A parallel heuristic for the uniform capacitated vertex k-center problem[1]

José Alejandro Cornejo Acosta
Instituto Nacional de Astrofísica, Óptica y Electrónica
Computer Science Department, Puebla, Mexico
E-mail: *alexcornejo@inaoep.mx*

Jesús García Díaz
Instituto Nacional de Astrofísica, Óptica y Electrónica
Computer Science Department, Puebla, Mexico
E-mail: *jesgadiaz@gmail.com*

Julio César Pérez Sansalvador
Instituto Nacional de Astrofísica, Óptica y Electrónica
Computer Science Department, Puebla, Mexico
E-mail: *jcp.sansalvador@inaoep.mx*

Roger Z. Ríos-Mercado
Universidad Autónoma de Nuevo León (UANL)
Graduate Program in Systems Engineering, San Nicolás de los Garza, Mexico
E-mail: *roger.rios@uanl.edu.mx*

Saúl Eduardo Pomares Hernández
Instituto Nacional de Astrofísica, Óptica y Electrónica
Computer Science Department, Puebla, Mexico
E-mail: *spomares@inaoep.mx*

October 2021

[1]Preprint PISIS-RR21-04, Graduate Program in Systems Engineering, UANL, San Nicolás de los Garza, Mexico, October 2021

Abstract

The uniform capacitated vertex k-center problem is an $\mathcal{NP}$-hard combinatorial optimization problem that models real situations where k centers can only attend a maximum number of clients, and the travel time or distance from the clients to their assigned center has to be minimized. This paper introduces a polynomial-time parallel constructive heuristic that exploits the relationship between this problem and the minimum capacitated dominating set problem. Besides, the proposed heuristic is based on the one-hop farthest-first heuristic that has proven effective for the uncapacitated version of the problem. We performed different empirical evaluations of the proposed algorithm, including an analysis on the effect of parallel computing, which significantly improved the running time for relatively large instances.

Keywords: Combinatorial optimization; Capacitated k-Center Problem; Heuristic search; Parallel heuristics.

1 Introduction

The uniform capacitated vertex k-center problem is an $\mathcal{NP}$-hard problem from the family of Location Problems [6, 22]. This problem receives as input a complete weighted graph $G = (V, E)$ with edge weights that follow a metric. Its goal is to find a set $C \subseteq V$ with at most k centers and an assignment $f_C : V \setminus C \to C$, such that the number of vertices assigned to each center is at most L and the weight of the edge connecting the farthest vertex to its assigned center is minimum [22]. Variables $k, L \in \mathbb{Z}^+$ are part of the input. In the non-uniform version of the problem, there is a demand $d_i \in \mathbb{Z}^+$ and a capacity $L_i \in \mathbb{Z}^+$ associated with each vertex v_i. The goal is to select centers and assign vertices to them in such a way that the total demand of the vertices does not exceed the capacity of their assigned centers. However, in this paper, we deal with the special case where the demand of every vertex is 1, and the capacity of each center is L, i.e., the uniform version. A typical application of this problem is the location of facilities, such as police stations, hospitals, supermarkets, and schools, in a context where the capacity of the facilities is limited and the travel time or distance from the clients to the facilities has to be minimized.

Aditionally, the uniform capacitated vertex k-center problem is a generalization of the uncapacitated vertex k-center problem. The only difference between these problems is that any number of vertices can be assigned to each center in the uncapacitated version [19, 12, 20]. The heuristic introduced in this paper is based on the main mechanism exploited by some of the most effective heuristics for the uncapacitated version of the problem. We refer to this mechanism as the one-hop farthest-first heuristic [16]. While many algorithms have been designed for the uncapacitated vertex k-center problem [19, 12, 20, 34, 27, 16, 30, 32, 17, 18], much less has been proposed for the capacitated version.

The capacitated vertex k-center problem has been approached through different algorithmic perspectives, such as exact, heuristic, metaheuristic, and approximation algorithms. Regarding exact algorithms, these struggle to find optimal solutions for instances from benchmark datasets with just some hundreds of vertices [26, 2, 23, 9]. Regarding heuristics and metaheuristics, they rely mostly on exploitation, and the experimental evidence shows that they are among the fastest for finding near-optimal solutions [33, 29]. Finally, while some conceptually simple algorithms achieve the best possible approximation factor for the uncapacitated version ($\rho = 2$ under $\mathcal{P} \neq \mathcal{NP}$) [19, 20, 16], the best-known approximation algorithms for the capacitated versions are conceptually more complicated and have an approximation factor of 6 and 9 for the uniform and non-uniform versions, respectively [22, 3]. All the algorithms designed for the non-uniform version of the problem can find solutions for the uniform version too. Nevertheless, this paper introduces a parallel constructive heuristic specifically designed for the uniform version of the problem. This heuristic takes advantage of the relationship between the uniform capacitated vertex kenter problem and the minimum capacitated dominating set problem [9]. Besides, it is based on the one-hop farthest-first heuristic [16].

The remaining part of the document is organized as follows. Section 2 presents a brief literature review for the capacitated vertex k-center problem, including a reduction from the uniform version of the problem to the minimum capacitated dominating set problem. Based on this reduction, Section 3 introduces the proposed heuristic and the parallel computing strategy that improves its performance. Specifically, we used the Single Instruction Multiple Data model (SIMD) [15]. Section 4 presents an empirical evaluation of the proposed heuristic. Finally, Section 5 presents the concluding remarks and possible future work.

2 The uniform capacitated vertex k-center problem

The uniform capacitated vertex k-center problem is an $\mathcal{NP}$-hard problem that receives as input a complete weighted graph $G = (V, E)$ and two positive integers k and L. The goal is to find a set $C \subseteq V$ with at most k vertices and an assignment $f_C : V \setminus C \to C$, such that the weight of the edge connecting the farthest vertex $v \in V \setminus C$ to its assigned center $c \in C$ is minimum. Besides, the number of vertices assigned to each center cannot be greater than L. Equation 1 shows the fitness function of the problem where $r(C, f_C)$ must be minimized:

$$r(C, f_C) = \max_{v \in V \setminus C} w(\{v, f_C(v)\}) \tag{1}$$

where

- $C \subseteq V$ is the set of centers,
- $f_C : V \setminus C \to C$ is an assignment, where $\forall v \in C, |\{(u, v) \in f_C\}| \leq L$,
- and $w(\{v_1, v_2\})$ is the weight of edge $\{v_1, v_2\}$.

Among the approximation algorithms for this problem are the 10-approximation algorithm of Barilan et al. [4], the 6-approximation algorithm of Khuller and Sussmann [22], and the 6-approximation algorithm of An et al. [3]. To date, no one has found an algorithm with a better approximation factor for this problem. In the case of the non-uniform capacitated vertex k-center problem, the situation is not better; the best-known approximation algorithm generates 9-approximated solutions [3]. Now, since the uniform version of the problem is a particular case of the non-uniform version, all the algorithms designed for the non-uniform version get feasible solutions for the uniform version too. Regarding heuristic and metaheuristic algorithms for the non-uniform version, most of them are based on a constructive phase followed by local search. These include Greedy Randomized Adaptive Search Procedures (GRASP) [28], Large-Scale Local Search [33], Iterated Greedy Local Search [28, 29], and Variable Neighborhood Descent [28, 29]. The experimental evidence shows that these algorithms are among the fastest for finding near-optimal solutions. Regarding exact algorithms, there are some proposals based on Integer and Mixed Integer Programming formulations [26, 23, 2]. As expected, solving these formulations becomes rapidly impractical as the input size grows. All the mentioned heuristics and metaheuristics are sequential, while the algorithm proposed in this paper has been designed to exploit parallelization as much as possible.

Through experimentation, many authors have observed that the exact resolution of the classical formulations for the uncapacitated and capacitated vertex k-center problems tends to be very time-consuming [25, 26]. For this reason, many alternative integer programming or mixed integer programming formulations have been proposed for both problems [8, 10, 21, 13, 1, 7, 5, 2, 26]. Some of these formulations are based on the relationship between the vertex k-center problem and other $\mathcal{NP}$-hard problems, such as the minimum dominating set problem [10, 24]. As an instance, Expressions (2) to (7) introduces a quadratic integer programming formulation for the uniform capacitated vertex k-center problem. This formulation is based on a reduction from this problem to the minimum capacitated dominating set problem. The following definitions may help understand this formulation.

Definition 2.1. *Given a graph $G = (V, E)$, a dominating set is a set $D \subseteq V$ such that for every vertex $v \in V \setminus D$, there is a vertex $u \in D$ such that $\{v, u\} \in E$.*

Definition 2.2. *A minimum dominating set is a set of minimum cardinality among all the dominating sets.*

Definition 2.3. *Given a graph $G = (V, E)$, and a capacity function $f_{cap} : V \to \mathbb{Z}^+$. A capacitated dominating set $D \subseteq V$ is a set such that every vertex $v \in V \setminus D$ is assigned to some vertex $u \in D \cap N(v)$, and the number of vertices assigned to each vertex $u \in D$ is not greater than its capacity $f_{cap}(u)$, where $N(u)$ is the open neighborhood of $u \in V$.*

Definition 2.4. *A minimum capacitated dominating set is a set of minimum cardinality among all the capacitated dominating sets.*

Definition 2.5. *Given a weighted graph $G = (V, E)$, a bottleneck graph $G_r = (V, E_r)$ is such that E_r consists of all the edges in E with weight less than or equal to r.*

Theorem 1. *The minimum capacitated dominating set over the bottleneck graph $G_{r^*} = (V, E_{r^*})$ is the optimal solution to the capacitated vertex k-center problem over the original input graph $G = (V, E)$, where r^* is the size of the optimal solution to the latter problem [9].*

It is well established that the uncapacitated vertex k-center problem is equivalent to the minimum dominating set problem when the size $r(C^*)$ of the optimal solution C^* is known ahead of time [24, 17, 11]. This same situation occurs with the capacitated vertex k-center, which is equivalent to the minimum capacitated dominating set problem when the size $r(C^*, f_{C^*})$ of the optimal solution (C^*, f_{C^*}) is known ahead of time (Theorem 1) [9]. The following quadratic integer programming formulation models this problem.

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} y_i && && (2)\\
\text{subject to} \quad & \sum_{i=1}^{n} x_{ij} \leq L && \forall j \in \{1, 2, ..., n\} && (3)\\
& \sum_{i=1}^{n} x_{ji} = 1 - y_j, && \forall j \in \{1, 2, ..., n\} && (4)\\
& x_{ij} \leq y_j, && \forall i, j \in \{1, 2, ..., n\} && (5)\\
& x_{ij} \leq a_{ij}, && \forall i, j \in \{1, 2, ..., n\} && (6)\\
& x_{ij}, y_i \in \{0, 1\}, && \forall i, j \in \{1, 2, ..., n\} && (7)
\end{aligned}$$

where

$$a_{ij} = \begin{cases} 1, & \text{if } w(\{v_i, v_j\}) \leq r(C^*, f_{C^*}) \text{ and } i \neq j \\ 0, & \text{otherwise} \end{cases} \qquad (8)$$

In this formulation, the variables $y_i \in \{0, 1\}$ indicate which vertices are part of the minimum capacitated dominating set. Namely, if $y_i = 1$, then v_i is in the minimum capacitated dominating set. For convenience, let us refer to these vertices as centers. The variables $x_{ij} \in \{0, 1\}$ indicates which vertices are assigned to which center, constraint (3) guarantees that the number of vertices assigned to each center is not greater than L, constraint (4) indicates that every non-center vertex has to be assigned to exactly one center and that centers must not be assigned to any other center. Finally, constraint (5) indicates that every non-center vertex has to be assigned only to centers and constraint (6) indicates that every non-center vertex can be assigned only to vertices in its neighborhood, which is established by the size $r(C^*, f_{C^*})$ of the optimal solution (C^*, f_{C^*}) for the uniform capacitated vertex k-center problem [9].

