of steps when $\text{Length}(VV) > x$ ($x \in \mathbb{Z}^+$). When $\text{Length}(VV) = x$, only Line 10 and 16 involve iteration or recursion and their termination after a finite number of steps implies BFSTreeGen terminates after finite steps. Since the set in Line 10 is finite, there are only a finite number of iterations for Line 10. Before the recursion in Line 16, Line 11 makes $VVTemp$ longer than VV , so Line 16 terminates after finite steps. By induction it follows that BFSTreeGen terminates after finite steps for all $\text{Length}(VV) = x$, $x = 1, 2, \dots$

Proof of (b): Because the only step modifying LRSList is Line 14 and it pushes $VVTemp$ into LRSList , it is enough to prove that $VVTemp$ in Line 14 belongs to $\text{LRS}(\infty, 0; n)$. By Proposition 2, it is enough to prove that in Line 14 $1 < |VVTemp| \leq n$ and both $\mathcal{G}[VVTemp]$ and $\mathcal{G}[VVTemp]^T$ have one-tree BFS forests. First, Line 13 guarantees $\mathcal{G}[VVTemp]^T$ has a one-tree BFS forest. Second, in each call of BFSTreeGen in Line 6, VV has length 1. If $A = \emptyset$, then $VVTemp$ in Line 11 must satisfy $1 < |VVTemp| \leq n$. In any recursion in Line 16, $VVTemp$ plays the role of VV and the same argument holds. Thus in Line 16 $1 < |VVTemp| \leq n$. Finally, by Proposition 7, $\mathcal{G}[VVTemp]$ has a one-tree BFS forest, if $\text{Cond}(\mathcal{G}, VVTemp, DVTemp)$ holds after Line 12. For Lines 11 and 12 of the initial call for BFSTreeGen initiated by Line 6 :

1. $VVTemp$ and $DVTemp$ are vectors obtained by extending VV and DV (which are of the same length) respectively with B and $\text{Rep}(\text{Back}(DV) + 1, |B|)$ (which are also of the same length). So $\text{Length}(VVTemp) = \text{Length}(DVTemp)$. Furthermore, there are no repeated elements in VV or B and, from Line 9, $VV \cap B = \emptyset$, so there are no repeated elements in $VVTemp$.
2. From Line 12, $DV[1] = DVTemp[1] = 0$ and $DVTemp[2] = 1$.
3. For $k = |D\dot{V}|$, $DVTemp[k+1] = \text{Back}(DV) + 1$ and $DVTemp[k] = \text{Back}(DV)$, so $DVTemp[k+1] - DVTemp[k] = 1$. Other cases are straightforward.
4. $VVTemp[1] = VV[1] = \min V\dot{V}$ and elements in B are larger than $VV[1]$ according to Line 9, so $VVTemp[1] = VV[1] = \min VVTemp$.
5. For all $|V\dot{V}| < k \leq |VVTemp|$, $\text{Cond}_5(\mathcal{G}, VVTemp, DVTemp, k)$ is true, from Line 9 and Condition 3 in Proposition 7. Other cases are easily seen.
6. For all $|V\dot{V}| < k < |VVTemp|$, $VVtemp[k+1] > VVtemp[k]$ from Condition 3 in Proposition 7 and Line 10. Other cases are trivial.

For a subsequent call of BFSTreeGen initiated by Line 16, $VVTemp$ and $DVTemp$ in the previous call become the VV and DV respectively with $\text{Cond}(\mathcal{G}, VV, DV)$ holding. Repeating the same argument as above establishes that $\text{Cond}(\mathcal{G}, VVTemp, DVTemp)$ holds after Line 12.

