Soft Comput
DOI 10.1007/s00500-017-2907-x

METHODOLOGIES AND APPLICATION

A parallel hybrid optimization algorithm for some network design problems

Ibrahima Diarrassouba[1] · **Mohamed Khalil Labidi**[2,3] · **A. Ridha Mahjoub**[3]

Abstract Network design problems have been widely studied in the last decades due to the importance of ICT in our daily life and are still the subject of extensive researches. Network design covers a large family of problems, and several algorithms, both exact and heuristic methods, have been proposed to address each of them. In this paper, we consider two variants of the so-called survivable network design problem and propose a generic parallel hybrid algorithm to solve them. The algorithm is based on the hybridization of a Lagrangian relaxation algorithm, a greedy algorithm and a genetic algorithm. We present, for each variant, a computational study showing the efficiency of our approach in producing both lower and upper bounds for the optimal solution.

Keywords Genetic algorithm · Hybridization · kHNDP · kESNDP · Lagrangian relaxation · Metaheuristic · Parallel computing

Communicated by V. Loia.

✉ A. Ridha Mahjoub
mahjoub@lamsade.dauphine.fr

Ibrahima Diarrassouba
ibrahima.diarrassouba@univ-lehavre.fr

Mohamed Khalil Labidi
mohamed-khalil.labidi@dauphine.fr

[1] LMAH, FR-CNRS-3335, Le Havre University, Le Havre, France

[2] University of Tunis El-Manar, Faculty of Science of Tunis, URAPOP, UR13ZS38, Tunis, Tunisia

[3] Paris Dauphine University, PSL Research University, CNRS UMR 7243, LAMSADE, 75016 Paris, France

1 Introduction

Designing efficient networks is nowadays of crucial importance since networks take a large place in many fields (telecommunications, logistics, economics, ICT, etc.). Addressing network design issues has raised a large class of problems.

Survivable network design problems (SNDP) are those network design problems which aim at designing networks that are still functioning even when failures occur. The importance of survivability in networks has led to a wide literature on these problems, and several algorithms, both exact and heuristic methods, have been proposed to address them.

In this paper, we consider two SNDP, namely the k-edge-connected survivable network design and the k-edge-connected hop-constrained survivable network design problems. The two problems are defined as follows. Given a weighted undirected graph $G = (V, E)$ where each edge e has a weight ω_e, a set of demands $D \subseteq V \times V$, a positive integer k, the *k-edge-connected survivable network design problem* (kESNDP) is to find a minimum weight subgraph of G such that for each demand $\{s, t\} \in D$, there exist k edge-disjoint paths between s and t. If, in addition, we require that, for all demand $\{s, t\} \in D$, there exist k edge-disjoint st-paths of length at most L, and for some integer $L \geq 2$, the problem obtained is the *k-edge-connected hop-constrained survivable network design problem* (kHNDP).

Both kESNDP and kHNDP are NP-hard and have been investigated in the literature. The kESNDP is a particular case of a more general problem, called *general survivable network design problem* (GSNDP) in which it is required that there exist r_{st} edge-disjoint paths between s and t, for all $\{s, t\} \in D$. This latter, and its variants, has been studied by several authors [see for instance Steiglitz et al. (1969), Magnanti and Raghavan (2005), Winter (1987), Grötschel

Published online: 13 November 2017

and Monma (1990), Grötschel et al. (1992) and Goemans and Bertsimas (1993)]. In Goemans and Bertsimas (1993), the authors studied a variant of the problem in which each node u of the graph is given an integer $r_u \geq 0$ which corresponds to the minimum number of paths connecting u to the rest of the network, and proposed a heuristic to solve this problem. Grötschel and Monma (1990) and Grötschel et al. (1992) considered the polytope associated with the GSNDP and gave some valid inequalities as well as conditions for these inequalities to define facets. Magnanti and Raghavan (2005) presented several integer programming formulations for the GSNDP for both undirected and directed graphs. They also discussed the efficiency of each formulation in terms of LP-relaxation. Kerivin and Mahjoub (2005) presented a review of the main models associated with the GSNDP as well as the main associated polyhedral results. Another well-known SNDP is the so-called *k-edge-connected subgraph problem* (kECSP) in which it is required that there exist k edge-disjoint paths between each pair of nodes of the graph. This problem corresponds to the kESNDP where $D = \{\{s, t\},$ for all $s, t \in V$ with $s \neq t\}$, and $r_{st} = k$, for all $\{s, t\} \in D$. Several papers deal with the structural properties of the solutions of the kECSP. Among these, we mention the papers of Kerivin et al. (2004) and Bendali et al. (2010) who proposed a polyhedral approach and Branch-and-Cut algorithms for the kECSP, respectively, when $k = 2$ and $k \geq 3$.

The kHNDP has also been widely studied. Dahl and Gouveia (2004) considered the directed version of the problem. They described some valid inequalities and give a complete description of the polytope of the problem when $k = 1$, $|D| = 1$ and $L \leq 3$. In Huygens et al. (2007), and the authors proposed a polyhedral approach for the kHNDP when $k = 2$, $|D| \geq 2$ and $L = 2, 3$. They introduced several valid inequalities and devised a Branch-and-Cut algorithm for the problem. Diarrassouba et al. (2016a) investigated the kHNDP when $L = 2, 3$. They presented several integer programming formulations based on graph transformations. They also compared these formulations both in terms of LP-relaxation and in terms of efficiency. In Botton et al. (2013), the authors present the first formulation of the k-HNDP for any $k, L \geq 1$ and use a Benders decomposition method to handle the big number of variables and constraints. They present as well a computational study of various cutting plane and Branch-and-Cut algorithms.

Diarrassouba et al. (2016b) also investigated a version of the kHNDP in which it is required that the paths are node disjoint. They presented some valid inequalities and devised a Branch-and-Cut algorithm for this problem when $k = 2$.

In this paper, we introduce two heuristics for the kESNDP and the kHNDP based on the hybridization of a Lagrangian relaxation algorithm, a greedy algorithm and a genetic algorithm, used in a parallel computing framework. We test the algorithms within an extensive computational study and compare their efficiency for solving both kESNDP and kHNDP against CPLEX.

The paper is organized as follows. In Sect. 2, we give the main motivations of this work. Then, in Sect. 3, we present the parallel hybrid algorithm, its main components and the computational results, for the kESNDP. In Sect. 4, we present the algorithm for the kHNDP, along with some computational results. In Sect. 5, we show and discuss the impact of the parallelization in our approach. Finally, in Sect. 6, we discuss the generalization of our framework and give some concluding remarks.

2 Motivations

Despite the continuous and significant development of computer's calculation performance, it remains difficult to solve to optimality, within a reasonable amount of time, many combinatorial optimization problems (COPs) for large-scale input data. For most of these hard problems, including the kESNDP and the kHNDP, one of the main issues when solving to optimality with Branch-and-Cut or Branch-and-Price algorithms is that solving even the linear relaxation of the considered integer programming formulation can be time-consuming for large-scale instances. This may prevent the Branch-and-Cut or Branch-and-Price algorithm from a good exploration of the enumeration tree and finding good quality solutions [see for instance Diarrassouba et al. (2016b)].

In this work, we address these two issues (excessive CPU time and good exploration of the solutions space) by using both parallel computing and hybridization. We first devise a Lagrangian relaxation algorithm for both the kESNDP and kHNDP and use parallel computing in order to reduce the CPU time needed to obtain good lower bound for the problem. Then, we devise a population-based metaheuristic, namely a genetic algorithm, in order to ensure a good exploration of the solutions space. We also devise a greedy heuristic for having simply and quickly heuristic solutions of the problem. Moreover, we hybridize the three algorithms in order to improve the quality of both the lower and upper bounds. As we will see below, the algorithm takes advantage of the structure of the integer programming formulations we consider for the two problems.

3 A parallel hybrid algorithm for the kESNDP

We present in this section the parallel hybrid algorithm for the kESNDP. We first present a flow-based integer programming formulation for the problem and then present the Lagrangian relaxation, the genetic and the greedy algorithms for the kESNDP. Finally, we present their hybridization and parallelization scheme. We recall that the kESNDP is defined by an

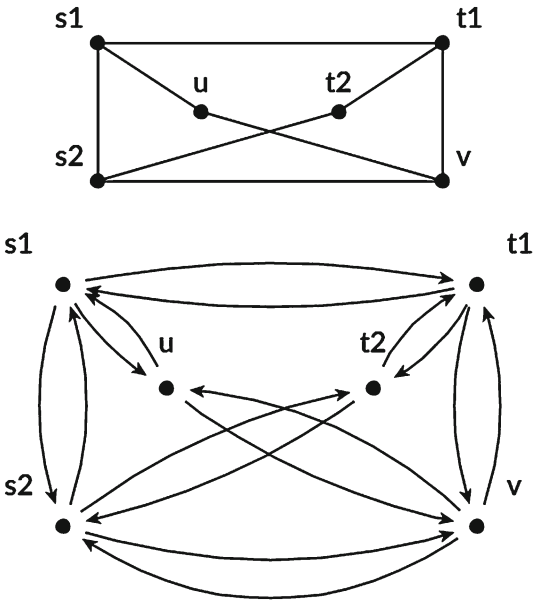

Fig. 1 Graph transformation for the kESNDP

undirected graph $G = (V, E)$, a set of demands $D \subseteq V \times V$ given by origin–destination pairs $\{s, t\} \in D$ with $s \neq t$, and a positive integer k.

3.1 Integer programming formulation

The kESNDP can be formulated as a flow-based integer program [see Magnanti and Raghavan (2005)]. For each edge $uv \in E$, let x_{uv} be the $0-1$ variable which takes value 1 if the edge uv is in a solution of the problem and 0 otherwise. Also let $\widetilde{G} = (V, A)$ be the directed graph obtained from G by replacing each edge $uv \in E$ by two arcs of the form (u, v) and (v, u) (see Fig. 1 for an illustration).