In summary, in the same way that the uncapacitated vertex k-center problem can be reduced to the minimum dominating set problem [24, 11, 17], the uniform capacitated vertex k-center problem can be reduced to the minimum capacitated dominating set problem [9]. Since many successful heuristics for the uncapacitated version are based on this relationship, we conjecture that the capacitated version can be approached similarly [16, 20, 32]. However, to exploit this relationship, the size $r(C^*, f_{C^*})$ of the optimal solutions has to be known in advance, which is not possible for arbitrary graphs. However, this issue can be lessened by performing binary search over the set of possible values of $r(C^*, f_C^*)$. At each iteration, the minimum capacitated dominating set problem is heuristically approached over the bottleneck input graph $G = (V, E_r)$, where r is a guess on the actual value of $r(C^*, f_C^*)$. The following section illustrates the specifics of this algorithm.

3 Proposed heuristic

This section introduces a new parallel constructive polynomial-time heuristic for the uniform capacitated vertex k-center problem. We refer to this proposal as the OHCKC heuristic. This heuristic is based on the relationship between the uniform capacitated vertex k-center problem and the minimum capacitated dominating set problem. Besides, it generalizes the one-hop farthest-first heuristics for the uncapacitated version [17, 16]. Another essential feature that distinguishes OHCKC from other heuristics from the literature is that it exploits parallelization. Algorithm 1 shows the pseudocode of the OHCKC heuristic.

In order to exploit the relationship between the uniform capacitated vertex k-center problem and the minimum capacitated dominating set problem, the size of the optimal solution of the former problem must be known in advance. One way to sort this out is by solving the minimum capacitated dominating set problem with all the possible values of the solution size, which are the elements of the set of edge weights. Nonetheless, since there are $(n^2 - n)/2$ edges, this can be very time-consuming, where $n = |V|$. So, in order to get a more practical execution time, the proposed heuristic executes a binary search over the ordered set of edge weights. Conceptually, OHCKC (Algorithm 1) is similar to the exact procedure reported in Cornejo Acosta et al. [9]. However, instead of solving each minimum capacitated dominating set problem to optimality, the proposed heuristic uses the *GetFeasibleSolution* procedure to get a near-optimal feasible solution (line 9 of Algorithm 1). The *GetFeasibleSolution* procedure is based on a mechanism used by the CDSh heuristic, which is among the algorithms for the uncapacitated version of the problem with better practical performance [16]. We refer to this mechanism as the one-hop farthest-first heuristic.

Due to its heuristic nature, *GetFeasibleSolution* will not necessarily return optimal solutions. For this reason, the proposed heuristic has to keep track of the best solution (C, f_C) that has been returned (lines 10 to 13 of Algorithm 1). Then, every time a better solution is found, *high* is set to $mid - 1$, allowing to keep searching for a better solution. Otherwise, *low* is set to $mid + 1$ (lines 14 to 18 of Algorithm 1). Finally, it may be the case that the binary search (lines 6 to 19) returns a solution with less than k centers. If

Algorithm 1: OHCKC

```
Input: A complete graph $G = (V, E)$, a positive integer $k$, a positive integer $L$, and a non-decreasing
       list of the $m$ edge weights of $G$, i.e., $w(e_1), w(e_2), ..., w(e_m)$, where $w(e_i) \leq w(e_{i+1})$
Output: A set of vertices $C \subseteq V$, $|C| \leq k$,
        and an assignment $f_C : V \setminus C \to C$
1  $high = m$ ;
2  $low = 1$ ;
3  $C = \emptyset$ ;
4  $f_C = \emptyset$ ;
5  $r(C) = \infty$ ;
6  while $low \leq high$ do
7      $mid = \lfloor (high + low)/2 \rfloor$ ;
8      $r = w(e_{mid})$ ;
9      $(C', f_{C'}) = GetFeasibleSolution(G, k, r, L)$ ;
10     if $r(C') \leq r(C)$ then
11         $(C, f_C) = (C', f_{C'})$ ;
12         $r(C) = r(C')$ ;
13     end
14     if $r(C) \leq w(e_{mid})$ then
15         $high = mid - 1$ ;
16     else
17         $low = mid + 1$ ;
18     end
19 end
   // Optional
20 if $|C| < k$ then
21     convert the $k - |C|$ farthest vertices into centers ;
22 end
   // Optional
23 foreach $c_i \in C$ do
24     $X = dom(f_{c_i}) \cup c_i$ ;
25     $c_j = \arg\ \min_{u \in X}\{\max_{v \in X}\{distance(u, v)\}\}$ ;
26     $f_C = f_C \setminus f_{c_i}$ ;
27     foreach $v \in X \setminus \{c_j\}$ do
28         $f_C = f_C \cup \{(v, c_j)\}$ ;
29     end
30 end
31 return $(C, f_C)$ ;
```

this happens, lines 20 to 22 of Algorithm 1 will complete the solution by converting the farthest non-center vertices into centers. This way, the farthest vertices, which define the size of the solution, are unassigned so that the solution size may be reduced. Then, lines 23 to 30 of Algorithm 1 will solve the 1-center problem over the set of vertices assigned to each center. These two last procedures are optional, but they may improve the solution's fitness. Besides, they do not increase the overall complexity of the heuristic.

Algorithm 2 shows the pseudocode of *GetFeasibleSolution*, which is a crucial piece of OHCKC. This algorithm iteratively constructs a feasible solution starting from the empty set. However, it first constructs a bottleneck graph $G_r = (V, E_r)$ from the original input graph (line 4 of Algorithm 2). Namely, all the edges with cost greater than r are removed. Then, at every iteration $i \in \mathbb{Z}^+$, $1 \leq i \leq k$, it computes each vertice's score, which is the number of vertices in its neighborhood not assigned to any center,

$$\forall v \in V, \quad score(v) = |\{u \in V : u \in N_{G_r}(v) \;\wedge\; u \notin \cup_{j=1}^{j=i-1}(\{c_j\} \cup dom(f_{c_j}))\}| \tag{9}$$

where,

- i - current iteration $i \in \mathbb{Z}^+$, $1 \leq i \leq k$.
- $N_{G_r}(v)$ - open neighborhood of vertex v over the bottleneck graph G_r.
- C - set of centers.
- c_j - center added to C at iteration j.
- $\cup_{j=1}^{j=i-1}\{c_j\}$ - set of centers selected before iteration i.
- f_C - assignment of vertices.
- f_{c_j} - subset of the assignment that contains only duples of the form (u, c_j).
- $dom(f_{c_j})$ - set of vertices assigned to center c_j.

Thus, $\cup_{j=1}^{j=i-1}(\{c_j\} \cup dom(f_{c_j}))$ is the set of centers that have been selected at previous iterations, along with the vertices that have been assigned to them. Now, at the beginning of *GetFeasibleSolution*, none vertex has been assigned (line 3 of Algorithm 2); therefore, the score of each vertex is initialized to $|N_{G_r}(v)|$ (lines 5 to 7 of Algorithm 2). Next, the algorithm tries to find a solution for the minimum capacitated dominating set problem over the bottleneck graph G_r (lines 8 to 23 of Algorithm 2). To do this, *GetFeasibleSolution* iteratively adds vertices of high score in the neighborhood of the farthest unassigned vertex to the set of already selected centers. At the same time, tries to minimize the distance from the unassigned vertices to their farthest center. This can be done in two different ways. The first one corresponds to the original definition of the Critical Dominating Set procedure defined by the CDSh heuristic [16, 17] (lines 12 to 15 of Algorithm 2) and the second one corresponds to *DistanceBasedSelection* (line 11 of Algorithm 2). These two pseudocode segments return a duple (c_i, f_{c_i}), i.e., a center and its assigned vertices. Let us leave the description of these two pseudocode segments for later. For now, let us describe what happens once the duple (c_i, f_{c_i}) is generated. Once having duple (c_i, f_{c_i}), the score of every vertex is updated. That is, every assigned vertex $v \in dom(f_{c_i}) \cup \{c_i\}$ reduces the score of its neighbors in one unit (lines 17 to 21 of Algorithm 2). The score of the neighbors of c_i is also updated because once a vertex is chosen as a center, it should not be covered by any center.

In summary, a center c_i of high score is selected from $N_{G_r}[v_f]$ at every iteration i of the *GetFeasibleSolution* procedure, where v_f is the farthest unassigned vertex from the current partial solution C (line 9 of Algorithm 2). Since solution C is initialized as the empty set (line 2 of Algorithm 2), the vertex v_f is selected at random at iteration $i = 1$. Notice that by selecting centers of high score from $N_{G_r}[f]$, we are exploiting the following intuitive observations. First, since the uniform capacitated vertex k-center problem aims to minimize the distance from the farthest vertex in V to its assigned center, it is necessary to assign the farthest vertex v_f to a center that is close to it, i.e., that is in its neighborhood. Secondly, by greedily selecting a center of high score we are taking the best local decision that brings us closer to a minimum capacitated dominating set.

Now, we proceed to describe the two different ways for generating duples (c_i, f_{c_i}). As said before, the first one corresponds to the original definition of the Critical Dominating Set procedure of the CDSh heuristic [16, 17] (lines 12 to 15 of Algorithm 2) which consists in selecting as a center c_i a vertex $v \in N_{G_r}[v_f]$ of maximum score (line 13 of Algorithm 2). Afterwards, all the vertices in $N_{G_r}[c_i]$ not yet assigned to any other center are assigned to it (line 14 of Algorithm 2). This way of selecting a center will only be executed when the score of all the elements of $N_{G_r}[v_f]$ is less than or equal to L. In other words, under these circumstances, the capacity constraint can be ignored, and the algorithm will work just like the original CDSh heuristic, where the goal is to select as center a vertex that maximizes the number of assigned vertices. However, if there is at least one element of $N_{G_r}[v_f]$ with score greater than L (line 10 of Algorithm 2), then the selection of center c_i and the assignment of vertices is made in a more elaborated manner. Here is where *DistanceBasedSelection* comes into play (line 11 of Algorithm 2).

To give a more precise explanation, let us describe the *DistanceBasedSelection* procedure through a specific test graph. Figure 1 shows this graph, which is a complete graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_6\}$ and E given by the distance matrix of Table 1. The other input variables are $k = 2$ and $L = 2$. Namely, we want to select two centers and assign no more than two vertices to each one. Now, given a covering radius $r = 4$, the bottleneck graph G_r corresponds to Figure 2. We use $r = 4$ because this is precisely the size of the optimal solution for this specific instance.

