An Incremental Evaluation Mechanism for the Critical Node Problem

ID: 236

Abstract

The Critical Node Problem (CNP) is to identify a subset of nodes in a graph whose removal maximally degrades pairwise connectivity. The CNP is an important variant of the Critical Node Detection Problem (CNDP) with wide applications. Due to its NP-hardness for general graphs, most works focus on local search algorithms that can return a good quality solution in a reasonable time. However, computing the objective function of CNP is a frequent procedure and is time-consuming (with complexity $O(|V| + |E|)$) during the search, which is a common problem that previous algorithms suffered from. In this paper, we propose a general incremental evaluation mechanism (IEM) to compute the objective function with much lower complexity. In this work, we improved two important greedy operations with IEM, along with experiments. Finally, we evaluate IEM by applying it into an evolutionary algorithm on two popular benchmarks, compared with the state-of-the-art approach. The experimental results showed the significance of IEM.

1 Introduction

Given an undirected graph $G = (V, E)$ and an integer K. The critical node detection problem (CNDP) is to identify a set of K nodes whose removal maximally degrades network connectivity according to some predefined connectivity metrics. An important variant of CNDP is critical node problem (CNP), in which the connectivity metric is defined as pairwise connectivity. Recently, the CNP has attracted much attention for its wildly real-world applications in a number of fields, *e.g.*, risk management [Arulselvan *et al.*, 2007], network vulnerability assessment [Shen *et al.*, 2013], biological molecule studies [Boginski and Commander, 2009; Tomaino *et al.*, 2012], and social network analysis [Fan and Pardalos, 2010; Leskovec *et al.*, 2007].

Di Summa *et al.*[2011] showed that the CNP can be solved in polynomial time with dynamic programming over trees. While for general graphs, the CNP is known to be NP-hard [Arulselvan *et al.*, 2009]. Currently, there are mainly two types of algorithms for CNP, *i.e.*, exact algorithms and local search algorithms. Exact algorithms solve the CNP by using Integer Linear Programming (ILP) [Arulselvan *et al.*, 2009; Di Summa *et al.*, 2012; Veremyev *et al.*, 2014a; Veremyev *et al.*, 2014b; Ventresca and Aleman, 2014a; Ventresca and Aleman, 2014c; Shen *et al.*, 2013]. These algorithms in ILP formulation can guarantee the optimality of their solutions, but the withdraw is that they will require exponential computation time in the general cases.

Consequently, many efforts have gone into the studies of local search algorithms that can return a good quality solution within a reasonable time. An early greedy algorithm was proposed by Arulselvan *et al.* [2009] and impoved later by many heuristic algorithms [Ventresca and Aleman, 2014b; Aringhieri *et al.*, 2015; Addis *et al.*, 2016; Aringhieri *et al.*, 2016b]. In [Aringhieri *et al.*, 2016b], the authors proposed a method based on a general Variable Neighborhood Search (VNS) framework and another one is based on an Iterated Local Search (ILS) framework. Moreover, two metaheuristic approaches, namely simulated annealing and population-based incremental learning methods have been explored [Ventresca, 2012] for large networks. Recently, two evolutional algorithms were proposed. The first is an efficient evolutionary framework for solving different variants of the CNDP, including the CNP [Aringhieri *et al.*, 2016a]. The second is an approach based on the *Memetic Algorithm* (MA), which achieves state-of-the-art performance [Zhou *et al.*, 2018]. For a detailed review of the CNP, we refer the reader to a comprehensive survey by Lalou *et al.* [2018].

Although considerable algorithms for CNP have been developed, they all suffer from the great computational complexity of calculating the objective function value, *i.e.*, the pairwise connectivity. A common drawback of all existing algorithms is that the objective function of a neighbor candidate solution have to be computed from scratch, resulting in a slow searching process, especially for the exploitation phase. Indeed, this drawback has also been pointed out in recent works [Aringhieri *et al.*, 2016b; Zhou *et al.*, 2018].

