Neural Computing and Applications (2024) 36:15373–15388
https://doi.org/10.1007/s00521-024-09887-5

ORIGINAL ARTICLE

# Solving the kidney exchange problem via graph neural networks with no supervision

Pedro F. Pimenta[1] · Pedro H. C. Avelar[2,3] · Luís C. Lamb[1]

Received: 27 November 2023 / Accepted: 23 April 2024 / Published online: 15 May 2024

**Abstract**
This paper introduces a new learning-based approach for approximately solving the Kidney-Exchange Problem (KEP), an NP-hard problem on graphs. The KEP consists of, given a pool of kidney donors and patients waiting for kidney donations, optimally selecting a set of donations to optimize the quantity and quality of transplants performed while respecting a set of constraints about the arrangement of these donations. The proposed technique consists of two major steps: the first is a Graph Neural Network (GNN) trained without supervision; the second is a deterministic non-learned search heuristic that uses the output of the GNN to find a valid solution. To allow for comparisons, we also implemented and tested an exact solution method using integer programming, two greedy search heuristics without the machine learning module, and the GNN alone without a heuristic. We analyze and compare the methods and conclude that the learning-based two-stage approach is the best solution quality, outputting approximate solutions on average 1.1 times more valuable than the ones from the deterministic heuristic alone.

**Keywords** Kidney exchange problem · Graph neural networks · Optimization · Machine learning · Deep learning · Graph theory and applications

## 1 Introduction

This study addresses machine learning approaches for the approximate solving of the Kidney Exchange Problem (KEP), an NP-Hard problem on graphs [1, 2]. This problem consists of, given a pool of kidney donors and patients waiting for kidney donations, optimally selecting a set of donations to optimize the quantity and quality of transplants performed while still respecting a set of constraints about the arrangement of these donations.

✉ Pedro F. Pimenta
pfpimenta@inf.ufrgs.br

✉ Luís C. Lamb
luislamb@acm.org

Pedro H. C. Avelar
pedro_henrique.da_costa_avelar@kcl.ac.uk

1 Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil

2 King's College, London, UK

3 A*STAR, Singapore, Singapore

Thus, this work's main objective is to answer the following question: Can the Kidney Exchange problem be better approximately solved with the help of machine learning? If positive, we want to evaluate the feasibility of utilizing such an approach in terms of the quality of the solutions it provides. Further, we are also interested in assessing how viable would such a method be in terms of computational time.

Additionally, one hopes that, by answering these questions, we may also better understand the limitations of the employed machine learning methods for this problem and the potential future research directions for solving the KEP and other optimization problems in graphs.

### 1.1 On kidney exchanges

Kidney disease impacts millions of people worldwide, and the two available treatment options for end-stage kidney disease are dialysis and kidney transplantation [3]. Transplantation is the preferred treatment for the most severe forms of kidney disease [1] because it is cheaper and offers a better quality of life and better life expectancy [4]. The source of the kidney can be either a cadaver or a live donor, as the human body has two kidneys, and often only one suffices.

The compatibility of a transplant between a donor and a recipient is determined by a number of different factors, such as the blood group compatibility, tissue-type compatibility, the ages and general health of the donor and the recipient, the size of the donor's kidney, and many others [3]. The lower the compatibility between a donor and a patient, the lower the chance of kidney transplant success between them.

In the last decades, there have started to be *paired kidney exchanges*, which are cycles involving donor–patient pairs such that each donor cannot give a kidney to their intended recipient because of some incompatibility. However, each patient can receive a kidney from a donor from another pair [1]. These cycles were first performed with only two donor–patient pair nodes, but later longer cycles of kidney exchanges were performed. Another possibility of an exchange scheme is to create exchange chains that begin with a donation of an altruistic or cadaveric kidney donor, followed by chained donations of patient–donor pairs, and then finish with a donation either to a patient with or without an associated donor. In this study, the donation chains are also referred to as *paths*, a term often used for describing sequences of connected nodes in graph problems. To find the best possible allocation, i.e., the optimal solution to the problem, considering a set of donors, patients, and patient–donor pairs, a mix of both cycles and chains can be selected as long as the cycles and paths do not intersect with each other.

These cycles and chains could have unlimited size. In real life, however, there is a practical limit to the size of the paired kidney exchange cycles and chains: The kidney donation surgeries in a chain often must be done simultaneously to ensure every patient receives a kidney before her associated donor donates her kidney. However, organizing many simultaneous surgeries is logistically complex and sometimes impractical or even impossible. Even if they do not have to be done simultaneously, it is generally required at least that every patient–donor pair receive a kidney before they give a kidney. Furthermore, numerous other logistical difficulties arise when dealing with longer cycles and chains, which makes it highly desirable or sometimes even necessary that these donation cycles and chains have limited size. The longest kidney transplant chain successfully performed had a size of 35 and happened between 6 January and 17 June 2015 in the USA [5], although, in most situations, the maximum reasonable size is considerably smaller.

## 1.2 The kidney exchange problem

The Kidney Exchange Problem (KEP) was first mathematically formalized by Roth et al. in [1], then slightly updated in various ways in subsequent works. A summary of the variations found in the literature and models and techniques currently employed to solve them can be found at [6].

In the formalization used in the study, each instance of the KEP is represented by a directed weighted graph $G = \{V, E\}$. Each patient, donor, and patient–donor pair is mapped to a graph node; they will be referred to as patient ($P$) nodes, non-directed (or altruistic) donor (NDD) nodes, and patient–donor pair (PDP) nodes. The set of nodes $V$ is thus accordingly partitioned into sets $P$, $NDD$ and $PDP$. The graph's edges represent donation compatibility: an edge from node A to node B represents that a kidney donation in this direction is possible; the edge weight encodes the donor's compatibility with the recipient.

Solving a KEP instance means optimally selecting a set of cycles and chains to optimize the transplants performed. This includes maximizing not only the quantity but also the quality of the transplants, which is encoded in the edge weights. The solution must also respect a set of constraints. Each node may participate at most in one transplant as a donor and another as a receiver. Also, PDP nodes can only donate a kidney if they receive one, although they can receive it without donating. Nodes of type $P$ can only receive donations, and NDD nodes can only donate. The problem is then described as follows: "*Given a list of kidney needing patients, kidney donors, and patient–donor pairs, and a compatibility index between each possible donor and receiver, what is the best possible selection of donations that can be performed so that the total quantity of transplants, weighted by the compatibility indexes, is maximized, while still respecting a given size limit to the kidney exchange cycles and chains in the solution?*"

Our KEP formalization is represented in the set of equations below, inspired on the so-called Recursive Algorithm formulation described by [2]. We use a binary variable $y_e$ for each edge $e \in E$, that indicates if the edge is part of the solution or not, as well as auxiliary variables flow in $f_v^i$ and flow out $f_v^o$ for each node $v \in V$, which represents the node's number of incoming and outcoming edges contained in the solution, and are defined at Eqs. 9 and 10. $\mathcal{C}$ represents the set of existing cycles and paths in graph, where $C \in \mathcal{C}$ is a collection of edges, i.e., $C \subset E$. $\mathcal{C}_k$ is a subset of $\mathcal{C}$ (i.e., $\mathcal{C}_k \subset \mathcal{C}$) containing the cycles and paths that use $k$ or fewer edges.

$$\max \sum_{e \in E} w_e y_e \tag{1}$$

$$\text{s.t.} \sum_{e \in \mathcal{N}in(v)} y_e = f_v^i \quad v \in V \tag{2}$$

$$\sum_{e \in \mathcal{N}in(v)} y_e = f_v^o \quad v \in V \tag{3}$$

$$f_v^o \le f_v^i \le 1 \quad v \in PDP \tag{4}$$

$$f_v^o \le 1 \quad v \in NDD \tag{5}$$

$$f_v^i \le 1 \quad v \in P \tag{6}$$

$$\sum_{e \in C} y_e \le |C| - 1 \quad C \in \mathcal{C} \setminus \mathcal{C}_k \tag{7}$$

$$y_e \in \{0, 1\} \quad e \in E \tag{8}$$

$$f_v^i = \sum_{e \in \mathcal{N}in(v)} y_e \quad v \in V \tag{9}$$

$$f_v^o = \sum_{e \in \mathcal{N}out(v)} y_e \quad v \in V \tag{10}$$

$$\mathcal{N}_{\text{in}}(v) = \{e \forall e \in E, e = (v', v)\}$$
$$\mathcal{N}_{\text{out}}(v) = \{e \forall e \in E, e = (v, v')\}$$

The constraints ensure the result is a valid solution for KEP: the first two (Eqs. 2 and 3) are necessary for the use of the flow in and flow out variables, the third one (Eq. 4) controls the flow in and flow out of the PDP nodes, the fourth and fifth ones (Eqs. 5 and 6) do the same but for NDD nodes and P nodes, respectively, the sixth one (Eq. 7) prohibits cycles or paths with length longer than a given limit $k$, and the seventh one (Eq. 8) defines the domain of the $\mathbf{y}$ variable. The objective (defined at Expression 1) is to maximize the number of edges in the solution $y$, weighted by the associated edge weights $w$, while still respecting the KEP constraints (Eqs. 4, 5, 6, 7, and 8).