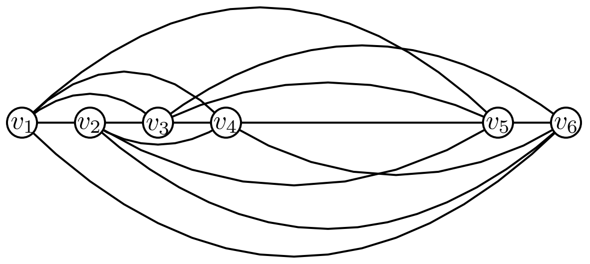

Figure 1: A test graph G.

From Figure 2, we can observe that the initial score (lines 8 to 10 of Algorithm 2) of each vertex is: $score(v_1) = 3$, $score(v_2) = 3$, $score(v_3) = 3$, $score(v_4) = 4$, $score(v_5) = 2$, and $score(v_6) = 1$. Notice that we are not considering self-loops. Then, a vertex v_f of maximal distance to the partial solution C is selected (line 9 of Algorithm 2). Since at the beginning C is the empty set, the vertex v_f is selected at random. In our example, we selected $v_f = v_1$. Then, line 10 of Algorithm 2 checks if vertex v_f or any of its neighbors, $N_{G_r}[v_f] = \{v_1, v_2, v_3, v_4\}$, has a score greater than $L = 2$. In this case, all vertices in $\{v_1, v_2, v_3, v_4\}$ have a

Algorithm 2: *GetFeasibleSolution*

```
Input: A complete graph $G = (V, E)$, a positive integer $k$, a covering radius $r$, and a capacity $L$
Output: A set of vertices $C \subseteq V$, $|C| = k$, and an assignment $f_C : V \setminus C \to C$
1  // $N_{G_r}(v)$ is the set of neighbors of $v$ in $G_r$ ;
2  $C = \emptyset$ ;
3  $f_C = \emptyset$ ;
4  $G_r = BottleneckGraph(G, r)$ ;
5  foreach $v \in V$ do
6      $score(v) = |N_{G_r}(v)|$ ;
7  end
8  for $i = 1$ to $k$ and $|f_C| < |V \setminus C|$ do
9      $v_f$ = A vertex $v \in V \wedge v \notin dom(f_C) \cup C$ such that $distance(v, C)$ is maximal ;
10     if $\exists v \in N_{G_r}[v_f] \setminus (dom(f_C) \cup C)\ :\ score(v) > L$ then
11         $(c_i, f_{c_i}) = DistanceBasedSelection(G, G_r, v_f, C, L, f_C)$ ;
12     else
13         $c_i$ = A vertex $v \in N_{G_r}[v_f] \wedge v \notin dom(f_C) \cup C$ of maximum score ;
14         $f_{c_i} = \{(u, c_i) : u \in N_{G_r}(c_i) \wedge u \notin dom(f_C) \cup C\ \}$;
15     end
16     $f_C = f_C \cup f_{c_i}$ ;
17     foreach $v \in dom(f_{c_i}) \cup \{c_i\}$ do
18         foreach $u \in N_{G_r}(v)$ do
19             $score(u) = score(u) - 1$ ;
20         end
21     end
22     $C = C \cup \{c_i\}$ ;
23 end
24 foreach $v \in V \setminus (dom(f_C) \cup C)$ do
25     $c_j$ = Nearest center in $\{c_i : c_i \in C\ \wedge\ |f_{c_i}| < L\}$ to $v$ ;
26     $f_C = f_C \cup \{(v, c_j)\}$ ;
27 end
28 return $(C, f_C)$ ;
```

Algorithm 3: *DistanceBasedSelection* heuristic

```
Input: A complete graph $G = (V, E)$, a bottleneck graph $G_r = (V, E_r)$, a vertex $v_f \in V$, a set of
       centers $C$, a positive integer $L$, and the assignment of already assigned vertices $f_C$
Output: A center $c'$ and an assignment $f_{c'} : V' \to \{c'\}$, where $V' \subseteq V \setminus C$
1  $c' = null$ ;
2  $f_{c'} = \emptyset$ ;
3  $d = \infty$ ;
4  foreach $v \in N_{G_r}[v_f] \wedge v \notin dom(f_C) \cup C \ : \ score(v) > L$ do
5      $v_{ref}$ = A vertex $u \in V$ such that $distance(u, C \cup \{v\})$ is maximal ;
6      $f_v = \{(u, v) : u \in F_L\}$ , where $F_L$ is the set of the $L$ farthest vertices in $N_{G_r}(v) \setminus (dom(f_C) \cup C)$
        to $v_{ref}$ ;
7      $d_v = distance(v_{ref}, u)$ , where $u$ is the $(L+1)^{th}$ farthest vertex in $N_{G_r}(v) \setminus (dom(f_C) \cup C)$ to
        $v_{ref}$ ;
8      if $d_v < d$ then
9          $d = d_v$ ;
10         $(c', f_{c'}) = (v, f_v)$;
11     end
12 end
13 return $(c', f_{c'})$ ;
```

Table 1: Distance matrix for the test graph G

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	2	3	7	8
v_2	1	0	1	2	6	7
v_3	2	1	0	1	5	6
v_4	3	2	1	0	4	5
v_5	7	6	5	4	0	1
v_6	8	7	6	5	1	0

Table 2: Reference matrix for the test graph G

	Farthest neighbors					
v_1	v_6	v_5	v_4	v_3	v_2	v_1
v_2	v_6	v_5	v_4	v_1	v_3	v_2
v_3	v_6	v_5	v_1	v_2	v_4	v_3
v_4	v_6	v_5	v_1	v_2	v_3	v_4
v_5	v_1	v_2	v_3	v_4	v_6	v_5
v_6	v_1	v_2	v_3	v_4	v_5	v_6

score greater than $L = 2$ (see Figure 3).

The next step is the execution of *DistanceBasedSelection* (Algorithm 3), which consists of a loop that iterates over the set of vertices that are candidates to be selected as a center. The elements of this set are the vertices $v \in N_{G_r}[v_f]$ with a score greater than $L = 2$ not yet covered by any center. In our example, this set is $\{v_1, v_2, v_3, v_4\}$. Starting with vertex v_1, *DistanceBasedSelection* selects a vertex v_{ref} of maximum distance from $C \cup \{v_1\}$. Since C is initialized as the empty set, then $C \cup \{v_1\} = \{v_1\}$; thus, $v_{ref} = v_6$. Next, the farthest L vertices from $N_{G_r}(v_1) \setminus (dom(f_C) \cup C)$ to v_{ref} are assigned to v_1, where f_C is the assignment of the already assigned vertices (to this point, $f_C = \emptyset$). The intuition behind this method is that some center will be selected from the neighborhood of the farthest vertex from C at the next iteration, which will be v_{ref} if vertex v_1 is selected as a center. Thus, it is convenient to keep the distance from the unassigned vertices to v_{ref} as small as possible. Following our example, the distance from v_{ref} to each vertex in $N_{G_r}(v_1) \setminus (dom(f_C) \cup C) = \{v_2, v_3, v_4, v_5, v_6\}$ is consulted from the original input graph G; these values are ordered. Then, the L farthest vertices from v_{ref} are assigned to vertex v_1. This way, $f_{v_1} = \{(v_2, v_1), (v_3, v_1)\}$ and the distance from v_{ref} to its farthest vertex $v \in (N_{G_r}[v_1]) \setminus (dom(f_{v_1}) \cup \{v_1\} \cup dom(f_C) \cup C)$ is assigned to variable d_v (see Figure 4 and Expression 10). Next, this same process is repeated for v_2, v_3, and v_4. Figures 5-7 and Expressions 11-13 show the resulting values. Be aware that for each of these vertices, its

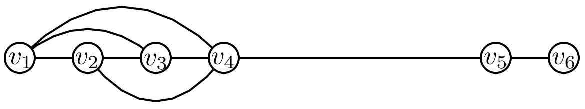

Figure 2: Bottleneck graph G_r with $r = 4$.

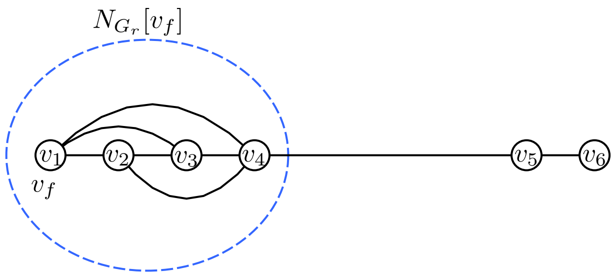

Figure 3: From $N_{G_r}[v_f]$ all vertices in $\{v_1, v_2, v_3, v_4\}$ have score greater than $L = 2$.

v_{ref} vertex may be different. In this case, v_{ref} was coincidentally the same for all vertices. Finally, the vertex selected as a center is the one in $\{v \in V : v \in N_{G_r}[v_f] \wedge v \notin dom(f_C) \cup C \wedge score(v) > L\} = \{v_1, v_2, v_3, v_4\}$ with minimum d_v. In this case, vertices v_1, v_2, and v_3 have the same value $d_{v_1} = d_{v_2} = d_{v_3} = 5$. So, we can select any of them as a center, let's say v_2. Then, the *DistanceBasedSelection* heuristic (Algorithm 3) returns the pair $(c', f_{c'}) = (v_2, f_{v_2})$ to *GetFeasibleSolution* (Algorithm 2), which will add center $c_i = v_2$ to the current solution C (line 22 of Algorithm 2) and will update the set of assigned centers $f_C = f_C \cup f_{c_i}$, where $f_{c_i} = f_{v_2}$ (line 16 of Algorithm 2). Finally, it keeps iterating until solution C is completed or the condition $V \setminus (dom(f_C) \cup C) = \emptyset$ is met. At last, if there are some unassigned vertices $v \in V \setminus (dom(f_C) \cup C)$, each one of them is assigned to its nearest center $c \in C$ with enough capacity (lines 24 to 27 of Algorithm 2).