Finally, let f_{uv}^{st} be the flow variable associated with every arc (u, v) of $\widetilde{G}$ and a pair $\{s, t\} \in D$. The kESNDP is equivalent to

$$\min \sum_{uv \in E} \omega_{uv} x_{uv}$$

s.t.

$$\sum_{v \in V \setminus \{u\}} f_{uv}^{st} - \sum_{l \in V \setminus \{u\}} f_{lu}^{st} = \left\{ \begin{array}{r} k, \text{ if } \ u = s, \\ -k, \text{ if } \ u = t, \\ 0, \text{ if } \ u \in V \setminus \{s, t\}, \end{array} \right\} \quad \text{for all} \quad u \in V \text{ and } \{s, t\} \in D, \tag{1}$$

$$\left. \begin{array}{l} f_{uv}^{st} \\ f_{vu}^{st} \end{array} \right\} \leq x_{uv}, \quad \text{for all} \quad uv \in E \text{ and } \{s, t\} \in D, \tag{2}$$

$$f_{uv}^{st}, f_{vu}^{st} \geq 0, \quad \text{for all} \quad uv \in E \text{ and } \{s, t\} \in D, \tag{3}$$

$$x_{uv} \leq 1, \quad \text{for all} \quad uv \in E, \tag{4}$$

$$x_{uv} \in \{0, 1\}, \quad \text{for all} \quad uv \in E, \tag{5}$$

$$f_{uv}^{st}, f_{vu}^{st} \in \{0, 1\}, \quad \text{for all} \quad uv \in E, \ \{s, t\} \in D. \tag{6}$$

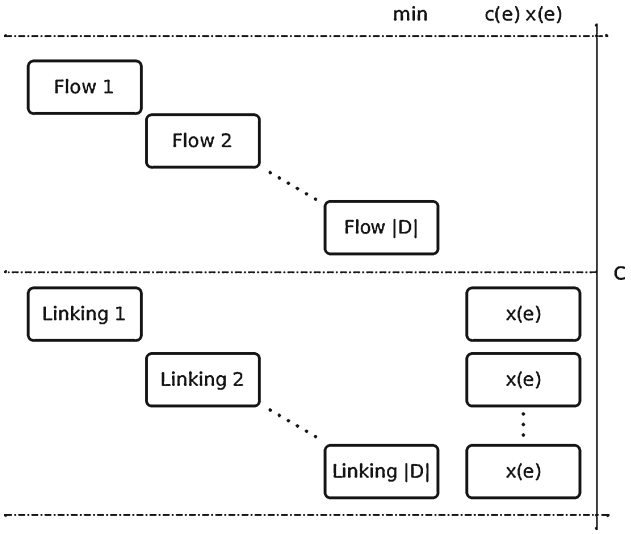

Fig. 2 Block structure of the kESNDP undirected flow formulation

The formulation is called "undirected flow formulation" (UFP). Inequalities (1) are the *flow conservation constraints*. They express the fact that between every pair of terminals (s, t) in the auxiliary graph, there exists an integer flow of value k. This flow yields k edge-disjoint paths between s and t in the original graph. Inequalities (2) are the *linking constraints*. They ensure that if an edge uv is not taken in the solution, then no st-flow can use this edge. Inequalities (3) and (4) are the trivial inequalities.

One can easily see that the above formulation has a block structure as illustrated in Fig. 2.

Each block "Flow i" corresponds to the flow conservation constraints associated with a demand $\{s, t\} \in D$.

3.2 The Lagrangian relaxation algorithm

When devising a Lagrangian relaxation algorithm, the main issues that must be taken into account are

- which constraints are relaxed,
- how the Lagrange multipliers are updated,
- what is the stopping criterion.

In our framework, the Lagrangian relaxation is obtained by relaxing the linking constraints (2). We denote by λ_{uv}^{st}, the Lagrange multipliers associated with the linking constraints (2), for all $\{s, t\} \in D$ and $(u, v) \in A$. The Lagrangian relaxation thus obtained, denoted by (LR), is given by

$$\min \sum_{uv \in E} \left[\omega_{uv} - \sum_{\{s,t\} \in D} \left(\lambda_{uv}^{st} + \lambda_{vu}^{st} \right) \right] x_{uv} + \sum_{\{s,t\} \in S} \sum_{uv \in E} \left(\lambda_{uv}^{st} f_{uv}^{st} + \lambda_{vu}^{st} f_{vu}^{st} \right)$$

s.t.

$$\sum_{v\in V} f_{uv}^{st} - \sum_{l\in V} f_{ul}^{st} = \left\{ \begin{array}{l} k, \text{ if } u = s, \\ -k, \text{ if } u = t, \\ 0, \text{ if } u \in V\setminus\{s,t\}, \end{array} \right\}, \quad \text{for all} \quad u \in V \text{and } \{s,t\} \in D,$$

$$0 \le f_{uv}^{st}, f_{vu}^{st} \le 1, \text{ for all} \quad uv \in E, \{s,t\} \in D,$$

$$0 \le x_{uv} \le 1, \quad \text{for all} \quad (u,v) \in E,$$

$$x_{uv} \in \mathbb{Z}, \quad \text{for all} \quad (u,v) \in E,$$

$$f_{uv}^{st} \in \mathbb{Z}, \quad \text{for all} \quad (u,v) \in A, \{s,t\} \in D.$$

Observe that, apart the objective function, the variable x appears in problem (LR) only in the trivial constraints $0 \le x_{uv} \le 1$. Thus, solving problem (LR) reduces to solving the problem (LR') as follows:

$$\min \sum_{\{s,t\}\in S} \sum_{uv\in E} \left(\lambda_{uv}^{st} f_{uv}^{st} + \lambda_{vu}^{st} f_{vu}^{st}\right)$$

s.t.

$$\sum_{v\in V} f_{uv}^{st} - \sum_{l\in V} f_{ul}^{st} = \left\{ \begin{array}{l} k, \text{ if } u = s, \\ -k, \text{ if } u = t, \\ 0, \text{ if } u \in V\setminus\{s,t\}, \end{array} \right\}, \quad \text{for all } u \in V \text{and } \{s,t\} \in D,$$

$$0 \le f_{uv}^{st}, f_{vu}^{st} \le 1, \text{ for all} \quad uv \in E, \{s,t\} \in D,$$

$$f_{uv}^{st} \in \mathbb{Z}, \quad \text{for all} \quad (u,v) \in A, \{s,t\} \in D.$$

One can easily see that if $(\overline{f}^{st}, \{s,t\} \in D)$ is an optimal solution for problem (LR'), then solution $(\overline{x}, \overline{f}^{st}, \{s,t\} \in D)$ where

$$\overline{x}_{uv} = \left\{ \begin{array}{ll} 1 & \text{if} \quad \omega_{uv} - \sum\limits_{\{s,t\}\in D} \left(\lambda_{uv}^{st} + \lambda_{vu}^{st}\right) < 0, \\ 0 & \text{otherwise,} \end{array} \right., \text{ for all } uv \in E,$$

is optimal for problem (LR).

Also, it is not hard to see that problem (LR') consists in $|D|$ independent minimum cost st-flow problems in graph $\widetilde{G}$. Thus, problem (LR') can be solved by using any combinatorial algorithm solving the minimum cost flow problem. The reader can refer to Ahuja et al. (1993) for more details on minimum cost flow problems and the associated algorithms. Moreover, since the minimum cost flow problems of (LR') are independent, we solve (LR') in a parallel multi-threaded fashion. Thus, we can solve (LR) in time O(MF) when using $|D|$ processors, where O(MF) is the runtime for solving a minimum cost st-flow problem.

An issue to address in the Lagrangian relaxation algorithm (Held and Karp 1971; Beasley 1993) is how the Lagrange multipliers, λ_{uv}^{st}, $(u,v) \in A$ and $\{s,t\} \in D$, are updated. For this, we use the so-called subgradient method, which is known for its easy implementation and its speed. Let $\lambda_k = (\lambda_{uv,k}^{st})_{uv\in E,\{s,t\}\in D}$ be the vector of the current Lagrange multipliers at iteration k of the Lagrangian relaxation algorithm, $(\overline{x}, \overline{f}^{st}, \{s,t\} \in D)$ an optimal solution of problem (LR), z_k the optimal value of (LR), and Z_{UB} an upper bound of the optimal solution of the original problem (UFP). Then, the Lagrange multipliers at iteration $k+1$ are given by

$$\lambda_{uv,k+1}^{st} = \max\left\{0, \lambda_{uv,k}^{st} - \rho_k \gamma_{uv,k}^{st}\right\}$$

where

$$\left. \begin{array}{l} \gamma_{uv,k}^{st} = \overline{x}_{uv}^{*} - \overline{f}_{uv}^{st} \\ \gamma_{uv,k}^{st} = \overline{x}_{uv}^{*} - \overline{f}_{vu}^{st} \end{array} \right\}, \quad \text{for all} \quad uv \in E,$$

$$\rho_k = \frac{1}{2^k} \frac{Z_{\text{UB}} - z_k}{\| \gamma_k \|^2}.$$

The upper bound Z_{UB} can be obtained by a heuristic running independently from the Lagrangian relaxation algorithm. In our case, we produce a feasible solution to the kESNDP at each iteration of the Lagrangian relaxation algorithm and choose Z_{UB} as the best value among all solutions thus obtained. A feasible solution, say $\overline{y} \in \mathbb{R}^E$, to kESNDP can be obtained from the optimal solution of problem (LR) by choosing $\overline{y}_{uv} = \max\{\overline{f}_{uv}^{st}, \overline{f}_{vu}^{st}, \text{ for all } \{s,t\} \in D\}$. Clearly, the solution $(\overline{y}, \overline{f}^{st}, \{s,t\} \in D)$ satisfies constraints (1)–(6).

Finally, the Lagrangian relaxation algorithm stops either when a CPU time of 2 h is reached or the algorithm processes 2000 iterations or after 100 iterations without improving the best known upper bound. The algorithm returns the best lower and upper bounds obtained.

3.3 The genetic algorithm

Now we turn our attention to the genetic algorithm. A genetic algorithm consists in considering a set of solutions, feasible or not, in order to produce one or more new feasible solutions. The set of solutions is called the *population*. The algorithm randomly chooses two solutions, called *parents*, in the population and combines them to produce one or more new solutions, called *children*. For a general combinatorial optimization problem, the children solutions may not be feasible for the problem. In this case, the algorithm tries to transform them into feasible solutions and put them into the population. In our case, we will see that both parents and children solutions we build are feasible for kESNDP.

The choice of implementing a genetic algorithm is motivated by the block structure of the flow formulation. Indeed, as said before, this allows to consider a feasible solution of the kESNDP as a sequence of st-flows, each of value k, for all $(s,t) \in D$. As we will see below, considering a solution as a sequence of st-flows allows to easily combine different

solutions in order to produce new feasible ones, which is the key idea of genetic algorithms.

The design of a genetic algorithm takes into account the following issues

- solution encoding and evaluation,
- the parent selection,
- the crossover (how the parent are combined),
- the population management,
- the stopping criterion.

For more details on genetic algorithms, the reader can refer to Talbi (2002). In the remainder of this section, we discuss these issues for our genetic algorithm for the kESNDP.

3.3.1 Encoding and evaluation of a solution

A solution of the kESNDP can be represented in different ways, but in our algorithm, we represent a solution by a set of $0-1$ vectors $(\overline{f}^{s_1t_1}, \ldots, \overline{f}^{s_dt_d})$, where $\overline{f}^{s_it_i} \in \{0,1\}^{|A|}$ is a flow vector associated with demand s_it_i.