Proof of (c): For a subgraph $\mathcal{G}[S]$ with a one-tree BFS forest and $1 < |S| \leq n$, let V_1 and V_2 be the VV and DV vectors respectively. It is enough to prove that there exists an iteration of the for loop in Line 10 such that V_1 and V_2 are $VVTemp$ and $DVTemp$ in Lines 11 and 12. We proceed by induction on $\text{Length}(V_2)$. When $\text{Back}(V_2) = 1$, for the call of BFSTreeGen in Line 6 where $i = V_1[1]$, by Proposition 7 we have $\dot{V}_1 \setminus \{V_1[1]\} \subseteq A$ in Line 9. Again, by Proposition 7, in the iteration of the for loop in Line 10 where $B = \dot{V}_1 \setminus \{V_1[1]\}$, $VVTemp$ and $DVTemp$ are V_1 and V_2 respectively. Suppose the result holds when $\text{Back}(V_2) = x(x \geq 1)$. If $\text{Back}(V_2) = x + 1$, let V'_1 and V'_2 be the vectors obtained by deleting the last y entries of V_1 and V_2 respectively, where y is the number of elements in V_2 equal to $x + 1$. By Proposition 7, V'_1 and V'_2 are VV and DV vectors for $\mathcal{G}[\dot{V}_1]$ which has a one-tree BFS forest with $1 < |\dot{V}_1| \leq n$. If in the for loop from line 10, V'_1 and V'_2 are $VVTemp$ and $DVTemp$ respectively in Line 11 and 12, then at the next recursion at Line 16, V'_1 and V'_2 become the new VV and DV respectively. By Proposition 7, $\{V_1[k] \mid 1 \leq k \leq |\dot{V}_1|, V_2[k] = x + 1\} \subseteq A$ and $\exists B = \{V_1[k] \mid 1 \leq k \leq |\dot{V}_1|, V_2[k] = x + 1\}$ in Line 10 and in this for loop $VVTemp$ and $DVTemp$ are V_1 and V_2 respectively.

Proof of (d): There are two possible ways to call BFSTreeGen : one is by Line 6 where VV must be of length 1, and the other is by Line 16 where VV (which is $VVTemp$ in the call involving this line) is of length larger than 1 (which is proved in (b)). For all $\mathcal{G}[S]$ with a one-tree BFS forest and $1 < |S| \leq n$, it is obvious that the VV and DV vectors (denoted by V_1

and V_2 respectively) are unique. It is enough to prove that there is only one iteration of the for loop in Line 10 among all calls for BFSTreeGen where $VTemp$ and $DVTemp$ are V_1 and V_2 in Line 11 and 12 respectively. We proceed by induction on $\text{Back}(DVTemp)$. When $\text{Back}(DVTemp) = 1$, in such iteration(s), VV must be $(V_1[1])$, so such iteration(s) must be in call(s) initiated by Line 6 and obviously such an iteration appears only once. Assume that when $\text{Back}(DVTemp) = x(x > 1)$ such an iteration only appears once. When $\text{Back}(DVTemp) = x + 1$, in such iteration(s) VV and DV must be V'_1 and V'_2 defined the same as in (c).

Because $\text{Back}(V'_2) = x$, such an iteration must be in a call for BFSTreeGen initiated by Line 16 and appears only once. $\square$

Appendix D.

Supplementary data

Supplementary material related to this article can be found online at <https://doi.org/10.1016/j.orhc.2018.10.002>.