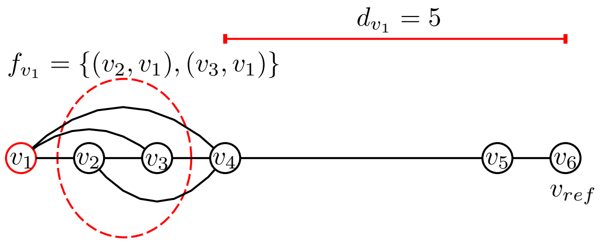

Figure 4: Vertices in $\{v_2, v_3\}$ are assigned to v_1.

$$\begin{array}{ccccccccc} d(v_{ref},v_6) & \leq & d(v_{ref},v_5) & \leq & d(v_{ref},v_4) & \leq & d(v_{ref},v_3) & \leq & d(v_{ref},v_2) \\ 0 & \leq & 1 & \leq & \underbrace{5}_{d_{v_1}=5} & \leq & \multicolumn{3}{c}{\underbrace{6 \qquad \leq \qquad 7}_{F_L=\{v_2,v_3\}}} \end{array} \tag{10}$$

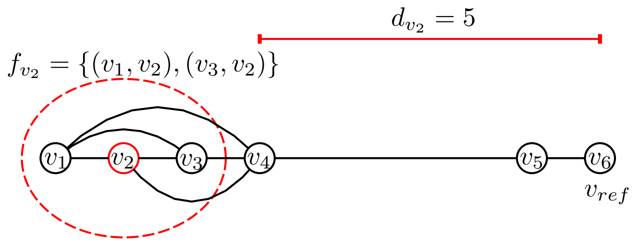

Figure 5: Vertices in $\{v_1, v_3\}$ are assigned to v_2.

$$\begin{array}{ccccccccc} d(v_{ref},v_6) & \leq & d(v_{ref},v_5) & \leq & d(v_{ref},v_4) & \leq & d(v_{ref},v_3) & \leq & d(v_{ref},v_1) \\ 0 & \leq & 1 & \leq & \underbrace{5}_{d_{v_2}=5} & \leq & \multicolumn{3}{c}{\underbrace{6 \qquad \leq \qquad 8}_{F_L=\{v_1,v_3\}}} \end{array} \tag{11}$$

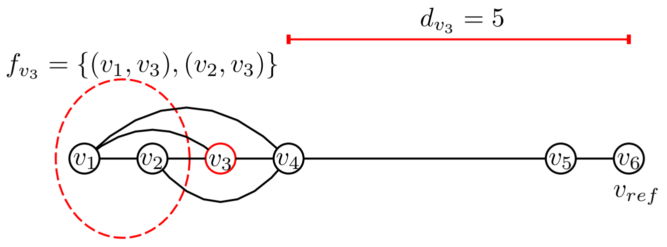

Figure 6: Vertices in $\{v_1, v_2\}$ are assigned to v_3.

$$\begin{array}{ccccccccc} d(v_{ref},v_6) & \leq & d(v_{ref},v_5) & \leq & d(v_{ref},v_4) & \leq & d(v_{ref},v_2) & \leq & d(v_{ref},v_1) \\ 0 & \leq & 1 & \leq & \underbrace{5}_{d_{v_3}=5} & \leq & \multicolumn{3}{c}{\underbrace{7 \qquad \leq \qquad 8}_{F_L=\{v_1,v_2\}}} \end{array} \tag{12}$$

$$\begin{array}{ccccccccc} d(v_{ref},v_6) & \leq & d(v_{ref},v_5) & \leq & d(v_{ref},v_3) & \leq & d(v_{ref},v_2) & \leq & d(v_{ref},v_1) \\ 0 & \leq & 1 & \leq & \underbrace{6}_{d_{v_4}=6} & \leq & \multicolumn{3}{c}{\underbrace{7 \qquad \leq \qquad 8}_{F_L=\{v_1,v_2\}}} \end{array} \tag{13}$$

3.1 Complexity analysis

This section presents the complexity analysis of the proposed OHCKC heuristic (Algorithm 1), which consists of a binary search that calls *GetFeasibleSolution* (Algorithm 2) and *DistanceBasedSelection* (Algorithm 3).

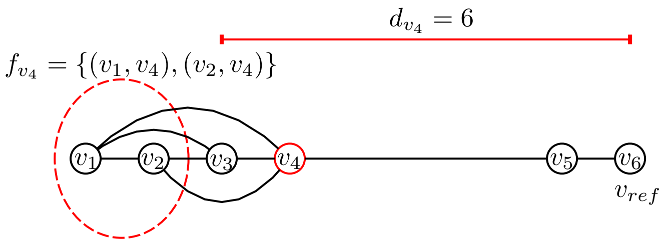

Figure 7: Vertices in $\{v_1, v_2\}$ are assigned to v_4.

3.1.1 *DistanceBasedSelection*

DistanceBasedSelection (Algorithm 3) begins by computing set $\{v \in V : v \in N_{G_r}[v_f] \ \wedge v \notin dom(f_C) \cup C \ \wedge \ score(v) > L\}$ in $O(n)$ steps by exploring the whole set V, where $|V| = n$ (line 4). The next loop (lines 4 to 12) is executed one time for each element of $\{v \in V : v \in N_{G_r}[v_f] \ \wedge v \notin dom(f_C) \cup C \ \wedge \ score(v) > L\}$, which has a cardinality of at most n. By keeping track of the distance from every vertex $v \in V$ to set C, selecting vertex v_{ref} takes $O(n)$ steps (line 5). Then, line 6 requires computing a set $F_L \subseteq N_{G_r}(v) \setminus (dom(f_C) \cup C)$ containing the L farthest vertices from $N_{G_r}(v) \setminus (dom(f_C) \cup C)$ to v_{ref}. We can get F_L in at most $O(n)$ steps if we compute the reference matrix of Table 2. With this matrix, we can efficiently find the farthest vertices of any given vertex. Otherwise, every time we need to get set F_L, we would first need to get the set $N_{G_r}(v) \setminus (dom(f_C) \cup C)$ in $O(n)$ steps, and after that, we would need to sort its elements according to their distances from v_{ref}. Since the best sorting algorithms run in $O(n \log n)$ steps, we would need to perform as many computations to obtain F_L. So, it is preferable to execute line 6 by using the reference matrix of Table 2. Although computing this reference matrix requires $O(n^2 \log n)$ steps, it is computed only once at the beginning of the algorithm. Next, the distance d_v from the $(L+1)^{th}$ farthest vertex $v \in N_{G_r}(v) \setminus (dom(f_C) \cup C)$ to v_{ref} can be obtained from line 6. Thus, line 7 does not increase the algorithm's complexity. Finally, vertex v and its corresponding assignment $f_{c'}$ with minimum d_v are returned (lines 8 to 11 and line 13). To sum up, the complexity of the whole *DistanceBasedSelection* heuristic is $O(n)$ times $O(n)$, namely $O(n^2)$.

3.1.2 *GetFeasibleSolution*

GetFeasibleSolution (Algorithm 2) begins by creating a bottleneck graph from the input graph G (line 4). This takes $O(n^2)$ steps due to checking every edge in the graph. Then, we have to compute the score of each vertex in V (lines 5 to 7). This computation takes $O(n^2)$ steps because the neighborhood of each vertex is explored. Then, the loop from lines 8 to 23 is repeated at most k times. At each iteration, the following actions take place. By keeping track of the distance from the vertices to C, getting an unassigned vertex v_f of maximal distance to C takes $O(n)$ steps. To obtain the set $(N_{G_r}[v_f]) \setminus (dom(f_C) \cup C)$ takes $O(n)$ steps too. If the condition of line 10 is met, the $O(n^2)$ *DistanceBaseSelection* algorithm (Algorithm 3) is executed. Otherwise, lines 13 and 14 are executed in $O(n)$ steps. So, the complexity from lines 10 to 15 is $O(n^2)$. Then, from line 17 to 21 the score of each vertex is updated. To do so, the score of the neighbors of every vertex in $dom(f_{c_i}) \cup \{c_i\}$ is reduced in one unit. Therefore, the complexity of the score update at each iteration is $O(n \cdot |dom(f_{c_i}) \cup \{c_i\}|)$. Recall that once a vertex is assigned to some center c_i, it cannot be reassigned to another center c_j $(i \neq j)$. Thus, the collection of f_{c_i} sets define a partition over f_C, where $i \in \{1, 2, ..., k\}$. So, $n \cdot |f_{C_1} \cup \{c_1\}| + n \cdot |f_{C_2} \cup \{c_2\}| + \cdots + n \cdot |f_{C_k} \cup \{c_k\}| = n(|f_{C_1} \cup \{c_1\}| + |f_{C_2} \cup \{c_2\}| + \cdots + |f_{C_k} \cup \{c_k\}|) = n^2$. Therefore, the complexity of the score updating is $O(n^2)$. Finally, lines 24 to 27 check for unassigned vertices and assign each one of them to its nearest center with enough capacity. The complexity of this last assignment is $O(kn)$ because each vertex $v \in V$ has to compare its distance to every center $c \in C$ with enough capacity. In conclusion, the overall complexity of Algorithm 2 is $O(k)$ times $O(n^2)$, which is $O(kn^2)$.

3.1.3 Overall complexity of OHCKC

The OHCKC heuristic (Algorithm 1) performs a binary search over the non-decreasing ordered list of edge weights ($|E| = m$), which takes $O(\log n)$ steps. The *GetFeasibleSolution* procedure is executed at each iteration of the binary search (Algorithm 2). So, the overall complexity of the OHCKC heuristic is $O(\log n)$ times the complexity of *GetFeasibleSolution*, which is $O(kn^2)$. Thus, the complexity of OHCKC is $O(kn^2 \log n)$. Finally, the last parts (lines 20 to 30) are optional. Lines 20 to 22 aim to compute a solution with exactly k centers. Lines 23 to 30 aim to perform an adjustment over each formed cluster by solving the 1-center problem. The complexity of these last operations is lower than $O(kn^2 \log n)$, so they do not affect the overall complexity of the proposed OHCKC heuristic. Now, notice that the capacity constraints can be ignored for $L \geq n$, and the problem becomes the uncapacitated vertex k-center problem. Under this scenario, OHCKC will never execute the *GetFeasibleSolution* heuristic because there will not be vertices with score greater than L (line 13 of Algorithm 2). Thus, OHCKC will behave exactly as the CDSh heuristic, which has a complexity of $O(n^2 \log n)$. In this sense, the OHCKC heuristic generalizes the CDSh heuristic.

3.2 Integrating parallel computing

One of the advantages of our proposal is that its runtime performance can be improved by using parallel computing. Specifically by using the SIMD model [15]. This model is particularly applicable to tasks where it is possible to perform the same computations over multiple input data simultaneously. Fortunately, our proposal has some segments with this characteristic: the *DistanceBasedSelection* procedure (Algorithm 3) and the computation of vertices' scores (lines 5-7 and 17-21 of Algorithm 2). As described before, the *DistanceBasedSelection* receives as input a farthest vertex v_f. Then, its closed neighborhood $N_{G_r}[v_f]$ is explored in order to get a pair $(c', f_{c'})$ with minimum d_v. Since the inspection of each element $v \in N_{G_r}[v_f]$ is independent of the others, this procedure can be parallelized. Thus, multiple processing units can work together to explore the closed neighborhood $N_{G_r}[v_f]$ in order to find the pair $(c', f_{c'})$ more quickly. Taking Figure 3 as an example, we have two processing units, pu_1 and pu_2. The set $N_{G_r}[v_f]$ can be partitioned so that pu_1 explores $\{v_1, v_2\}$ and pu_2 explores $\{v_3, v_4\}$. This way, pu_2 obtains the pair (c_3, f_{v_3}) while pu_1 obtains the pair (c_2, f_{v_2}). From these, the one with the minimum d_v is selected. Note that pu_1 could have obtained pair (c_1, f_{v_1}) as well because $d_{v_1} = d_{v_2}$.

For computing vertices' scores (lines 5-7 and 17-21 of Algorithm 2) the approach is similar. First, the set V is partitioned by the number of processing units (lines 5-7 of Algorithm 2). Then, each processing unit simultaneously computes the score of the vertices in a partition element. Regarding the score update (lines 17-21 of Algorithm 2), the parallel computing strategy works as follows. Given a center and its assigned vertices f_{c_i}, for each $v \in dom(f_{c_i}) \cup \{c_i\}$ the open neighborhood $N_{G_r}(v)$ is partitioned by the number of processing units. Then, the elements of the partition are simultaneously processed.