The evaluation of a solution encoded by $(\overline{f}^{s_1t_1}, \ldots, \overline{f}^{s_dt_d})$ consists in giving the weight of the subgraph of G corresponding to that solution. The weight of such a solution is given by $Z = \sum_{uv\in E} \omega_{uv}\overline{x}_{uv}$, where

$$\overline{x}_{uv} = \max\{\overline{f}^{st}_{uv}, \overline{f}^{st}_{vu}, \text{ for all } \{s,t\} \in D\}.$$

3.3.2 Parent selection and crossover

The generation of children solutions is done in the following way. For two solutions, say $P_1 = (\overline{f}^{s_1t_1}, \ldots, \overline{f}^{s_dt_d})$ and $P_2 = (\overline{g}^{s_1t_1}, \ldots, \overline{g}^{s_dt_d})$, we randomly choose two integers a and b with $2 \le a < b \le d$. Then, we cross P_1 and P_2 according to integers a and b and produce two solutions C_1 and C_2 such that

$$C_1 = \left(\overline{f}^{s_1t_1}, \ldots, \overline{f}^{s_{a-1}t_{a-1}}, \overline{g}^{s_at_a}, \ldots, \overline{g}^{s_{b-1}t_{b-1}}, \overline{f}^{s_bt_b}, \ldots, \overline{f}^{s_dt_d}\right)$$

and

$$C_2 = \left(\overline{g}^{s_1t_1}, \ldots, \overline{g}^{s_{a-1}t_{a-1}}, \overline{f}^{s_at_a}, \ldots, \overline{f}^{s_{b-1}t_{b-1}}, \overline{g}^{s_bt_b}, \ldots, \overline{g}^{s_dt_d}\right).$$

Clearly, C_1 and C_2 are feasible for the kESNDP if P_1 and P_2 are feasible for the kESNDP.

The children generation phase is done by applying the above procedure to $0.1N_{\text{pool}}$ randomly chosen pairs of solutions (P_1, P_2), where N_{pool} is the number of elements into the pool before the generation of the new solutions.

3.3.3 Population management

The pool of solutions is initialized with all feasible solutions produced by the greedy algorithm described in Sect. 3.4. During the algorithm, we ensure that the pool of solutions contains no more than 100 solutions. The solutions are sorted in increasing order w.r.t. their evaluation. For simplicity, we denote by N^i_{pool} the number of solutions in the pool after the generation of the new solutions at iteration i. In the following, we first discuss the case where the genetic algorithm runs simultaneously with the Lagrangian relaxation and greedy algorithms and then when it is not the case.

We discuss first the case where the genetic algorithm is running simultaneously with the Lagrangian relaxation and greedy algorithms. In this case, for every iteration $i \ge 1$, we remove from the pool the $N^i_{\text{pool}} - 100$ worst solutions, if $N^i_{\text{pool}} \ge 100$. Otherwise, we do not remove any solution.

Now we consider the case where the genetic algorithm is not running simultaneously with the Lagrangian relaxation and greedy algorithms. This is the case when both the Lagrangian relaxation and greedy algorithms are terminated or when the genetic algorithm is running separately from the hybridization. In this case, for every iteration i such that $1 \le i \le 200$, if $N^i_{\text{pool}} \ge 100$, then we remove the $N^i_{\text{pool}} - 100$ worst solutions. When $i > 200$, if $N^i_{\text{pool}} \ge 20$, we remove from the pool the $\left(N^i_{\text{pool}} - N^{i-1}_{\text{pool}}\right) + 5\%N^{i-1}_{\text{pool}}$ worst solutions. If $N^i_{\text{pool}} < 20$, then we remove the $\left(N^i_{\text{pool}} - N^{i-1}_{\text{pool}}\right)+1$ worst solutions.

Note that this management strategy of the pool guarantees that the number of solutions in the pool slowly decreases until it remains one solution in the pool, and this using the hybridization or not.

3.3.4 Stopping criterion

The genetic algorithm stops either when the CPU time reaches 2 h or the pool of solutions contains only one solution. Note that in both cases, the first solution of the pool is the best solution obtained by the algorithm.

3.4 The greedy algorithm

We choose to implement a greedy algorithm for the kESNDP because of its simplicity of implementation. Also, as we will see below, our greedy algorithm relies on a set of orderings of the demand set. Each of these orderings is considered independently from each other, which yields a possible parallel implementation of our greedy algorithm.

The main idea of our greedy heuristic is to produce an upper bound of the optimal solution of the kESNDP. For this, the algorithm produces a set of feasible solutions, each being

obtained by iteratively computing minimum cost st-flows, for all $\{s,t\} \in D$, and keep the best one. Moreover, each st-flow takes into account the minimum st-flows obtained prior.

More precisely, the algorithm starts by randomly generating a set of $|D|$ orderings of the demands. Each ordering will produce a feasible solution. For a given ordering of the demands, let $D = \{\{s_1,t_1\},\{s_2,t_2\},\ldots,\{s_d,t_d\}\}$ be the demands w.r.t that ordering. Next, we compute a minimum cost s_1t_1-flow of value k in $\widetilde{G}$, where all the arcs have capacity 1 and arcs (u,v) and (v,u) have a cost ω_{uv}. Let E_1 be the set of edges uv of G such that either (u,v) or (v,u) have a flow value of 1 in that minimum cost flow. Then, we compute a minimum cost s_2t_2-flow of value k in $\widetilde{G}$ with all the arcs having capacity 1, the arcs (u,v) and (v,u) such that $uv \in E_1$ have a cost 0 and all the arcs (u,v) and (v,u) such that $uv \in E \setminus E_1$ have a cost ω_{uv}. More generally, if E_i denotes the set of edges of E having a flow value 1 in the minimum cost s_it_i-flow, we compute a minimum cost $s_{i+1}t_{i+1}$-flow of value k with all the arcs having capacity 1, all the arcs corresponding to an edge of $\bigcup_{j=1}^{i} E_j$ having a cost 0, and all the arcs (u,v) corresponding to an edge uv of $E \setminus \bigcup_{j=1}^{i} E_j$ having a cost ω_{uv}.

It is not hard to see that the edge set $\bigcup_{j=1}^{|D|} E_j$ is a feasible solution of the kESNDP. This procedure is repeated for all the ordering of the demands, yielding $|D|$ feasible solutions for the kESNDP. The algorithm ends by finding the best one among these solutions.

Each minimum cost flow is computed with the network simplex algorithm, which can be implemented to run in polynomial time. Thus, the overall algorithm runs in $\mathcal{O}(|D||V|^2|E|\log(|V|C))$ with $C = \max(\omega_{uv})$.

3.5 The hybridization and parallelization scheme

Now we present the hybridization and parallelization scheme of the overall algorithm. For simplicity, we denote by Lagrangian relaxation algorithm by LRA, the genetic algorithm by GA and the greedy algorithm by SH. The parallel hybrid algorithm will be denoted by PHA.

We remark that in PHA, the three algorithms LRA, GA and SH run in parallel. We also hybridize the three algorithms in the following way. First, the solutions generated by algorithms SH and LRA are introduced in the pool of solutions of GA. Also, at the beginning of PHA, the global upper bound Z_{UB} is set to ∞. Then, each time LRA, SH and GA generate a feasible solution, and the value Z of this solution is compared to Z_{UB}. If $Z < Z_{\mathrm{UB}}$, then the best upper bound Z_{UB} is updated with the value of Z. Note that the value Z_{UB} is used by LRA to update the Lagrange multipliers.

Algorithm 1: Algorithm PHA for the kESNDP.

Data: An undirected graph $G = (V, E)$, the demand set D, a positive integer $k \geq 1$
Result: A lower and upper bounds of the optimal solution of the kESNDP

```
1  begin
2      Z_UB ← ∞;
3      Execute SH, LRA et GA in parallel;
4      SH:
5      Computes |D| arbitrary orderings of the demands;
6      for each ordering on the demands do
7          Computes a feasible solution, according to that ordering,
           using the Network Simplex Algorithm to solve a series of
           minimum cost flow subproblems);
8          Add the new solution obtained into the population pool.
           Let Z_SH be its value;
9          if Z_SH < Z_UB then
10             Z_UB ← Z_SH;
11     Informs GA that has ended;
12     LRA:
13     for each iteration do
14         Compute the Lagrange multipliers using Z_UB and the
           current solution of problem (LR). Let z be the optimal
           value of (LR);
15         Compute a feasible solution from the solution of problem
           (LR). Let Z_LRA be its value;
16         Add this solution into the pool;
17         if Z_LRA < Z_UB then
18             Z_UB ← Z_LRA;
19         if z > Z_LB then
20             Z_LB ← z;
21     Informs GA that has ended;
22     GA:
23     for each iteration do
24         Sort the solutions of the pool by increasing order w.r.t. to
           their weight;
25         Randomly choose several pairs of solutions from the pool
           and combine the solutions of each pair in order to
           generate new solutions;
26         for every new solution do
27             Let Z_GA be the value of the solution;
28             if Z_GA < Z_UB then
29                 Z_UB ← Z_GA;
30             Add the solution to the pool of solutions;
31         Delete the worst solutions from the pool (according to the
           pool management strategy);
32     return (Z_LB, Z_UB);
```

We also ensure that GA continues running after LRA and SH are terminated. This is done in order to give to GA enough time to process the solutions of LRA and SH.

Finally, notice that together with the global upper bound Z_{UB}, the Lagrangian relaxation algorithm also produces a lower bound of the optimal solution of the kESNDP.

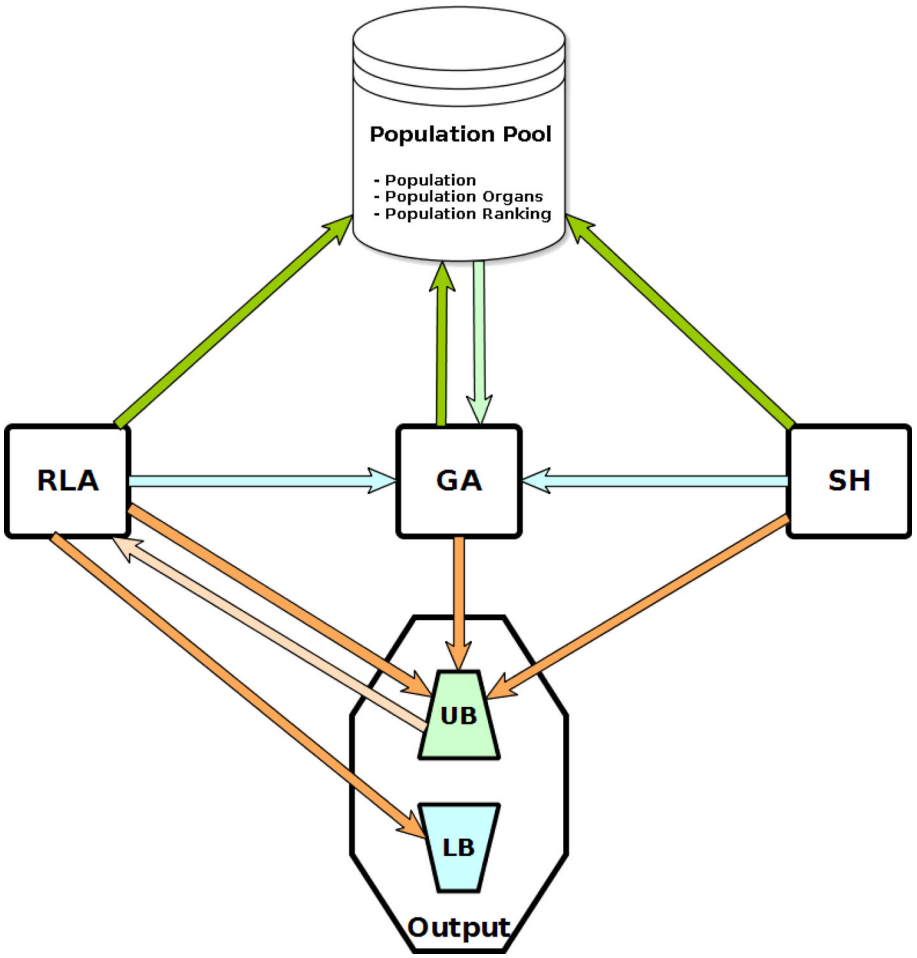

Fig. 3 PHA communication scheme

The main operations in PHA are summarized in Algorithm 1, while Fig. 3 describes the communication scheme of PHA.