It has been proven that this problem is NP-Hard [7], although it can become polynomial-time solvable if some of the constraints are relaxed, such as limiting the exchange cycles and chains length to 2, or removing the length restriction entirely.

## 2 Machine learning methods for optimization problems in graphs

In the last few years, many machine learning-based approaches that effectively solve several different optimization problems in graphs have been proposed, although none of them designed for solving KEP. Graph optimization problems already solved with the help of machine learning include the Set Covering Problem [8], Graph Coloring [9, 10], Minimum Vertex Cover [11, 12], Maximum Cut [13], Graph Partitioning [14], Maximum Independent Set [15], Maximum Common Subgraph [16], and the Traveling Salesperson Problem (also called the traveling salesman problem or TSP), one of the most famous NP-hard problems, often used to represent the class, and some variants of it [17–22].

In 2015, the authors of [20] presented a new type of neural network called Pointer Networks, designed to learn how to reorder the elements of an input sequence; they validated the method by using it to solve 3 problems, including the TSP. In 2016, researchers presented in [23] a framework for combinatorial optimization problems using neural networks and reinforcement learning; the work focused on the TSP, which is a graph problem, but the approach was designed to work with any combinatorial optimization problem. In 2017, the authors of [13] proposed the utilization of a combination of graph representation learning and reinforcement learning to solve graph optimization problems; they showed that their proposed approach effectively learns to solve at least three of those problems: Minimum Vertex Cover, Maximum Cut, and TSP. In 2018, the decision variant of the TSP, called Decision Traveling Salesman Problem (DTSP), which is to decide if a given TSP instance admits a Hamiltonian route with a cost no greater than a given threshold C, and is also NP-Hard, was solved in [19] with a GNN.

In 2019, the authors of [17] tried to solve the TSP using a two-stage technique that is very similar to the one presented in this study (described at Sect. 4.3.6 and illustrated at Fig. 1): Firstly, a GNN processes the input graph and create scores for each edge of the graph; then, a non-learned search heuristic, which in this case was beam search, uses these scores to construct a solution. There have been other approaches that use similar techniques: in [24] the authors review graph learning methods for solving combinatorial optimization problems, with a focus on two-stage techniques, where the first is based on graph representation learning, which embeds the input graph into low-dimension vectors, and the second uses the embeddings learned in the first stage; [25] surveys the use of GNNs as a model of neural-symbolic computing and their applications, which includes combinatorial optimization problems; [18] unifies and refines several of such two-stage techniques for neural combinatorial optimization, and test it on the TSP.

## 3 Dataset

Deep learning methods such as GNNs require lots of data; usually, many thousands of instances, or even millions, depending on the problem and the size of the model, are necessary for the models to converge to a decent behavior. With insufficient data, the model is very prone to overfitting, or sometimes may not even learn anything at all.

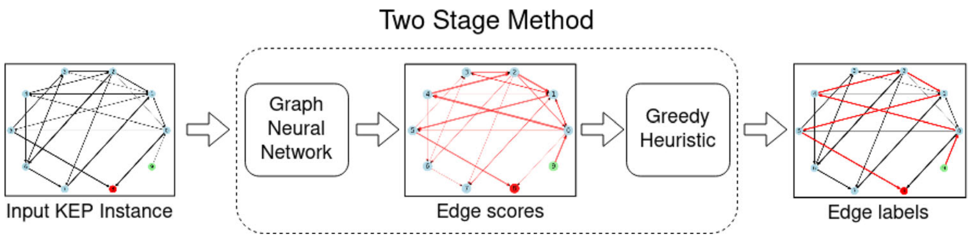

**Fig. 1** Diagram representing an overview of the *two-stage method*. The GNN takes the input KEP instance and computes a score for each edge of the graph; then, a greedy heuristic such as *GreedyCycles* or *GreedyPaths* uses these edge scores instead of the original edge weights to build an approximate solution, which is a binary label for each edge (*edge labels*), indicating if the edge is part of the approximate solution predicted or not. Source: Author

Medical data, however, is very scarce, and rarely available at this quantity. This happens for two main reasons. Firstly, a major problem with healthcare data is its sensitivity: as a lot of it is confidential information about the patients, it is usually highly protected and as a rule cannot be used without special consent, be it from the patients or at least from the health institute that owns the data. Furthermore, there are a limited number of medical cases of each given situation registered; although it would be useful to have a KEP dataset with millions of instances, this situation has not happened that many times, and not necessarily all of them have been registered digitally.

Due to the scarcity of the data, and considering that to train a machine learning model it usually takes at least tens of thousands of examples, the datasets were generated artificially. Three separate datasets were generated: the *train dataset*, with 10 thousand instances for the training of the model; the *validation dataset* with 100 instances, used for validation step during the training; and the *test dataset*, with 10 thousand instances, used to evaluate the performance of each one of the employed methods. This number of instances for the validation dataset was chosen because it was sufficiently big so that still kept roughly the same properties as the train and test datasets, but small enough that the validation does not slow down the training too much.

To generate each instance, first 300 separated nodes are created, and then 5500 edges are added sequentially linking random nodes, while still guaranteeing that no two edges connect the same two nodes in the same direction. The weight value of each edge is sampled from an uniform distribution of values between 0 and 1. To choose the number of nodes and edges of the KEP generated instances, the instances used for benchmarking in [2] were used as reference. Considering the 25 instances presented in Table S3, which they call "difficult" real data instances, the average number of nodes on a KEP instance is 265.84, and the average number of edges is 5695.92. Thus, the values for the number of nodes and number of edges chosen were 300 and 5500, respectively. The proportion of the types of nodes was also chosen to be similar to the instances: roughly 90% of the nodes are PDP nodes, 5% are NDD nodes, and 5% are P nodes.

To keep track of the instances, an unique ID was assigned to each. For this, the Weisfeiler Lehman graph hash [26] was used, which has strong guarantees that non-isomorphic graphs will get different hashes; it also considers node and edge features in its computation, thus differentiating even between graphs which have the same structure but different edge weight values.

## 4 Methods

We classified the methods for solving the KEP in 3 categories: integer programming methods, non-learnable heuristic methods, and learnable heuristic methods. All of them use the same input information, which is a KEP instance, and return the solution in the same format, which is a binary label for each edge, indicating if it is in the solution or not. All of them are evaluated on the test dataset, with the exception of the integer programming method, as later explained in Sect. 5, but only the learned heuristics use the training dataset, during their training phase.

### 4.1 Integer programming

To obtain the analytical solution, i.e., the optimal solution, the formulation presented in Sect. 1.2 was implemented using the PyCSP3 Python library [27]. There are other integer programming formulations for KEP, including two presented in [2], as well as others in [7, 28, 29]. This formulation was chosen because it is the most straightforward one.

### 4.2 Non-learnable heuristics

To evaluate the implemented methods that use machine learning, we decided to compare them to non-learnable heuristics, i.e., heuristic methods that do not use learning techniques. This section aims to describe these non-learnable heuristic methods. To the best of our knowledge, however, there are no canonical heuristics for the KEP. For this reason, we implemented two search heuristics, which are described below.

#### 4.2.1 Greedy paths

This algorithm greedily selects paths that start on NDD nodes and goes through PDP or P nodes one by one until there is no more nodes to be selected, or until a P node is reached. It starts by selecting the edge with the highest weight considering only the subset of edges that have an NDD node as source. Then, considering only the edges that come from the previous node, it follows by selecting always the next edge with the highest weight, until there are no more available edges left that would continue the path. After a path has been added to the solution, the edges connected to nodes of this path are masked, and Greedy-Paths repeats the process until no more NDD nodes with valid outgoing edges are available. This algorithm is described at Algorithm 1.