4 Empirical analysis

This section presents an empirical analysis of the OHCKC heuristic (Algorithm 1), and the OHCKC+ heuristic, where the OHCKC+ heuristic consists in executing the OHCKC heuristic n times, starting with a different center each time. All the executions of our proposal performs the optional parts of Algorithm 1. To test the algorithms we used three different datasets: A, B, and C. The purpose of each dataset is different. So, we separate each experimentation on different subsections. In all datasets, we used values of $k \in \{5, 10, 20, 40\}$. The values of L were selected as the smallest integer that ensures that all vertices can be assigned to the set of centers, including a 5% and 10% variation, i.e., $L \in \{\lceil (n-k)/k \rceil, \lceil \lceil (n-k)/k \rceil \cdot 1.05 \rceil, \lceil \lceil (n-k)/k \rceil \cdot 1.1 \rceil\}$. The main reason for selecting these values is that as the capacity increases, the problem comes closer to the uncapacitated version, which is relatively easier.

The goal of the experimentation over dataset A is to evaluate how the proposed OHCKC heuristic compares against a similar constructive heuristic from the literature. In this experimentation, only the fitness of the generated solutions is compared. The goal of the experimentation over dataset B is to evaluate the effect of the parallelization strategy. Namely, we compute the speedup of the implemented algorithm using a supercomputer with 32 physical cores. Finally, the goal of the experimentation over dataset C is to evaluate both fitness and running time of the OHCKC heuristic.

4.1 Constructive heuristic comparison

Dataset A consists of random instances generated with a uniform random distribution (prefix URDI) from a two-dimensional Euclidean plane with $n \in \{100, 150, 200, 250\}$. For each size, 30 instances were generated. We selected these values because the instances had to be solved optimally, and bigger instances could not be solved in a reasonable amount of time.

The goal of the experiments carried out over dataset A is to show that the solutions generated by OHCKC are competitive with regard to the solutions generated by a similar heuristic from the literature. The selected heuristic corresponds to the constructive phase of a Variable Neighborhood Descent metaheuristic for the capacitated vertex k-center problem [29]. It is important to emphasize that we only used the constructive phase to keep a fair comparison. Otherwise, we would be comparing the deterministic polynomial-time OHCKC heuristic against a stochastic non-polynomial-time metaheuristic with exploration and exploitation components.

Although the constructive phase mentioned above is a heuristic based on the p-dispersion problem, it generates feasible solutions for the uniform capacitated vertex k-center problem. We refer to this heuristic as PDISP [14, 29]. This heuristic has a random initialization, and according to our experimentation it is very sensitive to randomness. So, we removed its randomness, resulting in a $O(n^2 + L^3(n-k))$ deterministic heuristic with an empirically improved performance. One difference between the PDISP and OHCKC heuristics is that the assignment of the first is defined as $f_C : V \to C$, while it is defined as $f_C : V \setminus C \to C$ in the latter. In other words, OHCKC does not assign centers to other centers, while PDISP does. However, since the PDISP heuristic always assigns each center to itself, we can perform a fair comparison if the PDISP heuristic is executed with capacity values $L + 1$. Table 3 shows the results obtained by OHCKC and PDISP over dataset A. Column μ corresponds to the average fitness obtained from the 30 instances for each heuristic. The column with the check symbol ✓corresponds to the number of instances in which each heuristic obtained a better solution. The column GAP corresponds to the average GAP $= (\frac{\mu}{OPT} - 1) \cdot 100\%$ over the 30 instances, where OPT is the size of the optimal solution. To compute the optimal solutions we used the exact algorithm reported in Cornejo Acosta et al. [9]. We also performed the Mann–Whitney U nonparametric statistical test (a.k.a. Wilcoxon rank-sum test). We used the implementation of the Statistics package of the Apache Commons Math library `https://commons.apache.org/proper/commons-math/userguide/stat.html`. The tested null hypothesis H_0 is that there is no difference in the performance of the heuristics. The alternative hypothesis H_1 is that such difference exists. The column p corresponds to the p-value of the test and column α corresponds to significance levels with the corresponding test results. A value of 1 means a rejection of H_0. Table 3 shows that the OHCKC heuristic achieved a lower GAP in most cases. In addition, the OHCKC heuristic was able to find better solutions for most instances. There are some particular configurations where the number of checked instances for both heuristics is very similar, such as URDI-150 with $k = 10$ and $L = 16$, and URDI-250 with $k = 5$ and $L = 54$. Besides, the Wilcoxon rank-sum test rejects H_0 in most cases. Thus, for such cases, there is enough statistical evidence to conclude that the OHCKC heuristic outperforms the PDISP heuristic.

Regarding running time, the PDISP heuristic has complexity $O(n^2 + L^3(n-k))$ and the OHCKC heuristic $O(kn^2 \log n)$. This implies that PDISP tends to be faster for small values of L and large values of k, while OHCKC is faster for small values of k.

4.2 Parallelization analysis

Dataset B consists of relatively large instances with 1000, 2000, 3000, 4000, and 5000 vertices (one instance for each size). This experimentation aims to quantify the benefit of using the SIMD parallel computing approach into the OHCKC heuristic. Table 4 shows a comparison between sequential OHCKC and parallelized OHCKC, where the parallel version was executed with 2, 4, 8, 16, and 32 cores. For each configuration, column BKS (best-known solution) shows the best result found by OHCKC after 30 executions. Column S shows the average speedup, S$= \frac{T_{seq}}{T_{par}}$, where T_{seq} is the time taken by the sequential execution, whereas T_{par} is the time taken by the parallel execution. Figures 8 to 12 show how the speedup increases as the number of cores increase. Figures 13 to 17 show how the running time decreases as the number of cores increase. All these figures were generated with $L = \lceil (n-k)/k \rceil$ for each value of k.

Table 4 shows that using parallel computing with 32 cores allows us to speed up the running time on average up to 21.68 times. Besides, we observe the general trend that the larger the n value is, the bigger the speedup is. In some atypical cases, low speedup values were obtained. This includes instance URDI-1000 with $k = 40$ and $L = 26, 27$. However, in most cases, the speedup achieved is considerably good. Figures 8 to 12 show that the usage of parallel computing for this proposal has a good scaling, especially for the largest instances. In addition, these figures show that the value of k does not influence so much the speedup behavior when using parallel computing for instances with 2000 to 5000 vertices. This is good, as it implies that different values of k do not degrade the benefit of using parallel computing.

Table 3: Performance of the OHCKC heuristic over dataset A

instance	n	k	L	OHCKC			PDISP			Wilcoxon	
				μ	✓	GAP	μ	✓	GAP	p	$\alpha = 0.05, 0.01$
URDI	100	5	19	2202.37	28	38%	3036.80	2	91%	0	1, 1
			20	2213.23	20	42%	2594.20	10	67%	9E-03	1, 1
			21	2121.27	20	38%	2318.80	10	51%	1E-01	0, 0
		10	9	1979.13	30	67%	3232.33	0	170%	0	1, 1
			10	1592.63	25	52%	2357.03	5	124%	0	1, 1
		20	4	2098.50	25	138%	3248.67	5	267%	0	1, 1
			5	1012.20	28	44%	1593.40	2	126%	0	1, 1
		40	2	664.60	30	31%	1429.70	0	182%	0	1, 1
			3	488.27	30	9%	806.90	0	80%	0	1, 1
URDI	150	5	29	2182.90	30	40%	2845.33	0	83%	0	1, 1
			31	2140.87	22	40%	2524.70	8	66%	2E-04	1, 1
			32	2128.47	18	40%	2310.47	12	52%	6E-02	0, 0
		10	14	1919.50	29	68%	3096.83	1	170%	0	1, 1
			15	1852.07	22	77%	2276.43	8	117%	2E-03	1, 1
			16	1652.10	16	60%	1790.47	14	73%	7E-01	0, 0
		20	7	1340.73	28	79%	2254.27	2	200%	0	1, 1
			8	1047.47	27	48%	1460.90	3	106%	0	1, 1
		40	3	971.70	30	72%	2070.53	0	263%	0	1, 1
			4	623.73	30	28%	1125.97	0	131%	0	1, 1
URDI	200	5	39	2168.67	27	38%	3068.60	3	95%	0	1, 1
			41	2183.13	21	42%	2576.90	9	68%	2E-03	1, 1
			43	2153.20	15	42%	2326.40	15	53%	2E-01	0, 0
		10	19	2034.20	29	82%	3094.17	1	176%	0	1, 1
			20	1925.43	23	84%	2308.93	7	121%	3E-03	1, 1
			21	1672.73	19	62%	1911.47	11	85%	1E-01	0, 0
		20	9	2212.10	30	170%	3282.57	0	300%	0	1, 1
			10	1210.43	26	71%	1775.40	4	150%	0	1, 1
		40	4	2386.17	29	265%	3713.57	1	467%	0	1, 1
			5	730.73	30	49%	1273.73	0	160%	0	1, 1
URDI	250	5	49	2227.97	29	43%	3102.67	1	98%	0	1, 1
			52	2204.67	20	43%	2432.80	10	58%	2E-02	1, 0
			54	2152.33	16	40%	2242.13	14	46%	3E-01	0, 0
		10	24	1909.43	29	73%	3208.00	1	190%	0	1, 1
			26	1838.40	20	76%	2183.83	10	110%	1E-02	1, 0
			27	1741.10	17	68%	1889.87	13	82%	5E-01	0, 0
		20	12	1562.40	26	111%	2284.17	4	209%	4E-06	1, 1
			13	1210.77	26	70%	1719.17	4	141%	0	1, 1
			14	1091.37	29	55%	1519.17	0	116%	0	1, 1
		40	6	860.43	29	73%	1603.80	1	222%	0	1, 1
			7	692.30	30	45%	1013.13	0	112%	0	1, 1
Averages						65%			137%		