3.6 Computational results

In this section, we present the computational experiments we have conducted for the kESNDP. The aim is to show the efficiency of our algorithm in producing solutions of good quality for the problem, and this, in a relatively short computation time. To do this, we first compare the results obtained, for several instances, by algorithm PHA against those obtained by solving the flow formulation with CPLEX. Then, we compare each algorithm SH, LRA and GA against the overall PHA algorithm for solving the kESNDP.

All the algorithms have been implemented in C++, and we have used CPLEX (12.5) for solving the undirected flow formulation of the kESNDP. The experiments have been conducted on a computer equipped with an Intel i7 processor at 2.5 Ghz with 8 Gb of RAM, running under Linux. We have set the maximum CPU time of all the algorithms, including CPLEX, to 2 h. The test problems have been composed of graphs from the TSPLIB (1995) library that are complete Euclidean graphs. For the demands, we have considered single-source multi-destination demand set (called rooted demands) and multi-source multi-destination demand set (arbitrary demands). The number of nodes of the graphs is from 30 to 318, while the number of demands varies from 10 to 159 for both rooted and arbitrary demands. We have also solved each instance with connectivity requirement $k = 3$. Each instance is described by its name followed by the number of nodes of the graph and the number of demands. When the demands are rooted, the number of demands is preceded by "r" while arbitrary demands are indicated by "a" before the number of demands. For example, berlin30-r10 denotes an instance composed of a graph from TSPLIB with 30 nodes and 10 rooted demands, and st70-a35 denotes an instance composed of a graph from TSPLIB with 70 nodes and 35 arbitrary demands.

Tables 1 and 2 report the results for the kESNDP with $k = 3$ by both PHA and CPLEX for rooted demands and arbitrary demands, respectively. The entries of each table are

$\|V\|$:	Number of nodes of the graph,
$\|D\|$:	Number of demands,
UB:	Best upper bound achieved by PHA (resp. CPLEX),
LB:	Best lower bound achieved by PHA (resp. CPLEX),
Gap:	Relative error between the best upper and lower bounds achieved by PHA (resp. CPLEX),
CPU:	Total CPU time in hours:min:sec achieved by PHA (resp. CPLEX).

Remark that for some instances, CPLEX has not been able to solve even the linear relaxation after the maximum CPU time (2 h). The results for these instances are indicated with "–."

From Table 1, we can observe that for the rooted instances, 2 instances over 22 have been solved to optimality by CPLEX, while PHA produces an upper bound of the optimal solution for all the instances. Also, for 9 instances, CPLEX produces a better feasible solution than that obtained by PHA. However, for all the other instances, the upper bound produced by PHA is better than that obtained by CPLEX after the maximum CPU time. For example, for instances st70-r35 and kroA100-r50, PHA produces an upper bound of 1265 and 47454, respectively, while CPLEX produces an upper bound of 1278 and 49323. We have also compared the lower bound achieved by PHA with that obtained by CPLEX. For several instances (12 over 22), the lower bound obtained by CPLEX after the maximum CPU time is better than that obtained by PHA. This can be explained by the fact that CPLEX, during the resolution process, adds several valid inequalities, such as Gomory cuts and general upper bound inequalities, which allows us strengthening the linear relaxation of the problem and improve the quality of the lower bound.

We can also observe that for small graphs (up to 70 nodes and 35 demands), CPLEX produces a better gap than PHA. However, PHA performs better for large size instances. Also remark that for large instances, the gap of CPLEX is high (near 90%). This shows the limit of CPLEX for solving large instances and the utility of our approach to handle such instances.

For the arbitrary instances (see Table 2), the observations are almost the same. PHA produces better upper bounds for 11 instances over 20. PHA is even able to produce both lower and upper bounds for 5 instances where CPLEX has not been able to solve the linear relaxation of the problem. Also, for 9 instances, PHA produces a better lower bound than that obtained by CPLEX. For the other instances, the lower bound achieved by CPLEX is better than that obtained PHA.

Finally, we compare CPLEX and PHA in terms of CPU time. We clearly see, from Tables 1 and 2, that except 2 instances with rooted demands instances and with arbitrary

Table 1 Results for PHA and CPLEX for the kESNDP and rooted demands with $k = 3$

Instances			PHA				CPLEX			
Name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	LB	Gap	CPU
berlin	30	10	7168	6072.29	15.29	00:00:11	6241	6241	0	00:43:55
berlin	30	15	10,970	7856.13	28.39	00:00:17	7982	7828	1.93	02:00:00
berlin	30	20	12,173	8391.95	31.06	00:00:27	8510	8510	0	00:56:08
berlin	30	25	12,805	8832.3	31.02	00:00:36	9040	9014.5	0.28	02:00:00
berlin	52	10	6815	4740.6	30.44	00:00:45	5399	4959.58	8.14	02:00:00
berlin	52	15	9871	6821.52	30.89	00:01:12	7505	6680.3	10.99	02:00:00
berlin	52	26	13,033	7639.51	41.38	00:01:58	9270	8223.08	11.29	02:00:00
st	70	15	761	523.647	31.19	00:01:21	607	438.95	27.69	02:00:00
st	70	26	1077	707.247	34.33	00:02:24	928	612.011	34.05	02:00:00
st	70	35	1265	805.399	36.33	00:03:22	1278	707.5	44.64	02:00:00
kroA	100	20	26, 677	11,864.8	55.52	00:05:52	22,637	12,354.3	45.42	02:00:00
kroA	100	35	39,378	16,182.3	58.91	00:10:12	32,738	19,189	41.39	02:00:00
kroA	100	50	47,454	18,875.4	60.22	00:15:03	49,323	23,501.4	52.35	02:00:00
kroA	150	30	36,340	11,596.3	68.09	00:19:09	43,987	16,106.5	63.38	02:00:00
kroA	150	50	47,764	15,774.2	66.97	00:33:04	64,280	22,374.3	65.19	02:00:00
kroA	150	75	57,909	17,742.1	69.36	00:50:48	418,670	28,031.4	93.3	02:00:00
kroA	200	40	41,724	12,080.4	71.05	00:45:44	65,762	17,020	74.12	02:00:00
kroA	200	75	59,482	16,096.8	72.94	01:28:01	–	–	–	02:00:00
kroA	200	100	69,462	18,792.2	72.95	02:00:00	–	–	–	02:00:00
lin	318	61	26,723	6068.56	77.29	02:00:00	–	–	–	02:00:00
lin	318	111	55,000	8702.16	84.18	02:00:00	–	–	–	02:00:00
lin	318	159	71,464	8719.74	87.8	02:00:00	–	–	–	02:00:00

Table 2 Results for PHA and CPLEX for the kESNDP and arbitrary demands with $k = 3$

Instances			PHA				CPLEX			
Name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	LB	Gap	CPU
berlin	30	10	10,422	8216.0	21.17	00:00:15	9276	9276.0	0	00:00:08
berlin	30	15	12,385	9716.2	21.55	00:00:24	10,749	10,655.2	0.87	02:00:00
berlin	52	10	10,553	7772.5	26.35	00:00:42	8508	8256.9	2.95	02:00:00
berlin	52	15	13,588	9141.0	32.73	00:01:23	10,270	9870.4	3.89	02:00:00
berlin	52	20	15,603	10,347.5	33.68	00:01:37	11,499	11,499.0	0	00:30:04
st	70	15	1176	678.3	42.32	00:01:31	773	631.3	18.33	02:00:00
st	70	26	1531	979.5	36.02	00:02:56	1067	866.5	18.79	02:00:00
st	70	35	1888	1202.6	36.31	00:04:20	1534	1082.5	29.43	02:00:00
kroA	100	20	42,845	19,839.5	53.69	00:06:58	49,089	19,602.5	60.07	02:00:00
kroA	100	35	61,802	26,609.3	56.94	00:12:54	86,377	28,331.3	67.2	02:00:00
kroA	100	50	68,537	31,378.5	54.22	00:18:27	155,967	33,839.0	78.3	02:00:00
kroA	150	30	56,725	22,773.6	59.85	00:23:24	49,126	24,173.6	50.79	02:00:00
kroA	150	50	71,392	29,684.8	58.42	00:40:14	154,870	32,461.8	79.04	02:00:00
kroA	150	75	88,672	36,283.2	59.08	01:02:03	553,614	42,973.0	92.24	02:00:00
kroA	200	40	64,500	23,745.5	63.19	00:54:46	353,805	26,323.2	92.56	02:00:00
kroA	200	75	91,687	32,872.0	64.15	01:50:27	–	–	–	02:00:00
kroA	200	100	102,137	37,932.2	62.86	02:00:00	–	–	–	02:00:00
lin	318	61	49,769	17,303.4	65.23	02:00:00	–	–	–	02:00:00
lin	318	111	86,928	21,634.2	75.11	02:00:00	–	–	–	02:00:00
lin	318	159	121,811	24,089.2	80.22	02:00:00	–	–	–	02:00:00

demands, CPLEX reaches the maximum CPU time for all the instances. On the contrary, the CPU time achieved by PHA is relatively small for most of the instances. Indeed, the CPU time is less than 6 min for 50% of the rooted demand instances and less than 7 min for 45% of the arbitrary demand instances. Only 4 instances for both the rooted demands and arbitrary demands have reached the maximum CPU time. This, together with the above observations, shows that PHA is able to obtain better solutions than CPLEX, and this, in quite short CPU time. Also it appears from Tables 1 and 2 that the nature of the instances, rooted or not, does not have a clear impact on the resolution of the problem either by CPLEX or PHA. However, we can see that the gaps produced by PHA for large instances with arbitrary demands are slightly tighter than those with rooted ones.

Now we turn our attention to the efficiency of PHA w.r.t its components, which are LRA, GA and SH. The aim is to see whether each algorithm taken separately is more efficient than the hybridization or not. For this, we compare the results obtained by LRA, GA and SH separately with those obtained by PHA. The results are given in Tables 3 and 4 for rooted and arbitrary demands, respectively.