To overcome this problem, Aringhieri *et al.* [2016b] presented an improved neighborhood search algorithm by performing a modified *Connect* algorithm. As a result, they obtained two refined neighborhood without loosing the quality of neighbors, namely *Neighborhood* N_1 and N_2, which improve the efficiency of origin $Swap$ operation. While other algorithms resort to reduce the size of the neighborhood candidate solutions by sacrificing the quality of the

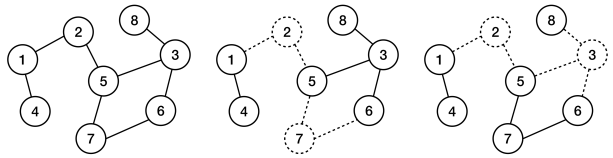
best neighbor during exploitation. For instance, Zhou and Hao [2017] breaks the traditional greedy $Swap$ operation into two distinct Add and $Remove$ operation. Another alternative method is to redefine the neighbors of a candidate solution by considering the problem feature, *e.g.*, the largest component in the residual graph [Zhou *et al.*, 2018]. Overall, no faster algorithm is found so far for the computation of the objective function.

In this paper, we propose the first incremental evaluation mechanism (IEM) for the CNP, which can speed up the computation of the objective function. The basic idea of IEM is to track the *component configuration* which is to maintain and utilize the size and index of each component and during the search. Based on the component configuration matained, the objective function of a candidate solution can be computed by means of obtained computed objective function value. The computation of the objective function value is necessary to evaluate a candidate solution, thus IEM can speed up the evaluation process for each iteration of all existing algorithms.

There are two important greedy operations in the local search algorithms for CNP. First is $Swap$ operation, which is widely used in previous algorithms [Arulselvan *et al.*, 2009; Aringhieri *et al.*, 2016b]. Aringhieri *et al.*[2016b] improved it to a more efficient $Swap$ operation, namely $SwapN$. Second is Add-$Remove$ operation, which can be regarded as a two-stage greedy operation, which is used in [Zhou and Hao, 2017; Zhou *et al.*, 2018]. In this work, we equip two operations with IEM to get two new operations, then we compare the computational complexity of proposed operations with previous works theoretically. Moreover, we carry out experiments to show the significant improvement of IEM on two greedy operations.

Indeed, a common ground of CNDP problems is that computing the objective function from scratch is costly, hence IEM can be easily generalized to other variants of CNDP. Finally, we implement a simple evolutionary algorithm and its improved version equipped with IEM to solve CNP[1]. By comparing our results with the state-of-the-art algorithm, we found out the effectiveness of IEM.

The paper is organized as follows. The next section introduce some necessary technical preliminaries. We introduce our main ideas IEM in Section 3. In Section 4, we present an application of IEM, *i.e.*, improving $Swap$ operation, along with related experiments. Experiments of IEM in comparison with the state-of-the-art algorithm are shown in Section 5. Finally, we make conclusions and outline future work.



(a) A graph (b) $S = \{2, 7\}$ (c) $S = \{2, 3\}$

Figure 1: A CNP instance with $K = 2$.

[1]Due to space limit, omitted proofs and supporting materials are provided in the additional files and online appendix (http://tinyurl.com/IJCAI19-236).

2 Preliminary

In this section, we begin with some basic definitions and notations. Then we review the greedy operations for CNP.

2.1 Definitions and Notations

In the following, we use $G = (V, E)$ to denote a graph where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of nodes and $E = \{e_1, e_2, \ldots, e_m\}$ is a set of edges. The *size* of a graph is defined as the number of nodes. In a graph G, each edge is a 2-element subset of V. For an edge $e = \{v, u\}$, we say that nodes u and v are the endpoints of edge, and u is adjacent to v. A graph is *connected* when there is a path between every pair of nodes. The *component* is defined to be a connected subgraph $C = (V_C, E_C)$ where $V_C \subseteq V$ and $E_C \subseteq E$. We will use C_i and δ_i to denote the i-th component and its size, respectively. The *neighborhood* of a node u is $N(u) = \{v \in V \mid \{u, v\} \in E\}$.

Given an undirected graph $G = (V, E)$ and an integer K. The CNP seeks to find a set $S \subseteq V$ of at most K nodes, the deletion of which minimizes pairwise connectivity in the remaining graph $G[V \setminus S]$.

$$f(S) = \sum_{C_i \in G[V \setminus S]} \frac{\delta_i \left(\delta_i - 1\right)}{2} \tag{1}$$

Figure 1 shows a CNP instance, where the undirected graph consists of seven nodes, seven edges, and $K = 2$. A *candidate solution* is a set $S \subseteq V$ of K nodes. If a candidate solution is $S = \{2, 7\}$, then the graph is broken into three components whose nodes are $\{1, 4\}$ and $\{3, 5, 6, 8\}$. The pairwise connectivity of each component is $\frac{2(2-1)}{2}$ and $\frac{4(4-1)}{2}$, so the residual objective value is $f(S) = 7$.