References

- [1]. Park K, Moon JI, Kim SI, Kim YS, Exchange donor program in kidney transplantation, *Transplantation* 67 (2) (1999) 336–338. [PubMed: 10075605]
- [2]. Rapaport FT, The case for a living emotionally related international kidney donor exchange registry, in: *Transplantation Proceedings*, Vol. 18, 1986, pp. 5–9.
- [3]. Ross LF, Rubin DT, Siegler M, Josephson MA, Thistlethwaite JR, Woodle ES, Ethics of a paired-kidney-exchange program, *New Engl. J. Med* 336 (24) (1997) 1752–1755. [PubMed: 9180096]
- [4]. Wang W, Bray M, Song PX-K, Kalbfleisch J, A look-ahead strategy for non-directed donors in kidney paired donation, *Stat. Biosci* 10.1007/s12561-016-9155-y.
- [5]. Ashlagi I, Gilchrist DS, Roth AE, Rees MA, Nonsimultaneous chains and dominos in kidney-paired donation revisited, *Am. J. Transplantat* 11 (5) (2011) 984–994.
- [6]. Constantino M, Klimentova X, Viana A, Rais A, New insights on integer-programming models for the kidney exchange problem, *European J. Oper. Res* 231 (1) (2013) 57–68.
- [7]. Abraham DJ, Blum A, Sandholm T, Clearing algorithms for barter exchange markets: enabling on electronic commerce (2007), 295–304.
- [8]. Åæenver MU, Roth A, et al. Efficient kidney exchange: coincidence of wants in a markets with compatibility-based preferences, *Tech. rep.*, 2007.
- [9]. Dickerson JP, Procaccia AD, Sandholm T, Failure-aware kidney exchange, in: *Proceedings of the Fourteenth ACM Conference on Electronic Commerce*, ACM, 2013, pp. 323–340.
- [10]. Pedroso JP, Maximizing expectation on vertex-disjoint cycle packing, in: *International Conference on Computational Science and its Applications*, Springer, 2014, pp. 32–46.
- [11]. Alvelos F, Klimentova X, Rais A, Viana A, A compact formulation for maximizing the expected number of transplants in kidney exchange programs, in: *Journal of Physics: Conference Series*, Vol. 616, IOP Publishing, 2015, p. 012011.
- [12]. Li Y, Kalbfleisch J, Song PX, Zhou Y, Leichtman A, Rees M, Optimization and simulation of an evolving kidney paired donation (kpd) program
- [13]. Bray M, Wang W, Song PX-K, Leichtman A, Rees M, Ashby V, Eikstadt R, Goulding A, Kalbfleisch J, Planning for uncertainty and fallbacks can increase the number of transplants in a kidney-paired donation program, *Am. J. Transplantat* 15 (10) (2015) 2636–2645.
- [14]. Klimentova X, Pedroso JP, Viana A, Maximising expectation of the number of transplants in kidney exchange programmes, *Comput. Oper. Res* 73 (2016) 1–11.

- [15]. Pedroso JP, Ikeda S, Maximum-expectation matching under recourse, arXiv preprint arXiv: 1605.08616.
- [16]. Bray M, Wang W, Song PXX, Kalbfleisch JD, Valuing sets of potential transplants in a kidney paired donation network, Stat. Biosci (2018) 1–25.
- [17]. Moore EF, The Shortest Path Through a Maze, Bell Telephone System, 1959.
- [18]. Sharir M, A strong-connectivity algorithm and its applications in data flow analysis, Comput. Math. Appl 7 (1) (1981) 67–72.

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

Fig. 1.

Fig. 1(a) is an example of a cycle of length 3 consisting of candidate-donor pairs $C_1 \& D_1$, $C_2 \& D_2$, $C_3 \& D_3$. Fig. 1(b) is an example of a chain of length 2 consisting of an NDD and candidate-donor pairs $C_1 \& D_1$, $C_2 \& D_2$. Compatibility is denoted by arrows pointing from donors to recipients.

Fig. 2.
A KPD program with no NDDs represented by $\mathcal{G}(V, E)$. Vertices represent pairs in V and arrows represents edges in E .

Fig. 3. A simple directed graph $\mathcal{G}(V, E)$, which is not locally relevant, despite the back-edge formulation recognizing it as a cycle. Black vertices represent NDDs and white vertices represent pairs. Solid arrows represent edges in E and dotted arrows represent back-edges.

Fig. 4.