We can see from Table 3 that the upper bounds produced by PHA for all the instances with rooted demands are better than those obtained by LRA, SH and GA, taken separately. Also, for the arbitrary demands (see Table 4), PHA outperforms algorithms LRA, SH and GA taken separately. This clearly shows that, with both rooted and arbitrary demands, the hybridization of the three components produces better results than each component taken separately. When comparing PHA and LRA in terms of lower bounds, we can see that for almost all the instances, with both rooted and arbitrary demands, the lower bound obtained by LRA is better than that obtained by PHA. Thus, clearly, the hybridization helps in producing better upper bounds for most of the instances, but does not help in improving the lower bounds.

4 A parallel hybrid algorithm for the kHNDP

We present in this section the parallel hybrid algorithm for the kHNDP when $L \geq 3$. The algorithm is similar to that devised for the kESNDP and is based on the hybridization of a Lagrangian relaxation algorithm, a genetic algorithm and a greedy heuristic. However, we will discuss some specific aspects of the algorithm in the context of the kHNDP. First, we present a flow-based integer programming formulation for the problem. Then we present the Lagrangian relaxation algorithm, the genetic algorithm and the greedy heuristic we have devised for the problem. Finally, we present several computational results and discuss the efficiency of the overall algorithm.

Recall that the kHNDP is defined by an undirected graph $G = (V, E)$, a demand set D and two positive integers $k \geq 1$ and $L \geq 3$.

Table 3 Results for PHA versus LRA, SH and GA for the kESNDP and rooted demands with $k = 3$

Instances			PHA		LRA		SH	GA
Name	$\|V\|$	$\|D\|$	UB	LB	UB	LB	UB	UB
berlin	30	10	7168	6072.3	10,543	6298.8	9128	7459
berlin	30	15	10,970	7856.1	16,661	8163.8	12,930	10,970
berlin	30	20	12,173	8392.0	23,165	8356.8	14,271	12,311
berlin	30	25	12,805	8832.3	22,521	8905.6	15,355	13,271
berlin	52	10	6815	4740.6	13,012	5272.4	8668	7034
berlin	52	15	9871	6821.5	22,157	6894.8	12,212	10,373
berlin	52	26	13,033	7639.5	30,418	8000.7	15,257	13,559
st	70	15	761	523.6	2068	539.0	933	789
st	70	26	1077	707.2	3445	809.6	1271	1095
st	70	35	1265	805.4	4304	899.0	1442	1368
kroA	100	20	26,677	11,864.8	71,777	12,605.4	34,141	28,204
kroA	100	35	39,378	16,182.3	114,336	16,922.2	47,090	39,857
kroA	100	50	47,454	18875.4	160,719	20,416.9	53,756	47,531
kroA	150	30	36,340	11,596.3	101,835	13,239.7	44,497	37,824
kroA	150	50	47,764	15,774.2	170,169	19,149.2	53,690	48,303
kroA	150	75	57,909	17,742.1	224,847	23,497.4	66,579	60,797
kroA	200	40	41,724	12,080.4	154,211	14,451.4	51,335	43,110
kroA	200	75	59,482	16,096.8	277,544	19,038.9	69,033	60,606
kroA	200	100	69,462	18,792.2	361,814	23,728.7	80,717	69,929
lin	318	61	26,723	6068.6	139,868	7713.9	31,788	27,143
lin	318	111	55,000	8702.2	375,430	13,495.8	56,199	55,559
lin	318	159	71,464	8719.7	559,326	14,637.2	77,340	75,389

Table 4 Results for PHA versus LRA, SH and GA for the kESNDP and arbitrary demands with $k = 3$

Instances			PHA		LRA		SH	GA
Name	$\|V\|$	$\|D\|$	UB	LB	UB	LB	UB	UB
berlin	30	10	10,422	8216.0	11,884	8384.9	12,808	10,597
berlin	30	15	12,385	9716.2	18,118	9683.7	15,283	12,584
berlin	52	10	10,553	7772.5	14,052	7691.9	12,369	10,725
berlin	52	15	13,588	9141.0	19,042	9210.8	15,496	13,771
berlin	52	20	15,603	10,347.5	22,341	10,364.2	17,908	16,152
st	70	15	1176	678.3	2236	692.1	1267	1208
st	70	26	1531	979.5	3277	1030.3	1743	1567
st	70	35	1888	1202.6	4673	1289.0	2137	1908
kroA	100	20	42,845	19,839.5	110,812	20,089.4	48,435	44,661
kroA	100	35	61,802	26,609.3	183,685	27,237.4	65,024	61,802
kroA	100	50	68,537	31,378.5	269,704	31,777.2	74,250	69,406
kroA	150	30	56,725	22,773.6	164,720	23,388.5	59,947	56,725
kroA	150	50	71,392	29,684.8	269,616	30,290.8	74,614	71,392
kroA	150	75	88,672	36,283.2	418,821	38,724.2	92,894	89,342
kroA	200	40	64,500	23,745.5	223,326	24,728.0	71,033	66,202
kroA	200	75	91,687	32,872.0	439,205	35,395.6	99,364	92,285
kroA	200	100	102,137	37,932.2	566,823	42,194.5	113,464	102,577
lin	318	61	49,769	17,303.4	74,080	21,788.7	53,054	50,168
lin	318	111	86,928	21,634.2	137,343	31,754.5	92,973	89,158
lin	318	159	121,811	24,089.2	206,426	39,037.3	129,651	122,911

4.1 Integer programming formulation for the *k*HNDP

In order to formulate the problem, we use the graph transformation proposed by Diarrassouba et al. (2017), which transforms the original graph G into a set of directed layered graphs. This transformation is presented below. For each demand $\{s,t\} \in D$, first let $V_{st}^l = \{(u,l),$ for all $u \in V \setminus \{s,t\}\}, l = 1,\ldots,L-1$. Then, let $\widetilde{G}_{st} = (\widetilde{V}_{st}, \widetilde{A}_{st})$ be the directed graph where $\widetilde{V}_{st} = \bigcup_{l=1}^{L-1} V_{st}^l \cup \{s,t\}$. The arc set $\widetilde{A}_{st}$ is obtained by adding in $\widetilde{G}_{st}$

- two arcs of the form $((u,l),(v,l+1))$ and $((v,l),(u,l+1))$, for all $l \in \{1,\ldots,L-2\}$ and every edge $uv \in E$ such that $u, v \in V \setminus \{s,t\}$,
- an arc of the form $(s,(u,1))$, for every edge $su \in E$ with $u \in V \setminus \{s,t\}$,
- an arc of the form $((u,L-1),t)$, for every edge $ut \in E$ with $u \in V \setminus \{s,t\}$,
- an arc of the form (s,t), for every edge $st \in E$,
- an arc of the form $((u,l),(u,l+1))$, for every $u \in V \setminus \{s,t\}$ and $l \in \{1,\ldots,L-2\}$.

Figure 4 gives an illustration with $D = \{\{s_1,t_1\},\{s_2,t_2\}\}$ and $L = 4$.

Diarrassouba et al. (2017) showed that every L-st-path of G corresponds to an st-path in $\widetilde{G}_{st}$ and vice versa. Moreover, They showed that there exists k edge-disjoint L-st-paths in G if and only if there exists k arc-disjoint st-paths in $\widetilde{G}_{st}$ such that any pair of arcs used in these paths corresponds to the same edge of G.

This yields to the following integer programming formulation, called *flow formulation*, for the *k*HNDP when $L \geq 3$.

Let x_e, for each edge $e \in E$, be the $0-1$ variable which takes value 1 if edge e is in the solution and 0 otherwise. Let f_a^{st} be an integer flow variable associated with arc a of $\widetilde{G}_{st}$. The *k*HNDP is equivalent to the following integer program

$$\min \quad \sum_{uv \in E} \omega_{uv} x_{uv}$$

s.t.

$$\sum_{a \in \delta^+(u)} f_a^d - \sum_{a \in \delta^-(u)} f_a^d = \left\{ \begin{array}{ll} k & \text{if } u = s \\ -k & \text{if } u = t \\ 0 & \text{if } u \in \widetilde{V}_{st} \setminus \{s,t\} \end{array} \right\}, \quad \text{for all } u \in \widetilde{V}_{st} \text{ and } \{s,t\} \in D, \tag{7}$$

$$\sum_{a \in A_{st}(e)} f_a^{st} \leq x_e, \text{ for all } e \in E \text{ and } \{s,t\} \in D, \tag{8}$$

$$f_a^{st} \geq 0, \text{ for all } a \in \widetilde{A}_{st} \text{ and } \{s,t\} \in D, \tag{9}$$

$$x_e \leq 1, \text{ for all } e \in E, \tag{10}$$

$$x_e \in \{0,1\}, \text{ for all } e \in E, \tag{11}$$

$$f_a^{st} \in \{0,1\}, \text{ for all } a \in \widetilde{A}_{st} \text{ and } \{s,t\} \in D. \tag{12}$$

4.2 The Lagrangian relaxation algorithm

As for the *k*ESNDP, our Lagrangian algorithm for the *k*HNDP is based on the relaxation of the linking constraints (8). This yields the following linear program, called problem (LR), where λ_{uv}^{st}, for all $uv \in E$, are the Lagrange multipliers associated with the linking constraints,

$$\min \sum_{uv \in E} \left(\omega_{uv} - \sum_{\{s,t\} \in D} \lambda_{uv}^{st} \right) x_{uv} + \sum_{\{s,t\} \in D} \sum_{uv \in E} \lambda_{uv}^{st} \sum_{a \in \widetilde{A}_{st}(uv)} f_a^{st}$$

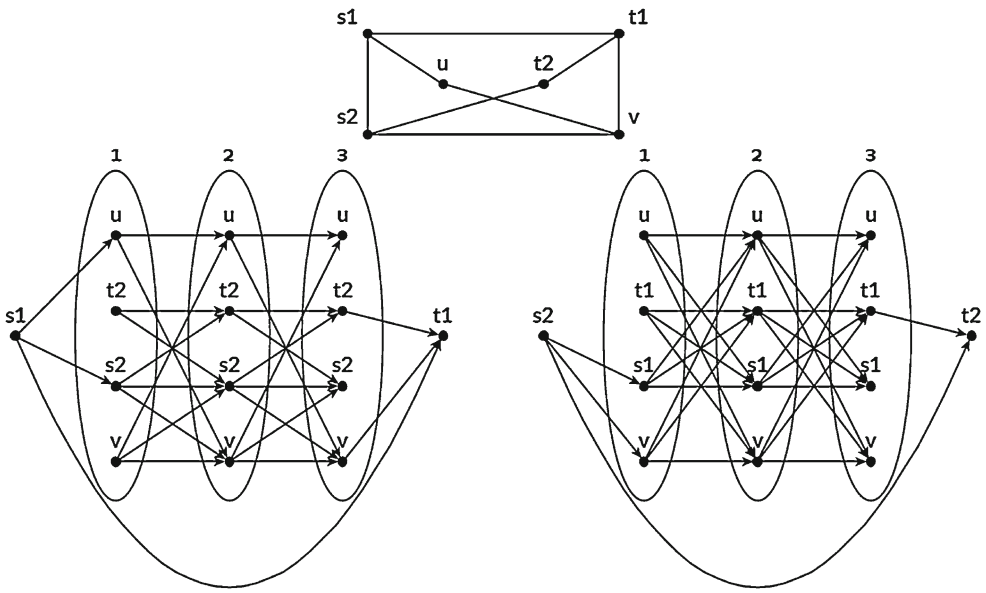

Fig. 4 Graph transformation for $L = 4$

s.t.

$$\sum_{a\in\delta^+(u)} f_a^d - \sum_{a\in\delta^-(u)} f_a^d = \left\{ \begin{array}{rl} k & \text{if } u = s \\ -k & \text{if } u = t \\ 0 & \text{if } u \in \widetilde{V}_{st}\setminus\{s,t\}, \end{array} \right\},$$

for all $u \in \widetilde{V}_{st}$ and $\{s,t\}\in D$,

$0 \le f_a^{st} \le 1$, for all $a \in \widetilde{A}_{st}$ and $\{s,t\} \in D$,

$0 \le x_e \le 1$, for all $e \in E$,

$x_e \in \{0,1\}$, for all $e \in E$,

$f_a^{st} \in \{0,1\}$, for all $a \in \widetilde{A}_{st}$ and $\{s,t\} \in D$.