#### 4.2.2 Greedy cycles

This algorithm greedily selects cycles of PDP nodes. It starts by selecting the edge with the highest weight considering only the subset of edges that have a PDP node as source and a PDP node as destination. Then, PDP nodes are greedily added to the solution in the same way as done by the GreedyPaths method until the cycle ends, or until it arrives at a node already added in the cycle, in which case the cycle is closed and the nodes before the current node are removed from the cycle. After a cycle is added to the solution, the GreedyCycles algorithm applies a mask on the edges connected to nodes of this cycle, and then repeats the process until no more PDP nodes with valid outcoming edges are available. This algorithm is described at Algorithm 2.

**Algorithm 1** GreedyPaths

```
procedure GREEDY-PATHS($G = (N, E), k$)
    paths ← []
    while $|\mathrm{GP}(G, k)| > 0$ do
        path ← $\mathrm{GP}(G, k)$
        paths ← paths $\oplus$[path]
        $G \leftarrow (N \setminus \{n\ \forall\ n \in \text{paths}\}, E \setminus \{e\ \forall\ e \in E, src(e) = n \vee tgt(e) = n\})$
    return paths
procedure GP($G = (N, E), k$)                                  ▷ Gets one Greedy Path
    $E_{NDD} \leftarrow \{e\ \forall\ e \in E, src(e) \in NDD\}$
    if $|E_{NDD}| = 0$ then
        return []
    $e_c \leftarrow \arg\max_{e \in E_{NDD}\}} w_e$
    path ← $[src(e_c)]$
    while $|\mathrm{OE}(tgt(e_c))| > 0 \wedge |\text{path}| < (k+1)$ do
        path ← path $\oplus[tgt(e_c)]$
        $e_c \leftarrow \arg\max_{e \in \mathrm{OE}(tgt(e_c)) \setminus \bigcup_{n \in \text{path}} \mathrm{IE}(n)} w_e$
    return path
procedure OE($G = (N, E), n$)                                  ▷ Outgoing Edges
    return $\{e\ \forall\ e \in E, src(e) = n\}$
procedure IE($G = (N, E), n$)                                  ▷ Incoming Edges
    return $\{e\ \forall\ e \in E, tgt(e) = n\}$
```

**Algorithm 2** GreedyCycles

```
procedure GREEDY-CYCLES($G = (N, E), k$)
    cycles ← []
    while $|\mathrm{GC}(G, k)| > 0$ do
        cycle ← $\mathrm{GP}(G, k)$
        cycles ← cycles ⊕[cycle]
        $G \leftarrow (N \setminus \{n \ \forall \ n \in \text{cycles}\}, E \setminus \{e \ \forall \ e \in E, src(e) = n \vee tgt(e) = n\})$
    return cycles
procedure GP($G = (N, E), k$)                                   ▷ Gets one Greedy Cycle
    $E_{PDP} \leftarrow \{e \ \forall \ e \in E, src(e) \in PDP, dst(e) \in PDP\}$
    if $|E_{PDP}| = 0$ then
        return []
    $e_c \leftarrow \arg\max_{e \in E_{PDP}\}} w_e$
    cycle ← $[src(e_c)]$
    while $tgt(e_c) \notin$ cycle do
        if $|\mathrm{OE}(tgt(e_c))| \leq 0 \vee |\text{cycle}| \geq k$ then
            return []                                           ▷ Unable to close cycle (dead end)
        cycle ← cycle $\oplus[tgt(e_c)]$
        $e_c \leftarrow \arg\max_{e \in \mathrm{OE}(tgt(e_c)) \setminus \bigcup_{n \in \text{cycle}} \mathrm{IE}(n)} w_e$
    return cycle
procedure OE($G = (N, E), n$)                                   ▷ Outgoing Edges
    return $\{e \ \forall \ e \in E, src(e) = n\}$
procedure IE($G = (N, E), n$)                                   ▷ Incoming Edges
    return $\{e \ \forall \ e \in E, tgt(e) = n\}$
```

### 4.3 Learnable heuristics

As KEP instances are graphs, using GNNs to extract more detailed and abstract information can help in constructing an approximate solution. Therefore, GNN models were chosen as the main learning module for the machine learning methods.

#### 4.3.1 GNN architecture

The architecture of the GNN used in this work is the following: firstly, there is a message passing phase, where the original node features are passed through a PNA layer [30], and then through two consecutive GATv2 layers [31]; after each message passing layer, a ReLU activation function is applied, followed by a dropout regularization; next, the node features are passed through a fully connected feed forward neural network, followed by another ReLU activation function; then, the edge features are constructed by concatenating the original input edge features with the node features of the origin and destination nodes associated to each edge; these edge features are then passed through a fully connected feed forward neural network, which outputs a score for each edge; at this point, a skip connection adds the original edge weights to the edge scores; finally, a node-wise softmax operation, which is described below, is applied so as to normalize these scores in relation to the scores of other edges that share the same source node.

In order to make information flow not only in the original direction of the original edges, each message passing layer is accompanied by an associated layer, which we call *counter edge layer*, that is exactly similar, but with different learned weights, and with the difference that the information is propagated in the opposite direction, i.e., flowing from the destination node to the origin node. Each time message layers are executed, the output of the original and of the counter edge layers is concatenated before being passed to the next layers.

#### 4.3.2 KEP unsupervised loss

This loss function was designed to capture, without the need for the exact solution as a label or any other supervision, the essence of what we are trying to maximize: the sum of weights of edges that are in the predicted solution. It is defined as the log of the sum of weights of all edges of the input instance over the sum of weights of edges that are in the predicted solution, weighted by the scores predicted by the GNN. This loss function is presented in Formula 11,