(a) A vertex-induced subgraph, $\mathcal{G}[S]$ where $\mathcal{G}$ is not shown, is $\text{LR}(0, \infty)$. Black vertices represent NDDs and white vertices represent pairs. Vertices are numbered according to the default order. (b) Rectangles and dashed arrows represent respectively vertices and edges of graph $\overline{\mathcal{G}[S]}$. In each vertex of graph $\overline{\mathcal{G}[S]}$, there is a number indicating its default order and a tree indicating the corresponding BFS tree of $\mathcal{G}[S]$ in Fig. 4(a). (c) Rectangles and dashed lines represent respectively vertices and edges of the graph $\overline{\mathcal{G}[S]}$. In each vertex of graph $\overline{\mathcal{G}[S]}$, there is a number indicating its default order and a tree indicating the corresponding BFS tree of $\mathcal{G}[S]$ in Fig. 4(a). (d) BFS forest of graph $\overline{\mathcal{G}[S]}$ in Fig. 4(c).

Fig. 5.
 $\mathcal{G}[S]_{3,2}$ of $\mathcal{G}[S]$ in Fig. 4(a).

Fig. 6.

Mean runtime to enumerate the elements in $\text{LRS}(3, 2; n)$ in simulated KPD programs with N vertices where 10% of vertices are NDDs. Note that if the time exceeded 2 h in Approach II and Approach III, the time was recorded as two hours. There was no time limit imposed on Approach I. The only points plotted are those for which the average time is under 2 h. The missing plotting points following the last point on each curve are above 120 min.

Fig. 7. Log mean number of cycles and chains, with length no larger than x and y respectively, in simulated KPD programs with N vertices where 10% of vertices are NDDs.

Table 1

Steps to conduct a match run in a KPD program on the basis of (a) chains and cycles and (b) locally relevant subgraphs.

Step	(a) A match run on the basis of chains and cycles	(b) A match run on the basis of locally relevant subgraphs
1	Enumerate all chains and cycles with lengths no larger than a given number	Enumerate all locally relevant subgraphs with sizes no larger than a given number
2	Calculate the number of potential transplants for each chain and cycle	Calculate the expected number of transplants for each locally relevant subgraph, taking account of fallback options
3	Determine the set of disjoint chains and cycles that maximizes the number of potential transplants	Determine the set of disjoint locally relevant subgraphs that maximizes the expected number of transplants
4	Determine which vertices and edges are viable, and carry out transplants in selected chains and cycles	Determine which vertices and edges are viable, and carry out transplants in each selected subgraph to maximize the number of transplants

Table 2

Listed in chronological order are the changes in $VVTemp$, $DVTemp$ and eligible vertices set A over the process of $LRSINFOENUM(M, T, 4)$ in Line 6 of Algorithm 4, in which M and T are respectively the adjacency matrix and the vertex type vector of the graph in Fig. 2.

$VVTemp$	$DVTemp$	A	Whether $\mathcal{G}[VVTemp]$ has a one-tree BFS forest
(1, 3)	(0, 1)	{3}	False
(1, 3, 2)	(0, 1, 2)	{2, 4, 6}	True
(1, 3, 2, 5)	(0, 1, 2, 3)	{5}	False
(1, 3, 4)	(0, 1, 2)	{2, 4, 6}	False
(1, 3, 6)	(0, 1, 2)	{2, 4, 6}	False
(1, 3, 2, 4)	(0, 1, 2, 2)	{2, 4, 6}	True
(1, 3, 2, 6)	(0, 1, 2, 2)	{2, 4, 6}	False
(1, 3, 4, 6)	(0, 1, 2, 2)	{2, 4, 6}	False
(2, 5)	(0, 1)	{5}	False
(2, 5, 6)	(0, 1, 2)	{6}	False
(2, 5, 6, 4)	(0, 1, 2, 3)	{4}	True
(3, 4)	(0, 1)	{4, 6}	True
(3, 6)	(0, 1)	{4, 6}	False
(3, 4, 6)	(0, 1, 1)	{4, 6}	True
(5, 6)	(0, 1)	{6}	False
$LRS(\infty, 0; 4) = \{(1, 3, 2), \{1, 3, 2, 4\}, \{2, 5, 6, 4\}, \{3, 4\}, \{3, 4, 6\}\}$			