As before, one can easily see that solving problem (LR) reduces to solving $|D|$ minimum cost st-flow of value k in graphs $\widetilde{G}_{st}$, which can be done in polynomial time. Also, as for the kESNDP, the Lagrange multipliers are updated using the subgradient method.

It should be noticed that, contrarily to the kESNDP, the solution $\overline{y} \in \mathbb{R}^E$ such that

$$\overline{y}_e = \max\{\overline{f}_a^{st}, \text{ for all } a \in \widetilde{A}_{st}(e) \text{ and } \{s,t\} \in D\}, \text{ for all } e \in E,$$

may not be feasible for the kHNDP. Thus, at each iteration of the algorithm, we check whether $\overline{y}$ is feasible for the kHNDP, and if not, we transform $\overline{y}$ into a feasible solution. To see whether $\overline{y}$ is feasible or not, we simply check whether each flow vector $\overline{f}^{st}$, for every $\{s,t\} \in D$, is such that

$$\sum_{a\in\widetilde{A}_{st}(e)} \overline{f}_a^{st} \le \overline{y}_e. \tag{13}$$

If $\overline{y}$ is not feasible for the kHNDP, then for each $\{s,t\} \in D$, let F_{st} be the set of edges of G for which an inequality (13) is violated. We build a new solution $\overline{y}' \in \mathbb{R}^E$ as follows. If for a demand $\{s,t\} \in D$, $F_{st} = \emptyset$, then let $\overline{g}^{st}$ be a flow vector such that $\overline{g}^{st} = \overline{f}^{st}$. If $F_{st} \neq \emptyset$, then for all $e \in F_{st}$, we arbitrarily choose an arc $a_0 \in \widetilde{A}_{st}(e)$, give it a capacity 1, give a capacity 0 to all the arcs of $\widetilde{A}_{st}(e) \setminus a_0$, and compute a minimum cost st-flow of value k, each arc $a \in \widetilde{A}_{st}(uv)$ having a cost ω_{uv}, for all $uv \in E$. Let $\overline{g}^{st}$ be the value of that minimum cost flow. Finally, the solution $\overline{y}'$ is such that

$$\overline{y}'_e = \max\{\overline{g}_a^{st}, \text{ for all } a \in \widetilde{A}_{st}(e) \text{ and } \{s,t\} \in D\}, \text{ all } e \in E,$$

and is clearly feasible for the kHNDP.

4.3 The genetic algorithm

Our genetic algorithm for the kHNDP follows the same lines that we have devised for the kESNDP. The reader can refer to Sect. 3.3 for the details.

4.4 Greedy algorithm

Our greedy heuristic for the kHNDP works similarly to that we have devised for the kESNDP. We start by randomly generating a set of $|D|$ orderings of the demands. For a given ordering, let $D = \{\{s_1,t_1\},\{s_2,t_2\},\ldots\{s_d,t_d\}\}$ be the demands w.r.t that ordering. For $i \in \{1,\ldots,|D|\}$, we denote by E_i the set of edges of G having a flow value of 1 in a minimum cost s_it_i-flow of value k in $\widetilde{G}_{s_it_i}$. Moreover, in this flow at most one arc corresponding to the same edge has a flow value of 1. The edge set E_i is computed as follows. First, compute a minimum cost s_it_i-flow of value k in $\widetilde{G}_{s_it_i}$, where all the arcs have capacity 1, all the arcs corresponding to an edge of $\bigcup_{j=1}^{i-1} E_j$ have a cost of 0, and all the arcs corresponding to the edges $e \in E \setminus \bigcup_{j=1}^{i-1} E_j$ have a cost ω_e. Let $\overline{f}^{s_it_i}$ be the obtained flow vector and let $F_{s_it_i}$ be the set of edges $e \in E$ such that $\sum_{a\in\widetilde{A}_{s_it_i}(e)} \overline{f}_a^{s_it_i} > 1$. If $F_{s_it_i} = \emptyset$, then let E_i be the set of edges $e \in E$ such that $\sum_{a\in\widetilde{A}_{s_it_i}(e)} \overline{f}_a^{s_it_i} = 1$. If $F_{s_it_i} \neq \emptyset$, then, for each edge $e \in F_{s_it_i}$, we arbitrarily choose an arc $a_0 \in \widetilde{A}_{s_it_i}(e)$, give him a capacity 1, give a capacity 0 to all the arcs of $\widetilde{A}_{s_it_i}(e) \setminus \{a_0\}$, and compute a minimum cost flow of value k in $\widetilde{G}_{s_it_i}$ with the same arc costs as before. If $\overline{f}'^{st}$ is the new flow vector, then let E_i be the set of edges $e \in E$ such that $\sum_{a\in\widetilde{A}_{s_it_i}(e)} \overline{f}_a'^{st} = 1$.

Finally, it is not hard to see that the edge set $\bigcup_{j=1}^{|D|} E_j$ is a feasible solution for the kHNDP. This procedure is repeated for all the ordering of the demands, yielding $|D|$ feasible solutions. The algorithm ends by finding the best one among these solutions.

4.5 The hybridization and parallelization scheme

The hybridization and parallelization schemes for the parallel hybrid algorithm for the kHNDP are the same as those devised for the kESNDP. We refer the reader to Sect. 3.5 for the details.

4.6 Computational results

Now we present the computational experiments we have conducted for the kHNDP. As for the kESNDP, the aim is to show the efficiency of algorithm PHA in producing good lower and upper bounds for the kHNDP. For this, we compare the results obtained by PHA with those obtained by solving the undirected flow formulation for the kHNDP against CPLEX. We also compare PHA against LRA, GA and SH taken separately.

All the algorithms have been implemented in C++ and we have used CPLEX (12.5) for solving the undirected flow formulation for the kHNDP. The experiments have been conducted on a computer equipped with an Intel i7 processor at

2.5 Ghz with 8 Gb of RAM, running under Linux. For all the algorithms, we have set the maximum CPU time to 2 h. The test problems are the same as those used for the kESNDP.

Tables 5 and 6 give the results obtained by PHA and CPLEX for the kHNDP with $k = 3$ and with $L = 3$ and $L = 4$, respectively. We only present here results for instances with arbitrary demands. In fact, in our experiments, we have noted that the behavior of our algorithm as well as that of CPLEX are similar for the instances with either arbitrary or rooted demands. The entries of the table are

$\|V\|$:	Number of nodes of the graph,
$\|D\|$:	Number of demands,
UB:	Best upper bound achieved by PHA (resp. CPLEX),
LB:	Best lower bound achieved by PHA (resp. CPLEX),
Gap:	Relative error between the best upper and lower bounds achieved by PHA (resp. CPLEX),
CPU:	Total CPU time in hours:min:sec achieved by PHA (resp. CPLEX).

As before, when CPLEX does not solve the linear relaxation of the problem after the maximum CPU time (2 h), the results are indicated with "–."

From Table 5, we can see that when $k = 3$ and $L = 3$, PHA outperforms CPLEX in producing upper bounds for 14 instances over 20. For the 6 instances, the solutions produced by PHA are close (at most 4.30% higher) to those obtained by CPLEX. It should be noticed that the solutions produced by PHA are obtained within a CPU time relatively short compared to CPLEX. Indeed, for PHA, 10 instances over 20 are solved in less than 30 min and 2 instances are solved in less than 40 min, while CPLEX reaches the maximum CPU (2 h) for all the instances except one. We also notice that PHA is able to produce both upper and lower bounds for all the instances, while CPLEX fails in producing a lower bound for large instances, even after 2 h of CPU time.

The observations are the same for the case where $L = 4$ and $L = 5$, with $k = 3$ (Tables 6 and 7). We observe that PHA outperforms CPLEX in producing upper bounds for 15 instances and 14 instances, when $L = 4$ and $L = 5$, respectively. Also, the CPU time spent by PHA is less than 30 min for 10 instances over 20 when $L = 4$, and 9 instances over 20 when $L = 5$, while CPLEX reaches, in both cases $L = 4$ and $L = 5$, the maximum CPU time for almost all the instances. Finally, as for $L = 3$, PHA produces both upper and lower bounds for all the instances, in particular for large instance, while CPLEX does not produce even a lower bound after 2 h of CPU time for 9 instances and 12 instances, when $L = 4$ and $L = 5$, respectively.

Now we compare PHA with each of its components. We do this comparison only in the case where $k = 3$ and $L = 3$, as our experiments lead to the same observations when $L = 4$ and $L = 5$. Table 8 gives the lower and upper bounds obtained by PHA and LRA and the upper bounds obtained by GA and SH, for the kHNDP with $k = 3$ and $L = 3$.

Table 8 clearly shows that PHA outperforms LRA, GA and SH taken separately, for all the instances. Indeed, the upper bound produced by PHA is better for all the instances than those obtained by GA, LRA and SH. However, when comparing the lower bounds obtained by PHA and LRA, except 2 instances (kroA200-a100 and lin318-a159), the lower bound produced by LRA is better than that obtained by PHA. Consequently, as before, the hybridization clearly helps in producing better feasible solutions for the kHNDP but does not produce better lower bounds.

5 The impact of the parallelization

In the previous sections, we have shown that using the Lagrangian relaxation, genetic and greedy algorithms in parallel allows an improvement of the quality of the feasible solutions known for the kESNDP and kHNDP. However, an important question is whether it is relevant to use parallel computing inside each component of PHA. In this section, we investigate this question. Indeed, as mentioned above, solving each subproblem of the Lagrangian relaxation (*i.e.,* LRA) and the greedy (*i.e.,* SH) algorithms reduces to $|D|$ independent minimum cost flow problems, which can be done in $\mathcal{O}(|V|^2|E|\log(|V|C))$ using $|D|$ processors in parallel. Also, in the genetic algorithm (*i.e.,* GA), parallel computing is used in the reproduction phase at each iteration, where, in parallel, we cross several pairs of parent solutions.