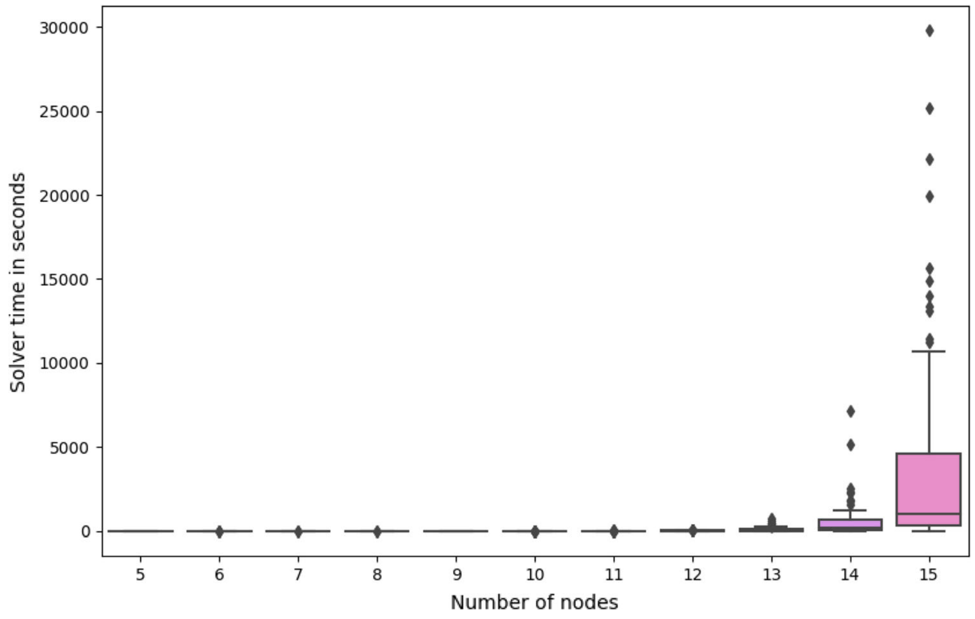

**Fig. 2** Boxplot of the time it takes to run the solver on KEP instances of sizes 5–15 (i.e., number of nodes)

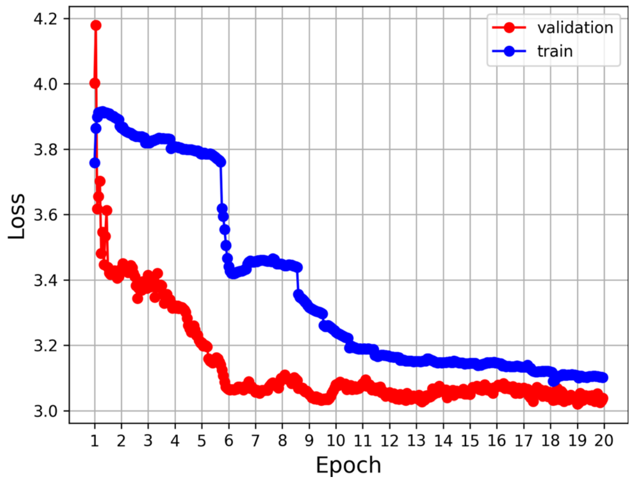

**Fig. 3** Evolution of the training and validation loss for *GNN+GreedyPaths* method

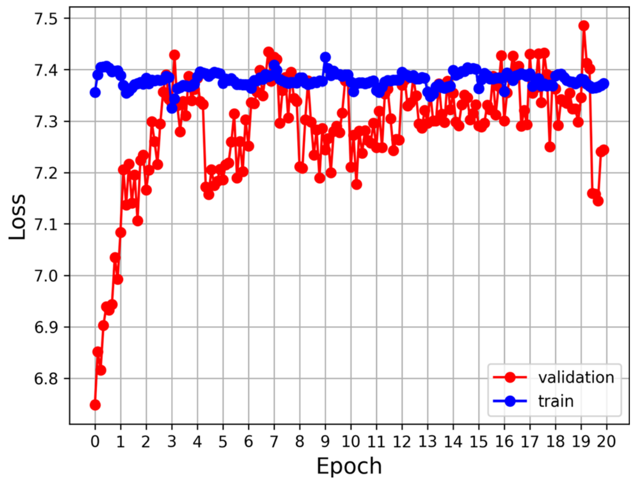

**Fig. 4** Evolution of the training and validation loss for *GNN+GreedyCycles* method

where $w$ represents the vector of edge weights, *pred* represents the vector of predicted classes (which indicates if each edge is contained in the solution or not), and $s$ represents the vector of scores attributed by the GNN for each edge.

$$\text{KEP\_Loss}(w, \text{pred}, s) = \log \frac{\sum_{e \in E} w_e}{\sum_{e \in E} w_e \text{pred}_e s_e} \tag{11}$$

### 4.3.3 Loss constraint regularization

In order to integrate information about the KEP flow constraints (Eqs. 4, 5 and 6) into the learning of the model, a loss regularization function was developed. It aims to model the restriction that each node must have at maximum one single outcoming edge and one incoming edge that are part of the solution. It is defined as the log of the division between the total quantity of edges in the solution and the number of unique nodes that appear in the solution as an origin/destination node. This makes it so that the regularization term value is proportional to the number of invalid edges in the solution, i.e., the total quantity in the graph of extra edges for each source/destination node. This function can then be added to the unsupervised loss (described in the subsection above) by summing their output values, weighted by coefficients, which become new hyperparameters of the training.

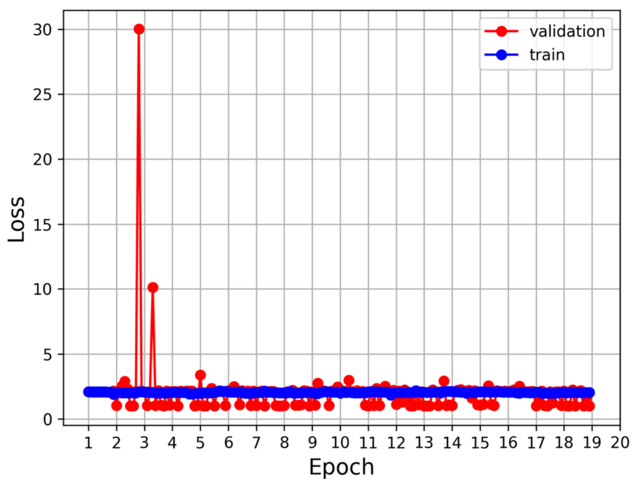

**Fig. 5** Evolution of the training and validation loss for the *Unsupervised GNN* method

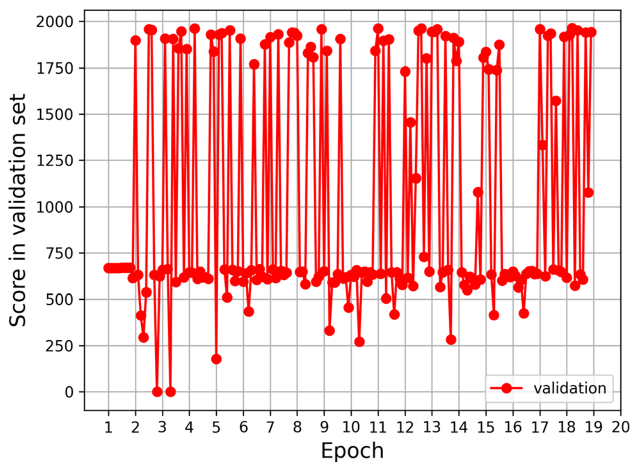

**Fig. 8** Evolution of the score measured in the validation dataset for the *Unsupervised GNN* method

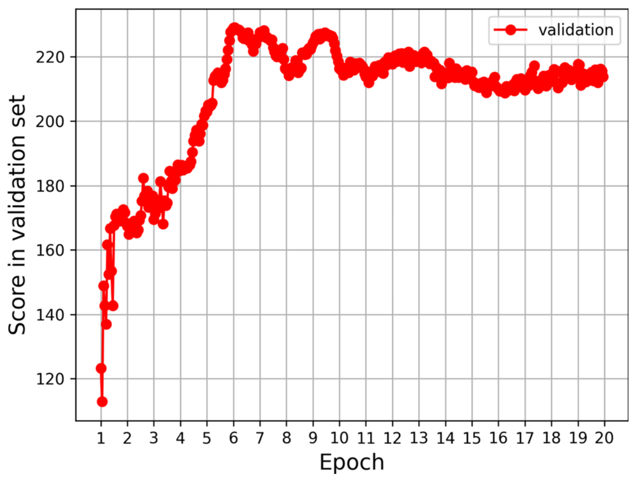

**Fig. 6** Evolution of the score measured in the validation dataset for the *GNN+GreedyPaths* method

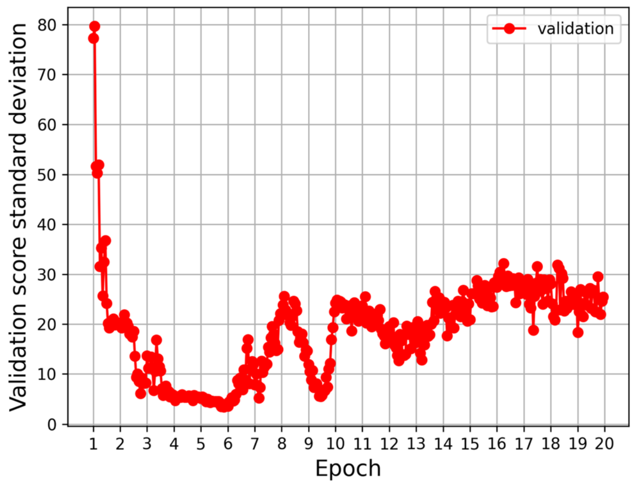

**Fig. 9** Evolution of the standard deviation of score measured in the validation dataset for the *GNN+GreedyPaths* method

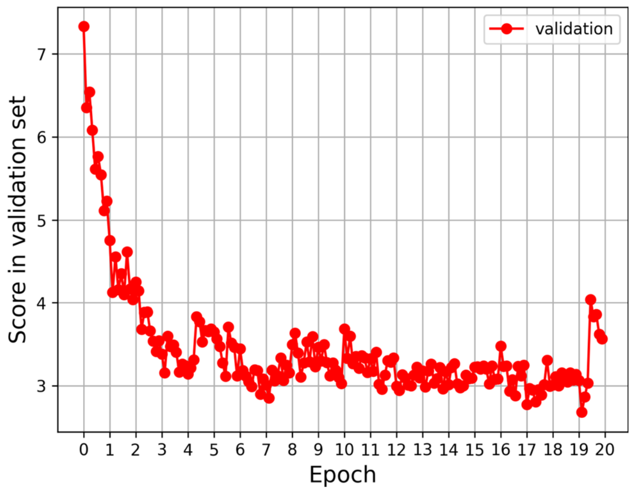

**Fig. 7** Evolution of the score measured in the validation dataset for the *GNN+GreedyCycles* method

### 4.3.4 Node-wise softmax

The node-wise softmax operation is the application of an independent softmax operation for each group of edges that share the same source node, as one can see in Eq. 12, where $s_{e_{(n_i,n_j)}}$ represents the edge score of the edge connecting node $i$ to node $j$. In this way, for each node we will have a probability for each outgoing edge; in KEP, these values may represent a probability distribution for the donation options of the donor for each source node. Although in this work the operation was used grouping edges by source node, with a simple change of a parameter it can group edges by groups of common destination node as well.