Table 4: Speedup results on dataset B using parallel computing

instance	n	k	L	BKS	2 cores	4 cores	8 cores	16 cores	32 cores
					S	S	S	S	S
URDI	1000	5	199	1951	1.91	3.71	6.91	11.92	17.83
			209	1879	1.96	3.76	7.07	12.88	17.76
			219	1796	1.91	3.73	7.06	11.91	17.46
		10	99	1783	1.91	3.64	6.73	11.52	16.50
			104	1560	1.91	3.64	6.86	11.34	16.76
			109	1528	1.93	3.66	6.86	11.12	15.19
		20	49	1445	1.89	3.70	6.45	9.79	13.84
			52	1216	1.88	3.57	6.11	9.05	11.89
			54	1122	1.93	3.54	6.02	9.25	11.13
		40	24	2279	1.94	3.63	6.50	10.81	15.16
			26	907	1.83	3.33	5.44	7.50	9.00
			27	881	1.84	3.38	4.75	7.41	8.35
URDI	2000	5	399	1960	1.97	3.85	7.34	13.51	21.69
			419	1943	1.97	3.85	7.31	13.51	20.88
			439	1875	1.97	3.84	7.42	13.38	21.07
		10	199	1546	1.96	3.81	7.20	13.09	20.31
			209	1540	1.95	3.82	7.20	12.96	20.98
			219	1530	1.94	3.76	7.27	12.67	20.01
		20	99	1488	1.95	3.78	7.17	13.09	20.26
			104	1211	1.95	3.77	7.14	12.84	20.48
			109	1215	1.93	3.70	6.88	11.97	18.94
		40	49	1733	1.94	3.76	7.12	12.84	19.97
			52	1017	1.90	3.64	6.51	11.24	16.16
			54	926	1.92	3.72	6.46	11.09	14.68
URDI	3000	5	599	2011	1.98	3.90	7.54	14.00	23.90
			629	1899	1.98	3.89	7.53	14.16	24.02
			659	1993	1.98	3.89	7.57	14.26	24.64
		10	299	1727	1.98	3.89	7.49	14.01	23.34
			314	2004	1.97	3.89	7.50	14.02	23.79
			329	1623	1.98	3.88	7.49	13.99	23.14
		20	149	2095	1.98	3.84	7.42	13.88	23.56
			157	1323	1.95	3.79	7.24	13.12	20.77
			164	1159	1.95	3.79	7.20	12.92	20.93
		40	74	1564	1.98	3.87	7.39	13.52	23.35
			78	1044	1.94	3.76	7.05	12.69	19.92
			82	888	1.94	3.71	7.05	12.22	19.06
URDI	4000	5	799	2138	1.97	3.91	7.66	14.52	25.73
			839	2022	1.99	3.93	7.68	14.66	25.85
			879	2102	1.99	3.93	7.66	14.63	25.83
		10	399	1641	1.99	3.91	7.56	14.26	24.86
			419	1646	1.99	3.91	7.58	14.26	24.73
			439	1568	1.98	3.91	7.55	14.31	23.77
		20	199	1338	1.98	3.87	7.51	14.15	23.95
			209	1306	1.96	3.84	7.43	13.98	23.48
			219	1200	1.96	3.85	7.45	13.55	21.61
		40	99	1400	1.97	3.89	7.52	14.15	24.64
			104	1095	1.95	3.82	7.29	13.32	22.02
			109	918	1.97	3.81	7.21	13.13	21.47
URDI	5000	5	999	2054	1.99	3.96	7.79	14.87	27.31
			1049	1971	1.99	3.92	7.72	14.85	26.38
			1099	1936	2.00	3.96	7.80	14.87	26.50
		10	499	1958	1.98	3.90	7.68	14.70	25.31
			524	1743	1.99	3.93	7.69	14.55	25.80
			549	1633	1.98	3.92	7.60	14.38	25.63
		20	249	2078	1.99	3.93	7.70	14.73	26.61
			262	1305	1.98	3.88	7.52	14.03	24.78
			274	1101	1.97	3.90	7.52	14.05	24.27
		40	124	2197	1.98	3.90	7.63	14.46	26.12
			131	1021	1.97	3.83	7.32	13.61	23.23
			137	928	1.96	3.86	7.39	13.61	21.57
Averages					1.95	3.80	7.20	13.02	21.14

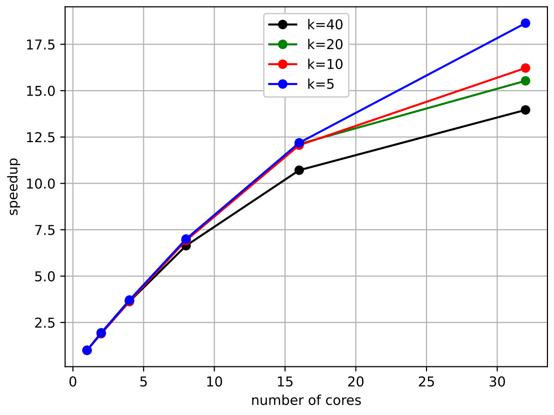

Figure 8: Speedup URDI-1000

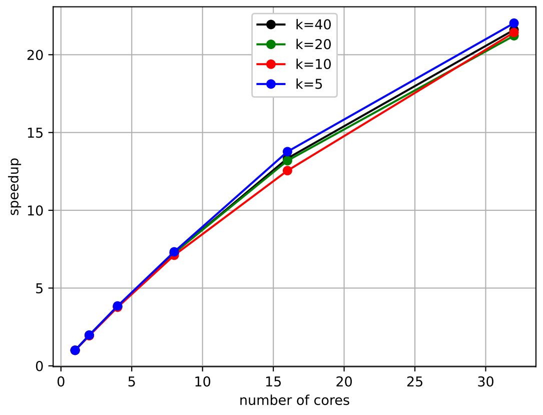

Figure 9: Speedup URDI-2000

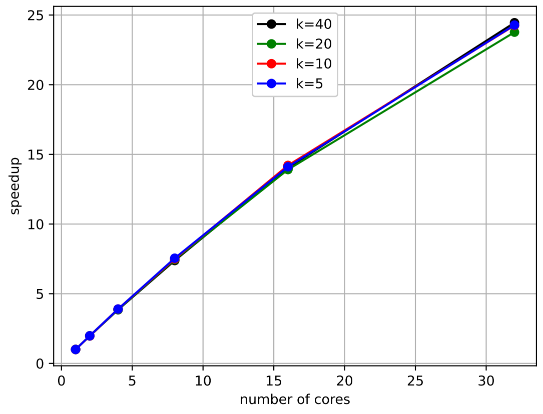

Figure 10: Speedup URDI-3000

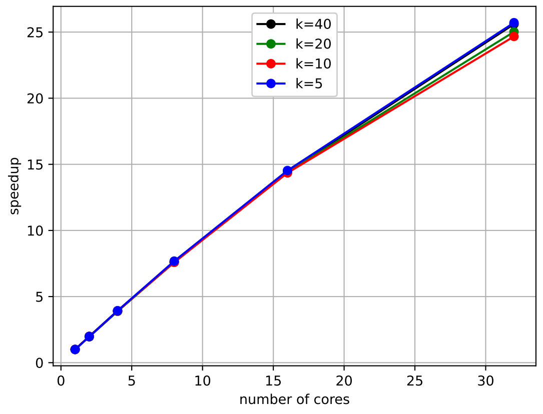

Figure 11: Speedup URDI-4000

Figures 13 to 17 show the following. First, the larger n and k values are, the greater the running time is when using only one core. Secondly, this behavior changes when the number of cores increases. Namely, the running time is almost the same in most instances when using 32 cores, independently of the value of k.

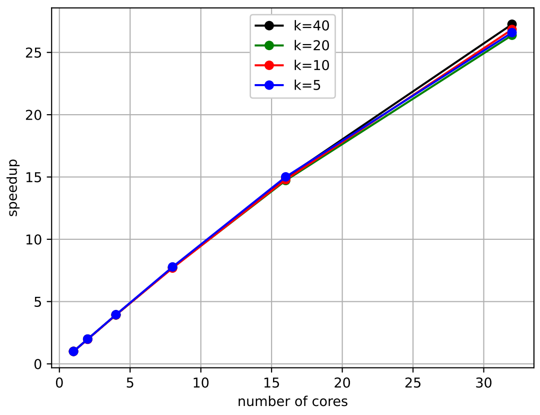

Figure 12: Speedup URDI-5000

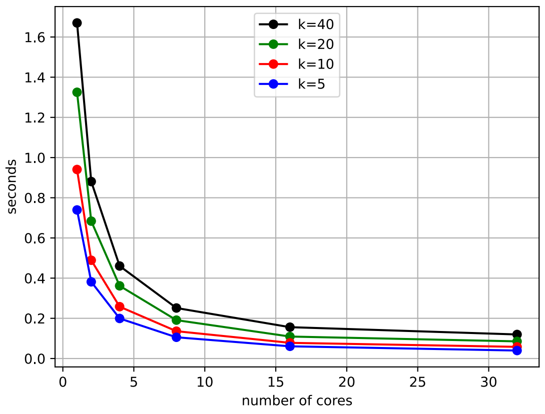

Figure 13: Times URDI-1000

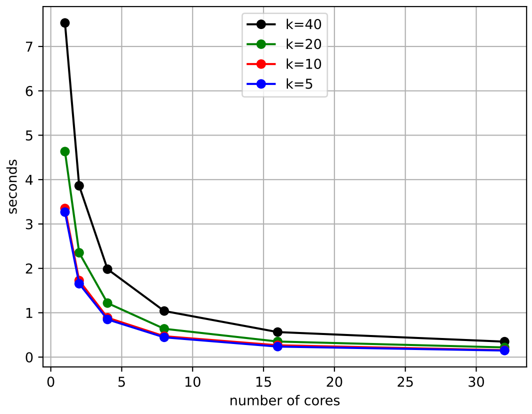

Figure 14: Times URDI-2000

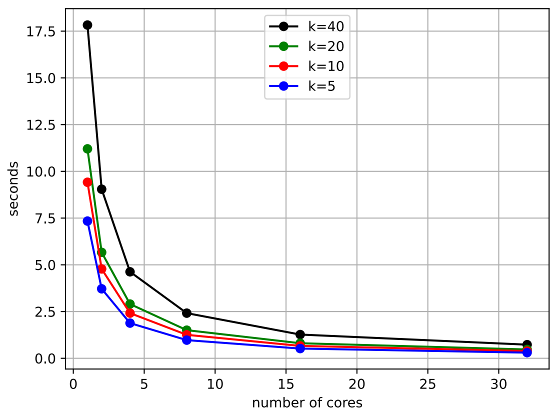

Figure 15: Times URDI-3000

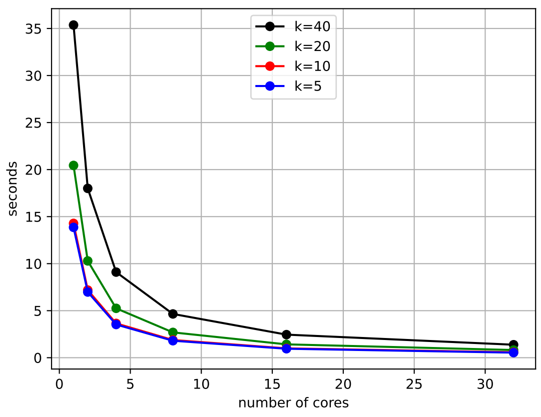

Figure 16: Times URDI-4000

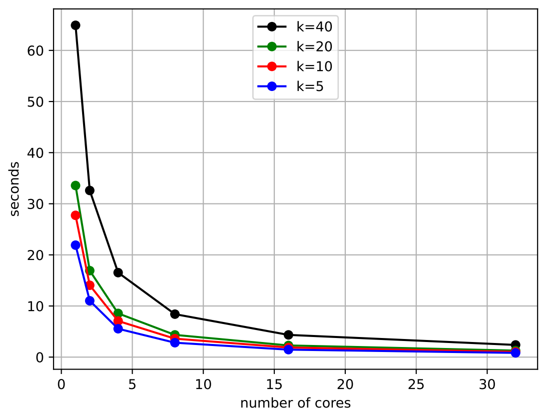

Figure 17: Times URDI-5000

4.3 Fitness evaluation on benchmark dataset

Dataset C consists of instances from the TSPLib library, divided into datasets C1 and C2 [31]. The order of the graphs in both datasets ranges from 195 to 280 and all of them have edge weights that follow a Euclidean metric. Table 5 shows the result of applying the OHCKC and OHCKC+ algorithms over C1. The OPT column shows the size of the optimal solution for each instance, which was computed with an exact algorithm from the literature [9]. Although OHCKC is a deterministic heuristic, it breaks ties at random. Thus, this heuristic was executed 30 times with a different seed. Columns under OHCKC show the average solution size μ, the standard deviation σ of the solution size, the average running time $t(s)$, the average speedup S, and the average GAP $= (\frac{\mu}{OPT} - 1) \cdot 100\%$. While OHCKC may generate different solutions at different executions, OHCKC+ tends to always generate the same solution. For this reason, it was executed only once. Columns under OHCKC+ show the size r of the found solution, the running time $t(s)$, the speedup S, and the GAP.