The question raised here is important since the CPU time needed to run an algorithm using several processors (for instance, for accessing a shared memory, for managing the interactions with the operating system or between the processors) may exceed the CPU time when using a single processor.

In our approach, because of the nested structure of the parallelism, we have used two libraries, OpenMP and the C++11 inner parallelism library `std::thread`. OpenMP is used in the deepest level, whereas the latter is used in the higher level. We have tested different combinations of libraries and the one with OpenMP and `std::thread` have shown more efficient.

In addition, to avoid incoherences in the solutions population pool that can be caused by the simultaneously access by the threads, we used the primitive `std::mutex` proposed by C++11 to protect the shared data.

In order to measure the impact of the parallelization in our hybrid algorithm, we have considered the CPU time for each algorithm LRA, SH and GA used for the kHNDP with $k = 3$ and $L = 3$. We have used $p = 1, 2, 4, 8$ processors. Figures 5, 6 and 7 present the CPU time evolution

Table 5 Results for PHA and CPLEX for the kHNDP and arbitrary demands with $k = 3$ and $L = 3$

Instances			PHA				CPLEX			
Name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	LB	Gap	CPU
berlin	30	10	10,695	9467.6	11.48	00:00:37	10,254	10,170.49	0.81	00:00:29
berlin	30	15	13,743	11,847.4	13.79	00:01:12	13429	12,829.37	4.47	02:00:00
berlin	52	10	10,298	9136	11.28	00:01:46	9919	9851.46	0.68	00:00:33
berlin	52	15	13,416	11,596.6	13.56	00:02:54	13,278	12,575.49	5.29	02:00:00
berlin	52	20	16,534	13,448.6	18.66	00:04:06	15,891	14,612.79	8.04	02:00:00
st	70	15	1389	971.2	30.07	00:03:21	1336	1119.59	16.20	02:00:00
st	70	26	2107	1312.4	37.72	00:07:09	2176	1610.73	25.98	02:00:00
st	70	35	2834	1714.4	39.50	00:11:37	5299	2142.75	59.56	02:00:00
kroA	100	20	62128	37,704.6	39.31	00:12:47	66,738	44,958.93	32.63	02:00:00
kroA	100	35	103,164	55,049.4	46.64	00:26:12	348,018	68,944.14	80.19	02:00:00
kroA	100	50	14,4842	72,889.6	49.68	00:40:42	55,8391	92,782.7	83.38	02:00:00
kroA	150	30	93,542	45,123	51.76	00:48:19	204,665	61,908.4	69.75	02:00:00
kroA	150	50	14,4861	60,073.2	58.53	01:39:41	–	–	–	02:00:00
kroA	150	75	209,575	75,856.6	63.80	02:00:00	–	–	–	02:00:00
kroA	200	40	114,116	46,091.6	59.61	02:00:00	442,010	75,495.36	82.92	02:00:00
kroA	200	75	212,504	66,209.2	68.84	02:00:00	–	–	–	02:00:00
kroA	200	100	274,917	67,317	75.51	02:00:00	–	–	–	02:00:00
lin	318	61	54,721	20,638.8	62.28	02:00:00	–	–	–	02:00:00
lin	318	111	100,666	26,664.8	73.51	02:00:00	–	–	–	02:00:00
lin	318	159	149,070	32,224.6	78.38	02:00:00	–	–	–	02:00:00

Table 6 Results for PHA and CPLEX for the kHNDP and arbitrary demands with $k = 3$ and $L = 4$

Instances			PHA				CPLEX			
Name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	LB	Gap	CPU
berlin	30	10	10, 552	8836.4	16.26	00:00:22	9433	9433.0	0	01:46:10
berlin	30	15	13, 774	10, 638.4	22.76	00:00:39	11,898	11,194.7	5.91	02:00:00
berlin	52	10	10, 457	8396.0	19.71	00:01:24	9252	8903.7	3.76	02:00:00
berlin	52	15	13, 672	10,283.6	24.78	00:02:20	12021	11,011.2	8.4	02:00:00
berlin	52	20	16,541	11,844.6	28.39	00:03:12	15,541	12,787.4	17.72	02:00:00
st	70	15	1326	825.5	37.75	00:03:57	1346	910.4	32.37	02:00:00
st	70	26	2023	1043.2	48.43	00:07:13	2572	1263.9	50.86	02:00:00
st	70	35	2818	1209.6	57.08	00:08:44	5588	1613.4	71.13	02:00:00
kroA	100	20	60,786	28,152.7	53.69	00:14:11	71728	34,511.5	51.89	02:00:00
kroA	100	35	100,081	39,413.7	60.62	00:25:33	–	–	–	02:00:00
kroA	100	50	14,1228	50,301.7	64.38	00:37:25	386,054	62,125.6	83.91	02:00:00
kroA	150	30	89,109	35,455.9	60.21	00:50:58	204,376	44,932.4	78.01	02:00:00
kroA	150	50	141,829	46,229.2	67.40	01:27:57	–	–	–	02:00:00
kroA	150	75	204,943	58,525.0	71.44	02:00:00	–	–	–	02:00:00
kroA	200	40	107,850	35,872.6	66.74	02:00:00	–	–	–	02:00:00
kroA	200	75	209,367	47,121.1	77.49	02:00:00	–	–	–	02:00:00
kroA	200	100	269,006	48,113.7	82.11	02:00:00	–	–	–	02:00:00
lin	318	61	53,359	17,987.0	66.29	02:00:00	–	–	–	02:00:00
lin	318	111	98,525	17,786.1	81.95	02:00:00	–	–	–	02:00:00
lin	318	159	141,712	15,760.9	88.88	02:00:00	–	–	–	02:00:00

Table 7 Results for PHA and CPLEX for the kHNDP and arbitrary demands with $k = 3$ and $L = 5$

Instances			PHA				CPLEX			
Name	$\|V\|$	$\|D\|$	UB	LB	Gap	CPU	UB	LB	Gap	CPU
berlin	30	10	9989	8527.6	14.63	00:00:47	8897	8897.0	0.00	00:06:24
berlin	30	15	12, 651	10,156.4	19.72	00:01:11	11,397	10,647.4	6.58	02:00:00
berlin	52	10	9967	8073.6	19.00	00:02:39	8855	8511.1	3.88	02:00:00
berlin	52	15	12, 848	9737.3	24.21	00:04:36	11,197	10,466.3	6.53	02:00:00
berlin	52	20	15, 870	11, 211.8	29.35	00:05:59	14,981	12,051.5	19.55	02:00:00
st	70	15	1242	733.4	40.95	00:06:38	1187	827.2	30.31	02:00:00
st	70	26	1808	954.7	47.20	00:10:18	4257	1120.7	73.68	02:00:00
st	70	35	2525	1172.1	53.58	00:15:33	–	–	–	02:00:00
kroA	100	20	56, 668	25, 532.4	54.94	00:23:19	71,110	30,451.0	57.18	02:00:00
kroA	100	35	90, 788	34, 670.5	61.81	00:40:52	–	–	–	02:00:00
kroA	100	50	125, 022	43, 359.5	65.32	00:59:12	–	–	–	02:00:00
kroA	150	30	80, 854	29, 698.5	63.27	01:19:47	–	–	–	02:00:00
kroA	150	50	126, 533	39,383.0	68.88	02:00:00	–	–	–	02:00:00
kroA	150	75	188,290	47,148.4	74.96	02:00:00	–	–	–	02:00:00
kroA	200	40	102,520	29,630.6	71.10	02:00:00	–	–	–	02:00:00
kroA	200	75	193,169	38,005.0	80.33	02:00:00	–	–	–	02:00:00
kroA	200	100	246,339	39,859.9	83.82	02:00:00	–	–	–	02:00:00
lin	318	61	50,570	14,214.8	71.89	02:00:00	–	–	–	02:00:00
lin	318	111	94,004	13,372.0	85.78	02:00:00	–	–	–	02:00:00
lin	318	159	137,215	12,026.7	91.24	02:00:00	–	–	–	02:00:00

Table 8 Results for PHA versus LRA, SH and GA for the kHNDP and arbitrary demands with $k = 3$ and $L = 3$

Instances			PHA		LRA		*SH*	*GA*
Name	$\|V\|$	$\|D\|$	UB	LB	UB	LB	UB	UB
berlin	30	10	10, 695	9467,6	11, 914	9566	11, 756	11, 376
berlin	30	15	13, 743	11,847.4	17, 198	12, 091	14, 854	14, 854
berlin	52	10	10, 298	9136	13, 715	9361	11, 359	10, 980
berlin	52	15	13, 416	11,596,6	19, 042	11, 828	14, 508	14, 491
berlin	52	20	16, 534	13,448.6	22,341	13,715	17,633	17,633
st	70	15	1389	971.2	2213	1038	1536	1424
st	70	26	2107	1312.4	3275	1421	2269	2192
st	70	35	2834	17,144	4665	1822	2975	2957
kroA	100	20	62,128	37,704.6	110,812	37,839	66,190	64,081
kroA	100	35	103,164	55,049.4	183,685	56,581	111,403	105,224
kroA	100	50	14,4842	72,889.6	269,705	74,624	151,175	147,820
kroA	150	30	93,542	45,123	165,239	46,682	101,709	95,986
kroA	150	50	144,861	60,073.2	270,609	68,616	149,949	146,986
kroA	150	75	209,575	75,856.6	420,836	75,891	214,646	212,735
kroA	200	40	114,116	46,091.6	223,098	48,169	117,931	114,989
kroA	200	75	212,504	66,209.2	437,497	76,827	218,219	214,878
kroA	200	100	274,917	67,317	565,116	59,827	278,363	273,934
lin	318	61	54,721	20,638.8	74,755	21,638.8	56,285	55,544
lin	318	111	100,666	26,664.8	137,906	26,664.8	103,297	101,115
lin	318	159	149,070	32,224.6	206,681	30,224.6	152,685	150,965