$$\text{node-wise-softmax}(s_{e_{(n_i,n_j)}}) = \frac{e^{-s_{e_{(n_i,n_j)}}}}{\sum_{n_k \in \mathcal{N}(n_i)} e^{-s_{e_{(n_i,n_k)}}}} \qquad (12)$$

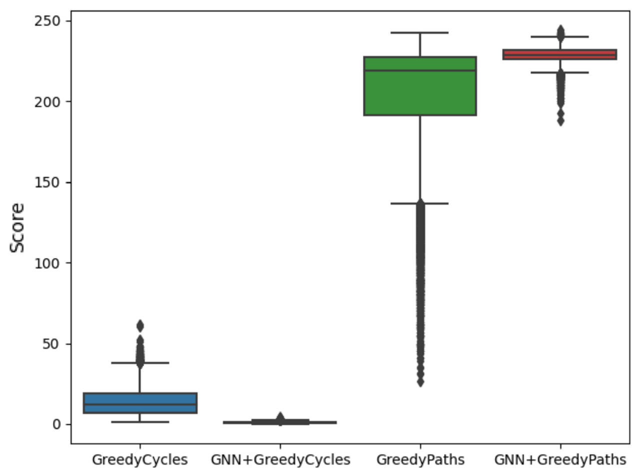

**Fig. 10** Box plot comparing the approximate solution scores obtained when each of the evaluated methods was used in the test dataset. The evaluated methods were two non-learnt heuristics, *GreedyCycles* and *GreedyPaths*, and their 2 stage method versions, *GNN+GreedyCycles* and *GNN+GreedyPaths*

To the best of our knowledge, this is the first time this operation is proposed in the literature. It is potentially useful for any edge classification task on graphs, specially when the problem involves constraints in which only one edge may be chosen per node, be it destination or origin node. These constraints are very common in optimization problems in graphs; this is the case for KEP, for example: Each donor or patient–donor pair may donate at most one kidney, and each patient or patient–donor pair may receive at most one kidney.

### 4.3.5 Unconstrained GNN model

This method, referred to from now on as *Unsupervised GNN*, consists of a GNN model which receives a KEP instance and outputs, for each edge of the input instance, a score and a binary prediction, which indicates if the edge is part of the predicted solution or not. The binary prediction is made independently for each edge, and consists of a simple decision threshold. This GNN model is trained using the unsupervised loss described at Sect. 4.3.2 with the loss regularization term described at Sect. 4.3.3. Although there is no guarantee that the solutions given by this method will be valid, the loss regularization term is used with the goal of inducing it to respect the problem constraints.

### 4.3.6 Two stage method

Inspired by the approach used in [17, 18], this method follows a two steps structure: firstly, the learnable step, which is a GNN, takes the KEP graph instance as an input and outputs a score for each edge; then, the non-learnable step, which is one of the heuristic methods described above in the Sect. 4.2 is executed, but using the scores given by the GNN instead of the edge weights. This process is illustrated in the diagram on Fig. 1. The intuition behind this idea is that the GNN model will learn to encode in the edge score contextual information that will change the decisions of the search heuristic so as to maximize the total score of the final output solution.

This GNN model is trained without supervision using loss described at Sect. 4.3.2. There is no need to use the

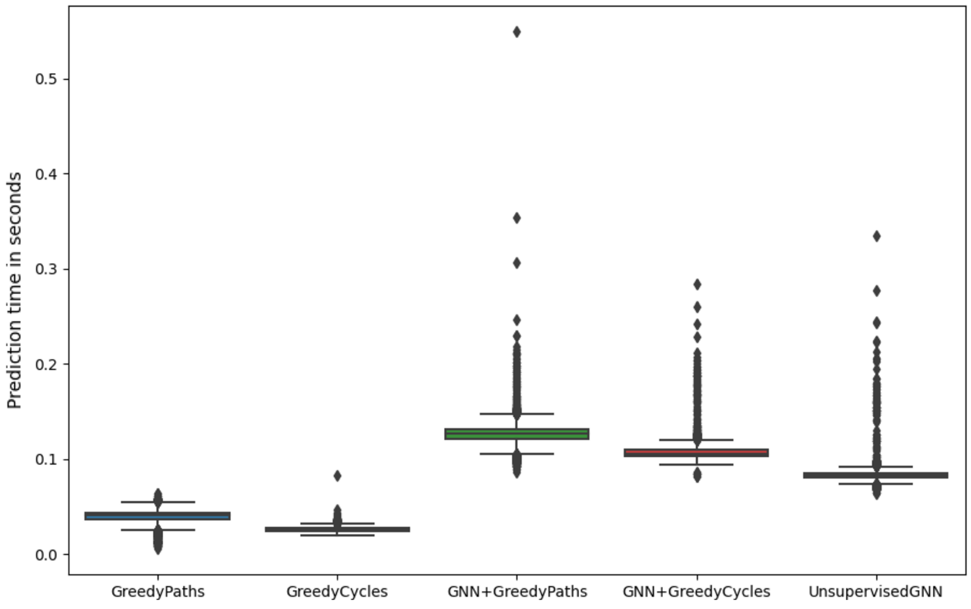

**Fig. 11** Box plot comparing the time to compute a solution on each of the 10 thousand KEP instances of the test dataset, each one with 300 nodes. The evaluated methods were two non-learnt heuristics, *GreedyCycles* and *GreedyPaths*, their 2 stage method versions, *GNN+GreedyCycles* and *GNN+GreedyPaths*, and *UnsupervisedGNN*, which is a GNN trained and used without an heuristic

loss regularization term described at Sect. 4.3.3 because the second step of the method ensures that the output will be a valid solution.

Two versions of the two-stage method were implemented for the experiments. Both use the GNN described in Sect. 4.3.1 for the first stage, but their second stage consist of different search heuristics. One uses the GreedyPaths search heuristic described at Sect. 4.2.1 and is referred to later on as *GNN+GreedyPaths*. The other uses the GreedyCycles search heuristic described at Sect. 4.2.2 and is referred to later on as *GNN+GreedyCycles*.

## 5 Experiments

The main goal of the experiments was to assess the machine learning methods, and compare them to the deterministic heuristics and to the exact solution, both in terms of the quality of the solutions as well as of their operational performance, i.e., the time it takes for a solution to be calculated. The integer programming method, however, could not be measured in the same dataset because it took too long to run the solver.

The objective of the Kidney Exchange Problem is to maximize the number of donations weighted by their compatibility index, i.e., their associated edge weights in the graph. Therefore, this was the main metric used to quantify the quality of each solution and to compare the different methods, and is also referred to as **score** in this study. As the operational performance of the methods was also to be compared, the total time that each method took to run per instance was also measured.

Ideally, the heuristics could be compared by using the optimality gap, which is defined as the distance between the heuristic solution score and the optimal solution score. However, as the randomly generated KEP instances of 300 nodes have shown to be intractable while using the PyCSP3 solver, i.e., impossible to solve optimally in a reasonable time, the optimality gap could not be measured.

For the trainable heuristics, the total training time was also measured and compared. To monitor and guide the training process, the evolution of the loss function value over the training time was also collected. Additionally, the mean score of the current model in the validation dataset was measured at each validation phase.

For a fair comparison between methods, all the measurements were made in the same test dataset, which is described in Sect. 3, with the exception of the exact solution method, as explained in Sect. 5 below.

### 5.1 Solver execution time analysis

To measure how much time it takes to solve a KEP instance in relation to the input size, the following experiment was designed: first, we randomly generate 100 instances of each graph size (i.e., number of nodes), starting from 5 nodes and up to 300 nodes; then, each one is solved optimally using the integer programming method described at Sect. 4.1. The elapsed time of each solver execution is collected for later analysis.

### 5.2 Training of the ML models

The training of the ML models was separated in epochs. In each epoch, we iterate through the 10 thousand instances of the train dataset, predicting, calculating the loss, and updating the GNN weights according to it. At every 500 instances, a validation phase is run, where the model is evaluated on the validation dataset and a checkpoint is saved. The chosen batch size was 1, which means that the predictions were done on one instance at a time, because it empirically seemed to be the best for the learning process. There were two machine learning models that were trained in this study: the unconstrained unsupervised GNN and the two-stage method, which are described at Sect. 4.3. For each training process, it was measured how the loss value evolved over time in the training and in the validation datasets.