Table 5: Performance of the OHCKC and OHCKC+ heuristics over dataset C1

instance	n	k	L	OPT	OHCKC (30 runnings) 8 cores					OHCKC+ 8 cores			
					μ	σ	t(s)	S	GAP	r	t(s)	S	GAP
rat195	195	5	38	64	101.47	6.64	0.0047	3.36	59%	83	0.8257	3.20	30%
			40	64	98.87	6.57	0.0047	3.21	54%	84	0.8057	3.22	31%
			42	64	95.70	13.78	0.0050	3.00	50%	78	0.7887	3.11	22%
		10	19	41	81.77	9.91	0.0063	3.16	99%	55	1.2140	2.41	34%
			20	41	80.63	10.72	0.0070	2.62	97%	52	1.1750	2.31	27%
			21	41	70.07	9.55	0.0070	2.38	71%	49	1.1930	2.25	20%
		20	9	26	91.57	16.71	0.0147	1.98	252%	44	2.0940	1.69	69%
			10	28	47.87	4.52	0.0137	1.37	71%	40	1.9157	1.71	43%
		40	4	21	56.13	9.26	0.0197	1.98	167%	35	3.8280	1.24	67%
			5	20	27.00	1.84	0.0203	0.98	35%	22	2.9593	1.13	10%
d198	198	5	39	1527	1651.43	119.67	0.0033	5.30	8%	1527	0.9870	3.95	0%
			41	1407	1581.57	7.31	0.0050	3.73	12%	1407	0.9557	4.00	0%
			43	1139	1525.33	176.37	0.0053	3.50	34%	1139	0.9043	3.92	0%
		10	19	1452	1490.47	56.29	0.0103	3.48	3%	1452	1.7573	3.71	0%
			20	1407	1446.77	12.05	0.0100	3.37	3%	1441	1.7377	3.63	2%
			21	1139	1629.83	186.20	0.0080	2.88	43%	1139	1.5617	3.40	0%
		20	9	1380	1785.27	63.12	0.0200	2.82	29%	1452	3.0837	3.25	5%
			10	240	598.53	108.46	0.0117	3.31	149%	314	2.2597	2.14	31%
		40	4	1347	1713.27	133.19	0.0323	2.67	27%	1347	6.2220	3.03	0%
			5	113	148.93	8.54	0.0180	1.30	32%	127	3.2593	1.40	12%
gr202	202	5	40	49	53.60	2.58	0.0050	4.73	9%	49	1.0953	4.32	0%
			42	49	54.00	2.27	0.0063	3.95	10%	49	1.0867	4.39	0%
			44	49	54.00	2.27	0.0060	3.83	10%	49	1.1123	4.27	0%
		10	20	49	49.00	0.00	0.0093	4.00	0%	49	1.9440	3.89	0%
			21	37	44.77	5.64	0.0083	4.44	21%	37	1.8270	3.90	0%
			22	21	21.40	0.86	0.0073	3.23	2%	21	1.5587	3.09	0%
		20	10	21	21.00	0.00	0.0167	2.52	0%	21	2.8993	2.77	0%
			11	15	17.37	1.38	0.0133	2.75	16%	15	2.8107	2.96	0%
		40	5	6	6.70	0.47	0.0210	1.90	12%	6	3.9333	1.76	0%
			6	4	5.13	0.35	0.0177	1.70	28%	5	3.6897	1.51	25%
tsp225	225	5	44	121	173.47	15.70	0.0053	4.50	43%	126	1.1477	3.34	4%
			47	114	174.77	16.23	0.0057	3.94	53%	127	1.0647	3.51	11%
			49	114	173.37	17.56	0.0060	4.11	52%	126	1.0627	3.48	11%
		10	22	77	124.43	20.75	0.0073	3.68	62%	92	1.6803	2.84	19%
			24	73	132.93	23.70	0.0077	3.30	82%	80	1.3553	2.98	10%
			25	73	116.13	8.69	0.0083	2.76	59%	80	1.3310	2.96	10%
		20	11	51	99.37	6.17	0.0127	2.50	95%	74	2.7373	2.22	45%
			12	46	80.77	6.30	0.0123	2.14	76%	60	2.4123	2.05	30%
			13	46	68.87	3.84	0.0107	2.06	50%	56	2.2480	2.08	22%
		40	5	30	72.73	4.20	0.0203	1.98	142%	48	4.2737	1.54	60%
			6	30	43.57	3.47	0.0180	1.22	45%	36	3.7777	1.45	20%
gr229	229	5	45	164	165.00	0.00	0.0067	4.75	1%	165	1.6283	4.40	1%
			48	159	159.07	0.37	0.0067	4.60	0%	159	1.6293	4.42	0%
			50	157	157.87	0.43	0.0070	4.43	1%	157	1.5287	4.47	0%
		10	22	157	157.00	0.00	0.0097	4.97	0%	157	2.5480	4.22	0%
			24	38	47.27	6.11	0.0090	3.67	24%	38	1.8407	3.25	0%
			25	38	45.73	4.88	0.0093	3.43	20%	38	1.7613	3.30	0%
		20	11	38	47.13	8.46	0.0167	3.08	24%	38	3.3390	2.81	0%
			12	27	33.70	2.59	0.0147	2.43	25%	28	2.7947	2.29	4%
			13	23	28.57	1.43	0.0123	2.27	24%	27	2.6963	2.25	17%
		40	5	22	30.70	4.02	0.0257	2.12	40%	24	5.3320	1.97	9%
			6	18	20.20	0.41	0.0233	1.61	12%	20	4.6750	1.87	11%
Averages							0.0114	3.06	45%		2.1990	2.91	14%

Table 6: Performance of the OHCKC and OHCKC+ heuristics over dataset C2

instance	n	k	L	OPT	OHCKC (30 runnings) 8 cores					OHCKC+ 8 cores			
					μ	σ	t(s)	S	GAP %	r	t(s)	S	GAP %
kroA200	200	5	39	911	1362.93	147.44	0.0057	3.94	50%	1094	0.8650	3.95	20%
			41	911	1410.37	181.61	0.0060	3.61	55%	992	0.8767	3.85	9%
			43	911	1432.57	185.53	0.0053	3.94	57%	960	0.8783	3.79	5%
		10	19	646	1230.67	189.26	0.0083	3.24	91%	946	1.5470	2.98	46%
			20	609	1293.97	200.47	0.0083	3.44	112%	712	1.2323	2.57	17%
			21	600	1016.83	156.28	0.0080	2.58	69%	707	1.2980	2.61	18%
		20	9	499	1654.37	119.19	0.0150	2.98	232%	776	2.3797	2.07	56%
			10	407	731.67	74.00	0.0110	2.06	80%	562	2.0607	1.80	38%
		40	4	365	1506.00	295.76	0.0237	2.80	313%	712	4.3627	1.70	95%
			5	283	417.63	21.50	0.0180	1.26	48%	336	3.5550	1.21	19%
kroB200	200	5	39	918	1433.70	165.79	0.0060	3.78	56%	1090	0.9097	3.94	19%
			41	898	1382.27	126.72	0.0050	5.13	54%	1068	0.8827	3.86	19%
			43	898	1377.03	105.30	0.0067	3.85	53%	1059	0.8593	3.87	18%
		10	19	619	1282.87	175.40	0.0097	3.03	107%	895	1.3680	2.82	45%
			20	595	1208.10	274.83	0.0087	3.08	103%	781	1.2890	2.70	31%
			21	589	1116.47	244.12	0.0073	3.45	90%	705	1.2957	2.60	20%
		20	9	445	1452.07	183.93	0.0153	2.61	226%	690	2.2860	2.03	55%
			10	395	733.17	88.71	0.0113	2.15	86%	518	2.0430	1.76	31%
		40	4	316	1699.73	231.05	0.0290	2.59	438%	629	4.2530	1.50	99%
			5	272	411.20	28.91	0.0167	1.44	51%	344	3.3087	1.26	26%
ts225	225	5	44	4123	6708.00	0.00	0.0063	4.37	63%	6708	1.4507	4.27	63%
			47	3905	6282.03	111.38	0.0050	4.00	61%	6265	1.1413	4.19	60%
			49	3905	6033.33	611.01	0.0050	3.93	55%	4743	1.0377	3.70	21%
		10	22	3041	5779.87	644.65	0.0077	4.22	90%	4000	1.5953	2.86	32%
			24	3041	4601.70	348.05	0.0087	2.46	51%	3905	1.5773	2.90	28%
			25	3041	4719.33	397.73	0.0073	3.05	55%	3606	1.5287	2.84	19%
		20	11	2000	3689.27	346.06	0.0130	2.13	84%	2693	2.5647	1.95	35%
			12	1803	3070.17	119.72	0.0130	1.74	70%	2550	2.4313	1.92	41%
			13	1803	2844.33	190.65	0.0113	1.76	58%	2121	1.8823	1.93	18%
		40	5	1118	2902.50	185.43	0.0203	1.97	160%	2121	4.3637	1.48	90%
			6	1118	1903.67	136.19	0.0187	1.41	70%	1581	3.8297	1.44	41%
pr226	226	5	45	4507	6240.07	474.43	0.0047	4.21	38%	5305	1.1143	3.73	18%
			48	4104	6487.23	620.73	0.0057	3.53	58%	4952	1.0553	3.70	21%
			50	4104	6321.90	697.59	0.0053	4.00	54%	4924	1.0547	3.68	20%
		10	22	3239	6497.03	1350.90	0.0100	3.03	101%	4243	1.6953	2.97	31%
			24	2864	5048.63	450.74	0.0087	2.77	76%	3650	1.5020	2.98	27%
			25	2864	4602.63	504.11	0.0060	3.44	61%	3640	1.5123	2.92	27%
		20	11	2463	4195.70	660.55	0.0127	2.42	70%	2879	2.5287	2.14	17%
			12	2455	3382.37	244.36	0.0120	2.03	38%	2754	2.3750	2.06	12%
			13	2451	3018.50	169.97	0.0093	2.50	23%	2600	2.1870	2.04	6%
		40	5	1707	2735.97	315.07	0.0207	1.81	60%	1950	4.6817	1.65	14%
			6	1166	1654.10	94.16	0.0177	1.57	42%	1379	3.9213	1.56	18%
a280	280	5	55	71	118.03	11.02	0.0083	4.28	66%	86	1.9863	4.28	21%
			58	68	109.77	20.79	0.0070	4.76	61%	77	1.9130	3.99	13%
			61	68	92.77	12.54	0.0067	4.45	36%	72	1.6690	3.92	6%
		10	27	47	86.97	9.44	0.0123	3.46	85%	62	2.8223	3.38	32%
			29	46	90.50	17.19	0.0100	4.40	97%	61	2.7410	3.35	33%
			30	46	72.10	6.61	0.0107	3.28	57%	58	2.4770	3.26	26%
		20	13	34	114.07	11.41	0.0243	3.58	235%	54	4.2747	2.78	59%
			14	29	62.63	7.90	0.0173	2.83	116%	44	4.2453	2.58	52%
			15	29	49.83	2.91	0.0163	2.06	72%	39	3.8427	2.44	34%
		40	6	24	101.33	34.51	0.0363	3.10	322%	51	8.4193	2.27	113%
			7	20	36.07	3.67	0.0200	2.50	80%	27	6.4923	1.76	35%
Average							0.0118	3.06	95%		2.3672	2.75	33%

Table 5 shows that the running time of the OHCKC heuristic (which varies from 0.0033 to 0.032 seconds) is much better than the running time of the OHCKC+ heuristic (which varies from 0.7643 to 5.82 seconds). This was expected, since OHCKC+ consists in executing the OHCKC heuristic n times. It was also expected that the OHCKC+ heuristic generates better solutions than OHCKC, and with better precision. Hence, in general the GAP values of OHCKC+ are better. The results from Table 6 are consistent with those from Table 5. The solutions computed by OHCKC+ are of better quality while their running times are longer. Specifically, OHCKC tends to return solutions up to 48% (98%) worse than the optimum, while OHCKC+ tends to return solutions up to 14% (36%) worse than the optimum over C1 (C2). From column S of Tables 5 and 6, observe that the average speedup is lower than the average speedup reported on Table 4 when using 8 cores. This may be due that, for small instances, the number of tasks assigned to each core are not enough to efficiently take advantage of each core. In fact, this confirms the intuitive observation that the bigger the instance, the greater the speedup.