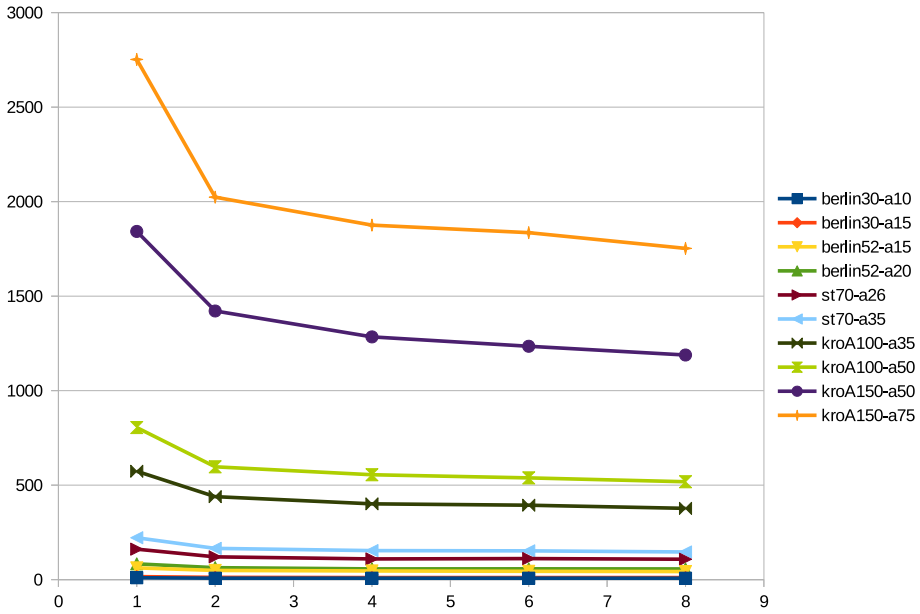

Fig. 5 The CPU time of LRA w.r.t. to the number of processors

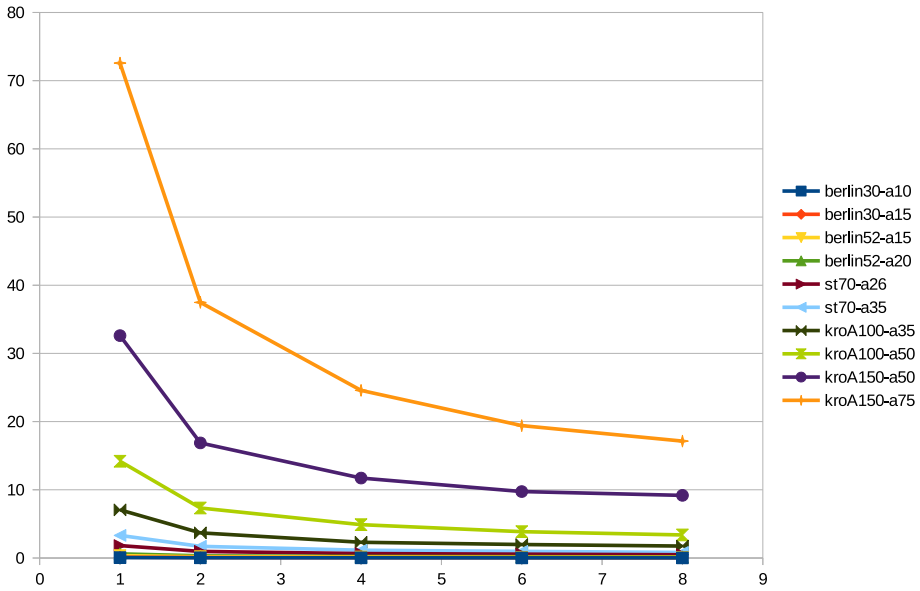

Fig. 6 The CPU time of SH w.r.t. to the number of processors

curve for LRA, SH and GA, respectively, as a function of the number of processors used to run the algorithms. We have limited the number of processors to 8 since the computer we have used for our experiments is equipped with a 8-cores processor.

The three figures clearly show that, for the three algorithms, the CPU time decreases when the number of processors increases. Moreover, using two processors significantly decreases the CPU compared to the usage of a single processor. We can also notice that, even if the CPU time decreases as the number of the processors increases, the gain of using eight processors is not significant compared to using two processors. In conclusion of these experiments, using several processors improve the CPU time of each component of PHA, and using two processors seems to be the best choice.

It should be noticed that the above conclusion holds if we do not use more than eight processors, and depends on the architecture of the computer used for the experiments. Using a different architecture and more than eight processors may lead to different conclusions.

6 Conclusion

In this paper, we have proposed two hybrid parallel algorithms for solving the *k*ESNDP and the *k*HNDP. The

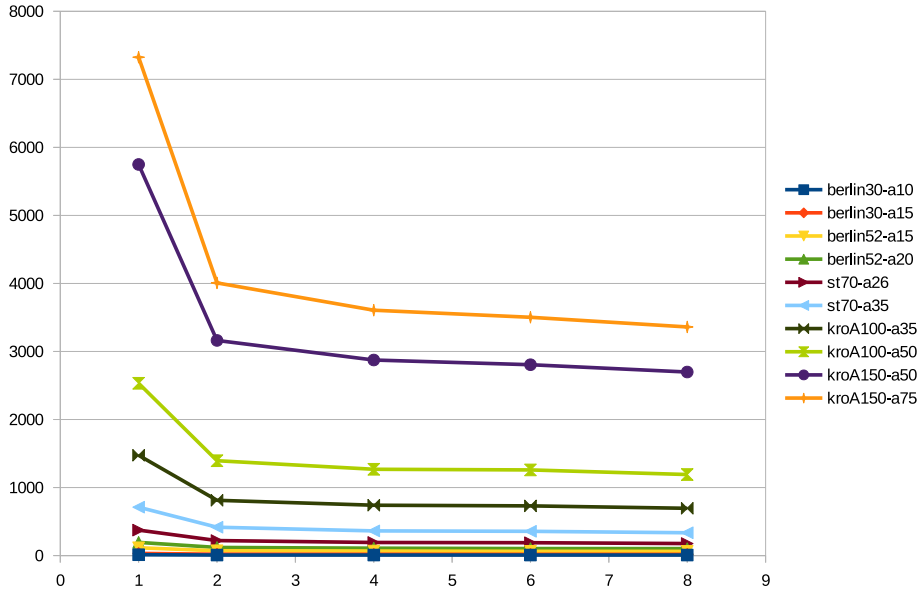

Fig. 7 The CPU time of GA w.r.t. to the number of processors

algorithms are based on a Lagrangian relaxation algorithm, a genetic algorithm and a greedy algorithm, and aims in producing both lower and upper bound for the two problems. The experiments conducted in the paper have shown that our hybrid algorithm outperforms CPLEX in producing good feasible solutions, even for large size instances, and this within a relatively short CPU time. They have also shown that the hybridization of the three components outperforms, in most cases, each component taken separately. Finally, we have shown that using several processors inside each component of the hybrid algorithm helps in decreasing the CPU time for each component.

The parallel computing framework we have proposed in this paper is generic and can be applied to all the network design problems, and not only, whose integer programming formulations have the same block structure as the *k*ESNDP and *k*HNDP. Also, the implementation we have proposed is quite simple and can be easily adapted to other problems.

It should be noticed that our algorithm PHA is heuristic, and contrarily to many heuristics, is able to produce both upper and lower bounds of the optimal solution. This can give an indication on the quality of the feasible solution obtained, in particular when the gap between the lower and upper bounds is small. Indeed, a very small gap (0% in the better case) means that the feasible solution obtained is close to the optimal solution of the problem. We can observe that in all our experiments the gaps between the lower and upper bounds are quite large (more than 40% for most of the instances). Thus, it would be interesting to further investigate a way of producing better lower bounds in order to know how close the solutions produced by algorithm PHA are to the optimal solution.

It would also be interesting to further investigate the parallelization of the hybrid algorithm and the use of more sophisticated architectures, like GPUs or grid computing.

Finally, one can also think of using the techniques proposed in this paper for devising efficient exact (Branch-and-Bound, Branch-and-Cut, Branch-and-Price, etc.) algorithms for the *k*ESNDP and *k*HNDP, and all the problems having the same structure.

Acknowledgements We would like to thank the anonymous referees for their valuable comments that permitted to improve the presentation of the paper

Compliance with ethical standards

Conflict of interest Dr. Ibrahima Diarrassouba, Mohamed Khalil Labidi and Prof. A. Ridha Mahjoub declare that they have no conflict of interest.

Human and animal ethical standards This article does not contain any studies with human participants or animals performed by any of the authors.

References

Ahuja RK, Magnanti TL, Orlin JB (1993) Network flows: theory, algorithms, and applications. Prentice-Hall Inc., Upper Saddle River

Beasley J E (1993) Modern heuristic techniques for combinatorial problems. Chapter lagrangian relaxation. Wiley, New York, pp 243–303

Bendali F, Diarrassouba I, Didi Biha M, Mahjoub AR, Mailfert J (2010) A branch-and-cut algorithm for the *k*-edge connected subgraph problem. Networks 55(1):13–32

Botton Q, Fortz B, Gouveia L, Poss M (2013) Benders decomposition for the hop-constrained survivable network design problem. INFORMS J Comput 25(1):13–26

CPLEX (12.5). IBM CPLEX. http://www-01.ibm.com/software/info/ilog

Dahl G, Gouveia L (2004) On the directed hop-constrained shortest path problem. Oper Res Lett 32(1):15–22

Diarrassouba I, Gabrel V, Gouveia L, Mahjoub AR, Pesneau P (2016a) Integer programming formulations for the k-edge-connected 3-hop-constrained network design problem. Networks 67(2):148–169

Diarrassouba I, Mahjoub AR, Kutucu H (2016b) Two node-disjoint hop-constrained survivable network design and polyhedra. Networks 67(4):316–337

Diarrassouba I, Mahjoub AR, Yaman H (2017) Integer programming formulations for k-node-connected hop-constrained survivable network design problem. In: Cahier du LAMSADE, vol 382

Goemans MX, Bertsimas DJ (1993) Survivable networks, linear programming relaxations and the parsimonious property. Math Program 60(1–3):145–166

Grötschel M, Monma CL (1990) Integer polyhedra arising from certain network design problems with connectivity constraints. SIAM J Discrete Math 3(4):502–523

Grötschel M, Monma CL, Stoer M (1992) Facets for polyhedra arising in the design of communication networks with low-connectivity constraints. SIAM J Optim 2(3):474–504

Held M, Karp RM (1971) The traveling-salesman problem and minimum spanning trees: part ii. Math Program 1(1):6–25

Huygens D, Labbé M, Mahjoub AR, Pesneau P (2007) The two-edge connected hop-constrained network design problem: valid inequalities and branch-and-cut. Networks 49(1):116–133

Kerivin H, Mahjoub AR (2005) Design of survivable networks: a survey. Networks 46(1):1–21

Kerivin H, Mahjoub AR, Nocq C (2004) (1, 2)-Survivable networks: facets and branch-and-cut. In: Grötschel M (ed) The Sharpest Cut, MPS-SIAM series in Optimization. SIAM, pp 121–152

Magnanti TL, Raghavan S (2005) Strong formulations for network design problems with connectivity requirements. Networks 45(2):61–79

Steiglitz K, Weiner P, Kleitman D (1969) The design of minimum-cost survivable networks. IEEE Trans Circuit Theory 16(4):455–460

Talbi E-G (2002) A taxonomy of hybrid metaheuristics. J Heuristics 8(5):541–564

TSPLIB (1995) Gerhard reinelt. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95. Accessed 13 Dec 2015

Winter P (1987) Steiner problem in networks: a survey. Networks 17(2):129–167