### 5.3 Evaluation of KEP solving methods

In order to do a fair comparison between the different heuristic methods, all of them were evaluated by predicting the solutions of all 10 thousand instances of the test dataset. We did not set the cycles and paths size limit (parameter $k$ in constraint represented by Eq. 7), i.e., the cycles and paths could have any length, as long as they respect the KEP constraints. We intend to assess the performance of these methods with limited cycles and paths length soon. For each prediction, it was measured the solution score, the number of edges in the solution, and the relation between the solution score (which is the sum of the edge weights of the solution edges) and the total sum of edge weights of the graph. In addition, the validity of the predicted solution was evaluated; if invalid, we measure the number of invalid edges in the solution, i.e., the number of edges that disrespect the restrictions. Furthermore, we measured the time it took for each method to solve each instance using a CPU. Then, it was measured, for each model in relation to the whole test dataset, the mean, standard deviation, and distribution for the scores and the prediction elapsed times.

## 6 Results

### 6.1 Solver time measurements

Figure 2 shows a box plot of the time that the solver took to optimally solve KEP instances in relation to the instance size. For that, graphs of sizes 5–15 (i.e., number of nodes) were used; initially, graphs with up to 300 nodes were going to be included in the analysis, but as solving graphs with 16 nodes or more would take several days to compute, they were excluded. To solve a hundred instances with 15 nodes, for instance, it took 101.2 h in total.

As we can see, the experiment results show a pattern of exponential growth of computational time in relation to the input size. The mean time it took when the graph node number was beneath 10 was always below 1.5 s. For instances with 15 nodes, the mean time measured was 3569.33 s, i.e., roughly one hour.

### 6.2 Training of the ML models

Figures 3, 4, and 5 show the evolution of the loss value on the training and validation datasets for the two-stage methods described at Sect. 4.3.6, *GNN+GreedyPaths* and *GNN+GreedyCycles*, and for the *Unsupervised GNN* method described at Sect. 4.3.5, respectively.

Figures 6, 7, and 8 show the evolution of the mean score value for *GNN+GreedyPaths*, *GNN+GreedyCycles*, and *Unsupervised GNN*, respectively. Figure 9 shows the evolution of the standard deviation of the scores predicted on the validation dataset.

### 6.3 Methods' performances

Figure 10 shows a box plot of the approximate solution scores (i.e., the sum of the weights of the edges contained in the approximate solution) achieved by each method in the test dataset, with the exception of *Unsupervised GNN* and *integer programming*. As all the solutions found by the *Unsupervised GNN* method were invalid, there was no reason to evaluate their quality. The integer programming method is also absent from the plot, as it was not possible to run it in instances with 300 nodes.

As we can see, the GreedyPaths heuristic method found much better solutions than GreedyCycles. The 2 stage method variations, *GNN+GreedyCycles* and *GNN+GreedyPaths*, obtained very different performances. Unfortunately, the GNN module in the *GNN+Greedy-Cycles* method did not manage to learn to output edge scores that help the heuristic; on the contrary, the quality of its approximate solutions are considerably worse than those from *GreedyCycles*. The *GNN+GreedyPaths* method, however, effectively learned to output better solutions than the non-learnt heuristic achieving a mean score of 228.40 on the test dataset; while, *GreedyPaths* obtained 203.79; this shows an improvement of 12% of the mean solution score with the use of the GNN. We can also see that, although the best scores achieved by each of them are very similar, the score distribution is very different: While *GreedyPaths* outputs approximate solutions with very large range of scores, *GNN+GreedyPaths*'s approximate solutions have much more consistent scores, obtaining a decent performance throughout all instances of the dataset.

### 6.4 Methods' computational time

The box plot in Fig. 11 shows the comparison between the time it took for each method to solve the KEP instances of the test dataset. As we can see, all of them took less than a second. Although the difference is not large, the two basic search heuristics took less time than the GNN based methods, and *GNN+GreedyPaths* took, on average, the most time.

## 7 Analysis of experimental results

### 7.1 Solver time analysis

Because the Kidney Exchange problem is NP-Hard, the time it takes to optimally solve each instance is expected to grow exponentially as the instance size grows. Regardless, considering that in [2] real life KEP instances could be optimally solved in a reasonable time, it was expected that we would also be able to optimally solve the ones used in this study, since they have been constructed to have similar sizes to the ones used in the article. However, as described in Sect. 6.1, instances of size as small as 15 already took in average 1 h to solve. Considering that we would want to optimally solve all 10 thousand instances of the test set in order to fairly compare to the other methods and to measure their optimality gap, this process would take an unreasonable time, estimated to be around 10 thousand hours, i.e., roughly 1.14 years. This estimate is only if the KEP instances on the test dataset had 15 nodes; for instances with 300 nodes, it would surely take an unreasonably enormous amount of time.

The number of nodes of the input instance is not, however, the sole factor that determines the time it takes to optimally solve it; in a set of instances with the same number of nodes, some are "harder" than others, i.e., take more time to solve. As the instance size increases, so does the variability of the time to solve it: The minimum and maximum times measured for instances with 15 nodes were 5.17 s and 29,779.01 s (i.e., roughly 8 h); for comparison, the minimum and maximum times for instances with 5 nodes were 0.8 and 1.3 s. Hence, it is possible that, while some real life instances are solvable in a reasonable time, a percentage of them would take too long to solve, thus becoming intractable.

There can be several reasons why the authors of [2] could compute the optimal solution of their KEP instances in much less time. First of all, they probably used a solver tool that is much more efficient than PyCSP3. In addition, the computer used may be much more powerful than the one used in this study. Also, it is important to remember that the NP-Hardness of KEP guarantees that the computational time it takes to solve the worst case scenario grows exponentially, but in practice real life instances may often have specific properties which may cause them to be either harder or easier to solve. Hence, another plausible reason is that their instances may be much easier to solve. Furthermore, the authors used a constrain relaxation technique that speeds up significantly the solving process. These set of reasons alone may not explain totally why they were able to optimally solve their KEP instances much faster than we did on our data; this may be further investigated in future work.

## 7.2 Training of the GNN model

As we can see in Fig. 3, the training of the *GNN+GreedyPaths* method was successful, seen as it managed to optimize the GNN by minimizing the loss function. The training of the *GNN+GreedyCycles* and *Unsupervised GNN* methods, however, were unsuccessful, as shown by the loss curves of Figs. 4 and 5, which do not decrease over time.

Although at first glance at Fig. 8 the *Unsupervised GNN* model seems to achieve great scores, unfortunately all its output solutions were invalid, i.e., they did not comply to the KEP constraints. It is clear that the loss constraint regularization (described at Sect. 4.3.3) added to the loss function was unsuccessful in helping the model learn to comply to the KEP constraints. This highlights the necessity of having a methods that guarantee with total certainty that all its output solutions are valid. However, even though there were many manual trials with different hyperparameter combinations, it is still possible that a variation of this technique could work with a different setting, i.e., another GNN architecture, other hyperparameters, and so on.

Because of the skip connection that sums the original edge weights to the predicted edge scores at the end of the GNN, the predicted solutions start off very similar to the ones made by *GreedyPaths*. Then, the changing of the GNN weights disrupts these scores, which increases the loss, but goes on to improve them, eventually arriving at a performance that is better than *GreedyPaths*. After some point (around epoch 6 in Fig. 6), the learning converges to a solution, and after a while the performance starts to slowly worsen. The final model was chosen from the checkpoint with the highest score measured on the validation dataset, which was in epoch 6, step 3500. As we can see on Fig. 9, at this point the model's scores on the validation also presented the lowest standard deviation, which indicates that the model's predictions were more consistent, maintaining a decent performance throughout all instances.

## 7.3 Methods' performances

Ideally, we would want to evaluate and compare each method by measuring their optimality gap for each instance, i.e., how far the approximate solution is from the optimal one. However, as we do not have access to the optimal solution, this was not possible. We can nevertheless estimate it roughly by examining an upper bound: Each instance has 300 nodes, and each node may donate and receive at most one kidney; thus, the solution with the most edges would be contain cycles that together comprehend all nodes. As each edge weight is a value between 0 and 1, the maximum score possible is equal to 300, when all edges in the cycle have a weight of 1. Hence, an upper bound for the score is the number of nodes, which in this case is 300. This is obviously extremely unrealistic, as it assumes that all nodes are PDPs, that there are a set of cycles that links all of them, and that the solution edge weights are equal to 1 (as the edge weights are values sampled from a uniform distribution between 0 and 1, their average value is 0.5). Considering the score upper bound of 300 as a very conservative estimate for the mean optimal solution value, we can estimate that the absolute and relative optimality gap for the *GreedyPaths* method would be 96.28 and 32%; for *GreedyCycles*, these values would be 286.29 and 95.4%; for *GNN+GreedyPaths*, 71.59 and 23.8%; for *GNN+GreedyCycles*, 299.15 and 99.7%. Hence, the improvement on the optimality gap of the *GNN+GreedyPaths* method in relation to *GreedyPaths* would be at least 34.4% (96.28–71.59), which is already a very substantial improvement.