Table 3

Table of updates for subgraph displayed in Fig. 4(a). Listed in chronological order are the changes in VV , $2Temp$, $DV2Temp$, $TDV(Temp)$, A and $flagTemp$ over the process of $BFSTreeGen2(M, T, 4, ((1)), ((0)), (0), true, LRSList)$ in Line 7 of Algorithm 5, in which M is the adjacency matrix and T is the vertex type vector of the graph in Fig. 4(a). The algorithm continues to call $BFSTreeGen2(M, T, 4, ((i)), ((0)), (0), true, LRSList)$ for $1 < i \leq a$ and finally yields $LRS(0, \infty; 4)$.

<i>VV 2-Temp</i>	<i>DV 2-Temp</i>	<i>TDV -(Temp)</i>	<i>A</i>	<i>flag-Temp</i>
((1, 6))	((0, 1))	(0)	{6, 7}	True
((1, 6, 5))	((0, 1, 2))	(0)	{5}	True
((1, 6, 5), (2))	((0, 1, 2), (0))	(0, 1)	{2, 3}	False
((1, 6, 5), (3))	((0, 1, 2), (0))	(0, 1)	{2, 3}	True
((1, 6), (2))	((0, 1), (0))	(0, 1)	{2, 3}	False
((1, 6), (2, 4))	((0, 1), (0, 1))	(0, 1)	{4}	False
((1, 6), (3))	((0, 1), (0))	(0, 1)	{2, 3}	False
((1, 6), (3, 4))	((0, 1), (0, 1))	(0, 1)	{4}	False
((1, 7))	((0, 1))	(0)	{6, 7}	True
((1, 7), (2))	((0, 1), (0))	(0, 1)	{2, 3}	False
((1, 7), (2, 4))	((0, 1), (0, 1))	(0, 1)	{4}	True
((1, 7), (3))	((0, 1), (0))	(0, 1)	{2, 3}	False
((1, 7), (3, 4))	((0, 1), (0, 1))	(0, 1)	{4, 5}	True
((1, 7), (3, 5))	((0, 1), (0, 1))	(0, 1)	{4, 5}	True
((1, 6, 7))	((0, 1, 1))	(0)	{6, 7}	True
((1, 6, 7, 5))	((0, 1, 1, 2))	(0)	{5}	True
((1, 6, 7), (2))	((0, 1, 1), (0))	(0, 1)	{2, 3}	False
((1, 6, 7), (3))	((0, 1, 1), (0))	(0, 1)	{2, 3}	False

LRS(0, ∞; 4, 1)^a=

{ {1, 6}, {1, 6, 5}, {1, 6, 5, 3}, {1, 7}, {1, 7, 2, 4}, {1, 7, 3, 4}, {1, 7, 3, 5}, {1, 6, 7}, {1, 6, 7, 5} }

^a $LRS(0, \infty; 4, 1)$ is the set of elements in $LRS(0, \infty; 4)$ that contains vertex 1.

Table 4

Average runtime in seconds based on 500 simulations for enumerations of $LRS(x, y, n)$ for various values of n and KPD size, N . Ave time 1 indicates the average time for Algorithms 4 and 5; Ave time 2 is the average time for Algorithm 4 using back-edges. The runtime depends only moderately on x and y , so we have reported the time averaged over values of x and y . The KPD programs were generated from the APD data with 10% NDDs.

n	N	Ave time 1	Ave time 2
4	100	0.98	0.94
4	200	17.0	24.0
4	400	265.2	273.7
5	100	16.5	16.0
5	200	558.2	604.5
5	400	17478.0	19179.0
6	100	308.5	242.0
6	200	17147.0	15390.0