The OHCKC and OHCKC+ heuristics were implemented in C++. All the experiments were performed on an platform with Intel(R) Xeon Phi(TM) CPU 7250@1.40GHz (x64), 384 GB RAM, under an OS CentOS 7.3.1611 Kernel 3.10.0-514.6.1.el7.x86_64, x86_64 architecture with a GCC 4.8.5 compiler. All datasets and implementations can be consulted from `https://github.com/alex-cornejo/heuristic_ckc`.

5 Conclusions

The capacitated vertex k-center problem is an $\mathcal{NP}$-hard problem that has been approached through approximation [22, 3], heuristic [33], metaheuristic [29], and exact algorithms [26, 2, 23]. Regarding approximation algorithms, the best known deliver 6-approximated solutions, i.e., solutions of size at most 6 times the size of the optimal solution [22, 3]. Although the solutions generated by these approximation algorithms may be impractical, they are considered efficient, because they run in polynomial time. Regarding heuristic and metaheuristic algorithms, they rely mainly on local search procedures. That is, they construct initial solutions that are iteratively improved by the relocation of centers and reassignment of vertices. According to the empirical evidence, these methods are among the best for solving this problem [29]. However, they are computationally intensive and may use more resources when applied to large scale instances. With regard to the exact algorithms, they are mainly based on integer programming and mixed integer programming formulations of the problem, and they tend to perform well on instances with some hundreds of vertices [26, 2, 23]. However, efficient heuristics are needed for obtaining provisional solutions for relatively large instances of the problem. In this context, the heuristic proposed in this paper comes in handy because it can quickly find solutions of relatively good quality thanks to its parallel implementation.

The capacitated vertex k-center problem is related to the minimum capacitated dominating set problem. This paper introduces a parallel constructive heuristic OHCKC for the uniform capacitated vertex k-center problem based on this relationship. In brief, this heuristic selects the locally best centers to be part of a capacitated dominating set. Every time a center is selected, up to L vertices are assigned to it. This assignment is performed trying to minimize the distance from the unassigned vertices to the centers selected at the next iterations. This way, OHCKC tries to construct a minimum capacitated dominating set. By doing so over the bottleneck graph G_r of the original input graph, it obtains a feasible solution for the uniform capacitated vertex k-center problem, where r is a guess on the size of the optimal solution. According to the results from Tables 5 and 6, both OHCKC and OHCKC+ can get *good* provisional solutions in relatively short amounts of time. Besides, for instances with some thousands of vertices, the implemented parallel strategy is capable to improve the running time up to 25 times for some instances.

The main property that distinguishes the OHCKC heuristic from other heuristics from the literature is that once it selects a center or assigns a vertex, that decision never changes. So, by taking this into account, the performance of the OHCKC and OHCKC+ heuristics is remarkable. The complexity of the OHCKC heuristic is $O(kn^2 \log n)$. However, for $L \geq n$ it behaves exactly as the CDSh algorithm, which has a complexity of $O(n^2 \log n)$. In this sense, the OHCKC heuristic generalizes the CDSh heuristic, which is one of the best heuristics for the uncapacitated vertex k-center problem. Finally, in the future we would like to integrate the OHCKC heuristic into a metaheuristic framework.

Acknowledgments: The authors acknowledge Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE) and Consejo Nacional de Ciencia y Tecnología (CONACYT) for providing the necessary resources for the development of this research. The authors thankfully acknowledge the computer resources, technical expertise and support provided by the Laboratorio Nacional de Supercómputo del Sureste de México, CONACYT member of the network of national laboratories. The research of Roger Ríos-Mercado has been supported by UANL through its Scientific and Technological Research Support Program (grant PAICYT CE1837-21).

References

[1] A. Al-Khedhairi and S. Salhi. Enhancements to two exact algorithms for solving the vertex P-center problem. *Journal of Mathematical Modelling and Algorithms*, 4(2):129–147, 2005.

[2] M. Albareda-Sambola, J. A. Díaz, and E. Fernández. Lagrangean duals and exact solution to the capacitated p-center problem. *European Journal of Operational Research*, 201(1):71–81, 2010.

[3] H.-C. An, A. Bhaskara, C. Chekuri, S. Gupta, V. Madan, and O. Svensson. Centrality of trees for capacitated k-center. *Mathematical Programming*, 154(1-2):29–53, 2015.

[4] J. Barilan, G. Kortsarz, and D. Peleg. How to allocate network centers. *Journal of Algorithms*, 15(3): 385–415, 1993.

[5] H. Calik and B. C. Tansel. Double bound method for solving the p-center location problem. *Computers & Operations Research*, 40(12):2991–2999, 2013.

[6] H. Çalık, M. Labbé, and H. Yaman. p-Center problems. In G. Laporte, S. Nickel, and F. Saldanha da Gama, editors, *Location Science*, chapter 3, pages 51–65. Springer, Cham, Switzerland, 2nd edition, 2019.

[7] D. Chen and R. Chen. New relaxation-based algorithms for the optimal solution of the continuous and discrete p-center problems. *Computers & Operations Research*, 36(5):1646–1655, 2009.

[8] C. Contardo, M. Iori, and R. Kramer. A scalable exact algorithm for the vertex p-center problem. *Computers & Operations Research*, 103:211–220, 2019.

[9] J. A. Cornejo Acosta, J. García Díaz, R. Menchaca-Méndez, and R. Menchaca-Méndez. Solving the capacitated vertex k-center problem through the minimum capacitated dominating set problem. *Mathematics*, 8(9):1551, 2020.

[10] M. S. Daskin. A new approach to solving the vertex p-center problem to optimality: Algorithm and computational results. *Communications of the Operations Research Society of Japan*, 45(9):428–436, 2000.

[11] M. S. Daskin. *Network and Discrete Location: Models, Algorithms, and Applications.* Wiley, New York, 2011.

[12] M. E. Dyer and A. M. Frieze. A simple heuristic for the p-centre problem. *Operations Research Letters*, 3(6):285–288, 1985.

[13] S. Elloumi, M. Labbé, and Y. Pochet. A new formulation and resolution method for the p-center problem. *INFORMS Journal on Computing*, 16(1):84–94, 2004.

[14] E. Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46(1):48–60, 1990.

[15] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C–21(9):948–960, 1972.

[16] J. Garcia-Diaz, J. Sanchez-Hernandez, R. Menchaca-Mendez, and R. Menchaca-Mendez. When a worse approximation factor gives better performance: a 3-approximation algorithm for the vertex k-center problem. *Journal of Heuristics*, 23(5):349–366, 2017.

[17] J. Garcia-Diaz, R. Menchaca-Mendez, R. Menchaca-Mendez, S. P. Hernández, J. C. Pérez-Sansalvador, and N. Lakouari. Approximation algorithms for the vertex k-center problem: Survey and experimental evaluation. *IEEE Access*, 7:109228–109245, 2019.

[18] J. García Diaz, R. Menchaca Méndez, J. Sánchez Hernandez, and R. Menchaca Méndez. Local search algorithms for the vertex k-center problem. *IEEE Latin America Transactions*, 16(6):1765–1771, 2018.

[19] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[20] D. S. Hochbaum and D. B. Shmoys. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985.

[21] T. Ilhan, F. Ozsoy, and M. Pinar. An efficient exact algorithm for the vertex p-center problem and computational experiments for different set covering subproblems. Technical report, Bilkent University, Department of Industrial Engineering, Ankara, Turkey, 2002.

[22] S. Khuller and Y. J. Sussmann. The capacitated K-center problem. *SIAM Journal on Discrete Mathematics*, 13(3):403–418, 2000.

[23] R. Kramer, M. Iori, and T. Vidal. Mathematical models and search algorithms for the capacitated p-center problem. *INFORMS Journal on Computing*, 32(2):444–460, 2020.

[24] E. Minieka. The m-center problem. *SIAM Review*, 12(1):138–139, 1970.

[25] N. Mladenović, M. Labbé, and P. Hansen. Solving the p-center problem with tabu search and variable neighborhood search. *Networks*, 42(1):48–64, 2003.

[26] F. A. Özsoy and M. Ç.. Pınar. An exact algorithm for the capacitated vertex p-center problem. *Computers & Operations Research*, 33(5):1420–1436, 2006.

[27] J. Plesník. A heuristic for the p-center problems in graphs. *Discrete Applied Mathematics*, 17(3): 263–268, 1987.

[28] D. R. Quevedo-Orozco and R. Z. Ríos-Mercado. A new heuristic for the capacitated vertex p-center problem. In C. Bielza, A. Salmerón, A. Alonso-Betanzos, J. I. Hidalgo, L. Martínez, A. Troncoso, E. Corchado, and J. M. Corchado, editors, *Advances in Artificial Intelligence*, volume 8109 of *Lecture Notes in Artificial Intelligence*, pages 279–288. Springer, Heidelberg, Germany, 2013.

[29] D. R. Quevedo-Orozco and R. Z. Ríos-Mercado. Improving the quality of heuristic solutions for the capacitated vertex p-center problem through iterated greedy local search with variable neighborhood descent. *Computers & Operations Research*, 62:133–144, 2015.

[30] R. Rana and D. Garg. The analytical study of k-center problem solving techniques. *Int. J. Inf. Technol. Knowl. Manag*, 1(2):527–535, 2008.

[31] G. Reinelt. TSPLIB – A traveling salesman problem library. *ORSA Journal on Computing*, 3(4): 376–384, 1991.

[32] B. Robič and J. Mihelič. Solving the k-center problem efficiently with a dominating set algorithm. *Journal of Computing and Information Technology*, 13(3):225–234, 2005.

[33] M. P. Scaparra, S. Pallottino, and M. G. Scutellà. Large-scale local search heuristics for the capacitated vertex p-center problem. *Networks*, 43(4):241–255, 2004.

[34] D. B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. *Combinatorial Optimization*, 20:355–397, 1995.