It is clear that the two methods that searched for paths performed much better than the ones that searched for cycles. There are many possible explanations for this observed behavior. Maybe the best cycles-only solution in a KEP instance is usually much worse than the best paths-only solution. This, however, can only be verified by making comparisons to the cycles only and paths only exact solutions, which are unavailable. Also, the *GreedyCycles* method is probably less efficient because it discards the path it is constructing if it does not end up closing a cycle. It also shortens the constructed cycle if it closes before the node it had begun on, which may also lead to worse performance overall. The *GreedyPaths* method does not have these issues, as it always keeps the edges it adds to each path it constructs.

We can also observe that the while the GNN module in the *GNN+GreedyPaths* improved the performance in relation to the basic non-learnable heuristic, in *GNN+GreedyCycles* it only worsened it. It is possible that its GNN module in *GNN+GreedyCycles* could not learn the needed context to know if a given edge would lead to a longer and higher-valued cycle because it is too complex, and does not depend that much on the 3-neighborhood context, which is the limit of information gather in each node with the GNN architecture used, as it only has 3 message passing GNN layers. Another possible explanation is that it is way harder for a model to learn to compute edge scores that help the choices of *GreedyCycles* because it is inherently more complex than *GreedyPaths*, i.e., it is not just a sequence of simple decisions, as it also has to keep track of the rest of the nodes of the cycle being constructed, check if it closed a cycle, and remove from the solution in construction the edges added before the node where the cycle was closed. Put simply, the more complex the second step heuristic is, the harder it is for a machine learning model to learn to help it.

As explained at Sect. 6.3, *GNN+GreedyPaths* approximate solutions have much more consistent scores, which suggests that it probably handle much better "hard" instances. A plausible interpretation is that the GNN module helps the subsequent greedy heuristic to avoid choosing edges that are only locally good, but lead to worse paths overall. It is able to do this because it considers information of the neighborhood context.

### 7.4 Methods' computational time

The GNN computational complexity is linear, as it performs a fixed amount of computations per graph node plus another fixed amount of computations per edge. As for the heuristic methods *GreedyPaths* and *GreedyCycles*, their computational complexity is also linear, as the worst case scenario one edge for each node will be added, one by one, into the solution; hence, it always performs a quantity of computational operations linearly proportional to the number of nodes of the input instance, at worst.

The results from Sect. 6.4 show that every method tested in this work took very little time to execute, with the exception of the *integer programming* method, which took so much time that applying it to 300 nodes instances became intractable. The *GreedyPaths* method took a bit more time than *GreedyCycles* probably because it found better solutions overall, and consequently took more computing steps to construct each solution. The same effect may also explain the prediction time difference between *GNN+GreedyPaths* and *GNN+GreedyCycles*. The *UnsupervisedGNN* method took a bit less time to execute than the two step methods, which was expected because it runs the same computations, but without the second step, which is the basic search heuristic.

## 8 Conclusion and future work

In this work, several heuristic methods with and without machine learning for approximately solving the Kidney Exchange Problem were proposed and investigated. They were tested on an artificial dataset and compared between each other and with an implementation of an exact solution method. Additionally, it was made an experiment for measuring the time it took for the exact solution method to solve an instance in relation to the instance size. The results of the evaluations and experiments were then analyzed and discussed.

### 8.1 Answers to the research questions

As seen from the results in Sect. 7.3, the main question presented at Sect. 1 was answered: Yes, the Kidney Exchange problem can be better approximately solved with the help of machine learning.

As for the feasibility of the ML methods, the *GNN+GreedyPaths* method surpassed all other evaluated heuristics in terms of the quality of the solutions it provides; among all evaluated methods, it remains the one that best approximately solves the dataset instances in a reasonable time. The other ML methods evaluated in the work (*UnsupervisedGNN* and *GNN+GreedyCycles*), however, did not achieve good results.

Regarding the viability of such methods in terms of computational time, the GNN module adds an almost insignificant overhead when compared to the non-learnable heuristics. When compared to the solver running the exact solution, it is several orders of magnitude faster for instances with at least 15 nodes. The complexity of the two-stage methods is linear, turning instances that were previously intractable due to their size into easily approximately solvable in a reasonable time.

As for the limitations of the employed machine learning methods for this problem, it is clear that they are not simple to use, as they need to be properly trained, which is not easy to do. Although using the two-stage method potentially improves considerably the performance in relation to the basic heuristics, as happened with the *GNN+GreedyPaths* method, it also introduces several new hyperparameters which need to be adequately set in order for the method to work. Applying supervised learning turned out to be unfeasible because of the need for the exact solution to be used as edge labels. Regarding the *UnsupervisedGNN* method, our results suggest that imposing the constraints through adding terms in a loss function is actually really hard; hence, the method never learns to output valid solutions, rendering it useless.

### 8.2 Main contributions

In the following list, a summary of each of the main contributions that this work provided is presented.

**Learnable heuristics for KEP** Although the two-stage approach was already used by past work [17, 18, 24], this was the first work to adapt it and apply it to KEP. Two variations of the approach were implemented: *GNN+GreedyPaths* and *GNN+GreedyCycles*, the first one having achieved satisfactory performance.

**Non-learnable heuristics for KEP** Two new deterministic heuristic methods for approximately solving the Kidney Exchange Problem were introduced: *GreedyPaths* and *GreedyCycles*. They were also evaluated in the test dataset, giving insight on their effectiveness.

**Node-wise softmax** A variation of the softmax activation function designed to be applied to edge scores in graph problems was created and implemented. It showed to be useful for the GNN, empirically improving its performance. The author plans to contribute to the PyTorch library with the implementation of this technique, thus making it available and easily usable by its future users.

**KEP unsupervised loss** A novel loss function (described at Sect. 4.3.2) designed for KEP was introduced. It optimizes the weighted sum of the edges in the predicted solution without the need for supervision. It was validated in the training of the *GNN+GreedyPaths* method, as seen in Fig. 3 in Sect. 6.2, where it lead the GNN to learn to effectively help the heuristic method construct better approximate solutions.

### 8.3 Future directions

**Using real data** Use data collected by countries' or hospital's healthcare system to evaluate how the presented methods would perform in real life situations. This would also allow us to compare our proposed methods with others that were already evaluated in the same data.

**Using better artificial data** There more sophisticated methods for generating artificial KEP instances, such as the ones presented at [3, 32]. They are still far from sufficiently similar to real data so as to substitute evaluating it. However, it would still probably give an evaluation of KEP solving methods that is closer to that of real life situations. Furthermore, training the model with these instances could also potentially lead to better results. Another possibility of generating better artificial data would be to use a graph generator model that learns to create instances similar to the real data; the Graph Variational Auto-Encoder presented at [33] and the MolGAN presented at [34] are good examples of candidate methods to be adapted for that goal.

**Training the models with supervised learning** Another direction is to develop a method that learns with supervision, evaluate it, and compare it to the other methods. For that, we would first have to obtain the optimal solutions, which would then be used as labels for the supervised training. This could be done either by improving a lot the integer programming method's speed and/or by training on much smaller instances (i.e., instances with less than 15 nodes). Another promising variation of this idea is to use the $N$ best solutions, and create soft labels where each edge would have a value between 0 and 1 that would indicate how often it appears in the best solutions, weighted by the quality of these solutions.

**Running a GNN before every step of the greedy heuristic** Instead of running the GNN once and then passing the edge scores to the greedy heuristic method, new node embeddings could be generated before each step of the heuristic. Although significantly costlier in terms of inference, as the GNN is executed many times per instance, this has shown good results for other graph route optimization problems [22]. This approach is called *autoregressive decoding* and explored for solving the TSP by [18].

**Funding** CAPES Finance code (001) and CNPq supported this research in part.

**Data availability** The code implemented and the data used in this paper are freely available at https://github.com/pfpimenta/kep_gnn.

## Declarations

**Conflict of interest** The authors declare no conflict of interest in this research.

## References

1. Roth A, Sönmez T, Unver U (2004) Kidney exchange. Q J Econ 119(2):457–488
2. Anderson R, Ashlagi I, Gamarnik D, Roth AE (2015) Finding long chains in kidney exchange using the traveling salesman problem. Proc Natl Acad Sci 112(3):663–668. https://doi.org/10.1073/pnas.1421853112
3. Delorme M, García S, Gondzio J, Kalcsics J, Manlove D, Pettersson W, Trimble J (2022) Improved instance generation for

kidney exchange programmes. Comput Oper Res 141:105707. https://doi.org/10.1016/j.cor.2022.105707

4. Axelrod DA, Schnitzler MA, Xiao H, Irish W, Tuttle-Newhall E, Chang S-H, Kasiske BL, Alhamad T, Lentine KL (2018) An economic assessment of contemporary kidney transplant practice. Am J Transplant 18(5):1168–1176. https://doi.org/10.1111/ajt.14702
5. Longest Kidney Transplant Chain—Guinness World Record Organization Distinguishes the National Kidney Registry for World's Longest Kidney Transplant Chain. Accessed: 2023-03-01 (2020). https://transplantsurgery.ucsf.edu/news–events/ucsf-news/88223/UCSF-Part-of-Longest-Kidney-Transplant-Chain—Guinness-World-Record-Organization-Distinguishes-the-National-Kidney-Registry-for-World%E2%80%99s-Longest-Kidney-Transplant-Chain
6. Biró P, van de Klundert J, Manlove D, Pettersson W, Andersson T, Burnapp L, Chromy P, Delgado P, Dworczak P, Haase B, Hemke A, Johnson R, Klimentova X, Kuypers D, Nanni Costa A, Smeulders B, Spieksma F, Valentín MO, Viana A (2021) Modelling and optimisation in European kidney exchange programmes. Eur J Oper Res 291(2):447–456. https://doi.org/10.1016/j.ejor.2019.09.006
7. Abraham DJ, Blum A, Sandholm T (2007) Clearing algorithms for barter exchange markets: enabling nationwide kidney exchanges. In: Proceedings of the 8th ACM conference on electronic commerce. EC'07, pp 295–304. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1250910.1250954
8. Yang Y, Rajgopal J (2020) Learning combined set covering and traveling salesman problem. arXiv preprint arXiv:2007.03203
9. Lemos H, Prates MOR, Avelar PHC, Lamb LC (2019) Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In: 31st IEEE international conference on tools with artificial intelligence ICTAI, pp 879–885. https://doi.org/10.1109/ICTAI.2019.00125
10. Santos HLd (2020) Solving the decision version of the graph coloring problem: a neural-symbolic approach using graph neural networks. Master's thesis, UFRGS Federal University, Porto Alegre, Brazil. https://search.ebscohost.com/login.aspx?direct=true &AuthType=shib &db=cat07377a &AN=sabi.001114939 &lang=pt-br &scope=site &authtype=guest,shib &custid=s5837110 &groupid=main &profile=eds
11. Sato R, Yamada M, Kashima H (2019) Approximation ratios of graph neural networks for combinatorial problems. In: Wallach HM, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox EB, Garnett R (eds) Advances in neural information processing systems, vol 32, pp 4083–4092
12. Abe K, Xu Z, Sato I, Sugiyama M (2019) Solving np-hard problems on graphs with extended AlphaGo zero. arXiv https://doi.org/10.48550/ARXIV.1905.11623
13. Khalil EB, Dai H, Zhang Y, Dilkina B, Song L (2017) Learning combinatorial optimization algorithms over graphs. In: Guyon I, Luxburg U, Bengio S, Wallach HM, Fergus R, Vishwanathan SVN, Garnett R (eds) Advances in neural information processing systems, vol 30, pp 6348–6358. https://proceedings.neurips.cc/paper/2017/hash/d9896106ca98d3d05b8cbdf4fd8b13a1-Abstract.html
14. Nazi A, Hang W, Goldie A, Ravi S, Mirhoseini A (2019) Gap: generalizable approximate graph partitioning framework. arXiv https://doi.org/10.48550/ARXIV.1903.00614
15. Li Z, Chen Q, Koltun V (2018) Combinatorial optimization with graph convolutional networks and guided tree search. In: Bengio S, Wallach HM, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R (eds) Advances in neural information processing systems, vol 31, pp 537–546. https://proceedings.neurips.cc/paper/2018/hash/8d3bba7425e7c98c50f52ca1b52d3735-Abstract.html
16. Bai Y, Xu D, Sun Y, Wang W (2021) GLSearch: maximum common subgraph detection via learning to search. In: Meila M, Zhang T (eds) Proceedings of the 38th international conference on machine learning, ICML, pp 588–598. http://proceedings.mlr.press/v139/bai21e.html
17. Joshi CK, Laurent T, Bresson X (2019) An efficient graph convolutional network technique for the travelling salesman problem. arXiv https://doi.org/10.48550/ARXIV.1906.01227
18. Joshi CK, Cappart Q, Rousseau L-M, Laurent T (2021) Learning tsp requires rethinking generalization. In: 27th international conference on principles and practice of constraint programming (CP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik
19. Prates MOR, Avelar PHC, Lemos H, Lamb LC, Vardi MY (2019) Learning to solve np-complete problems: A graph neural network for decision TSP. In: The thirty-third AAAI conference on artificial intelligence, AAAI 2019, pp 4731–4738. https://doi.org/10.1609/aaai.v33i01.33014731
20. Vinyals O, Fortunato M, Jaitly N (2015) Pointer networks. Adv Neural Inf Process Syst 28:3505
21. Wu Y, Song W, Cao Z, Zhang J, Lim A (2021) Learning improvement heuristics for solving routing problems. IEEE Trans Neural Netw Learn Syst 33(9):5057–5069
22. Kool W, Hoof H, Welling M (2019) Attention, learn to solve routing problems! In: 7th international conference on learning representations, ICLR 2019, New Orleans, LA, USA, May 6–9, 2019. OpenReview.net. https://openreview.net/forum?id=ByxBFsRqYm
23. Bello I, Pham H, Le QV, Norouzi M, Bengio S (2017) Neural combinatorial optimization with reinforcement learning. In: 5th international conference on learning representations, ICLR 2017, Toulon, France, April 24–26, 2017, Workshop track proceedings. OpenReview.net. https://openreview.net/forum?id=Bk9mxlSFx
24. Peng Y, Choi B, Xu J (2021) Graph learning for combinatorial optimization: a survey of state-of-the-art. Data Sci Eng 6(2):119–141. https://doi.org/10.1007/s41019-021-00155-3
25. Lamb LC, Garcez AS, Gori M, Prates MOR, Avelar PHC, Vardi MY (2020) Graph neural networks meet neural-symbolic computing: a survey and perspective. In: Bessiere C (ed) Proceedings of the twenty-ninth international joint conference on artificial intelligence, IJCAI 2020, pp 4877–4884. https://doi.org/10.24963/ijcai.2020/679
26. Shervashidze N, Schweitzer P, Leeuwen EJ, Mehlhorn K, Borgwardt KM (2011) Weisfeiler–Lehman graph kernels. J Mach Learn Res 12(77):2539–2561
27. Lecoutre C, Szczepanski N (2020) PYCSP3: modeling combinatorial constrained problems in python. CoRR arXiv:2009.00326
28. Roth AE, Sönmez T, Ünver MU (2007) Efficient kidney exchange: coincidence of wants in markets with compatibility-based preferences. Am Econ Rev 97(3):828–851. https://doi.org/10.1257/aer.97.3.828
29. Constantino M, Klimentova X, Viana A, Rais A (2013) New insights on integer-programming models for the kidney exchange problem. Eur J Oper Res 231(1):57–68. https://doi.org/10.1016/j.ejor.2013.05.025
30. Corso G, Cavalleri L, Beaini D, Liò P, Velič ković P (2020) Principal neighbourhood aggregation for graph nets. arXiv. https://doi.org/10.48550/ARXIV.2004.05718. arXiv:2004.05718
31. Brody S, Alon U, Yahav E (2021) How attentive are graph attention networks? arXiv. https://doi.org/10.48550/ARXIV.2105.14491. arXiv:2105.14491
32. Saidman SL, Roth AE, Sönmez T, Ünver MU, Delmonico FL (2006) Increasing the opportunity of live kidney donation by matching for two- and three-way exchanges. Transplantation 81(5):773–782

33. Simonovsky M, Komodakis N (2018) Graphvae: towards generation of small graphs using variational autoencoders. In: Artificial neural networks and machine learning—ICANN 2018. Lecture Notes in Computer Science, vol 11139, pp 412–422. https://doi.org/10.1007/978-3-030-01418-6_41
34. De Cao N, Kipf T (2018) Molgan: An implicit generative model for small molecular graphs. arXiv preprint https://doi.org/10.48550/ARXIV.1805.11973arXiv:1805.11973