The *neighbor* of a candidate solution is also a candidate solution that differs in only one node. We say a neighbor S_1 is better than neighbor S_2 if $f(S_1) < f(S_2)$. Given a candidate solution S, the *decrement* (resp. *increment*) of a node $v \in V \setminus S$ (resp. $u \in S$) is defined as the decremental (incremental) pairwise connectivity after removing node v (resp. reintroducing node u), which is $g(v) = f(S \setminus \{v\}) - f(S)$ (resp. $h(u) = f(S) - f(S \cup \{u\})$).

2.2 Review of Greedy Operations

Because a candidate solution of K nodes is always better than a solution of $K - 1$ nodes, thus the common practice for exploitation is to begin with an initial solution of size K, then updates it to the best neighbor by greedy operations, *e.g.*, $Swap$ operation, Add-$Remove$ operation, *etc.* We review the $Swap$ and the Add-$Remove$ below.

$Swap$: Given a candidate solution S, a complete neighborhood evaluation requires to select all the nodes $u \in S$, with $|S| = K$ and pair them with all the nodes $v \in V \setminus S$. Then the size of whole neighbors is $K(|V| - K)$. The $Swap$ operation is to find the best neighbor from $K(|V| - K)$ neighbors. Currently, $f(S)$ must be computed through a $Connect$ algorithm computing the connected components of a graph [Hopcroft and Tarjan, 1973], which requires $O(|V| + |E|)$. Hence the complexity to find the best neighbor is $O(K(|V| - K)(|V| + |E|))$, which is very time-consuming.

Recently, Aringhieri *et al.* [2016b] proposed an improved algorithm to do $Swap$, named $SwapN$ (with *neighborhood* N_1, we do not mention N_2 because it is not as good as N_1). The computational complexity of $SwapN$ is $O(K(|V| + |E|))$, which is the best $Swap$ operation up to now.

Add-$Remove$: Given a candidate solution S of K nodes, this operation first expend it to S' of $K+1$ nodes by adding a node $v \in V \setminus S$ with maximum decrement, where the complexity is $O((|V| - K)(|V| + |E|))$. Then it removes a node $u \neq v$ from S' with minimum increment, thus the complexity is $O(K(|V| + |E|))$. Overall, the whole complexity for this method is $O(|V|(|V| + |E|))$.

3 Main Ideas

In this section, we present the Incremental Evaluation Mechanism to speed up the computation of the objective function for CNP. We first explain the details of IEM, then analysis the computational complexity.

3.1 Incremental Evaluation Mechanism

In this subsection, we present a new mechanism, named IEM, to compute objective function incrementally. The basic idea of IEM is to maintain and utilize the component configuration during the search. The increment after removing node u from S can be computed by means of component configuration, and the decrement after adding node v to S can be computed by only traversing several components that are related to u and v. In the following, we first state two key definitions and explain IEM.

Definition 3.1. Given a candidate solution S, a *move* is an update action applied on S, denoted as (u, v), where u is a node to be removed from S and v is the node to be added to S.

Definition 3.2. The current *component configuration* is defined as a tuple $I = (\Delta, pos)$, where $\Delta = \{\delta_1, \ldots, \delta_m\}$ is a set of variables indicating the size of i-th component and pos: $V \to Z^+$ is a function which maps a node n to the index of the component containing n.

The IEM is described as the following three steps. First two steps track component configuration I. Last step is to compute objective function value incrementally by means of I.

Initialization: Given an initial candidate solution S, IEM computes $f(S)$ and simultaneously obtains the initial component configuration I by traversing the whole graph with a depth-first search (DFS) process [Cormen *et al.*, 2009].

Update I: Given current component configuration I and a move (u, v), the component configuration I' after the move can be updated with Algorithm 1. In this algorithm, we initialize I' with I (line 1), generate a new index for the component to be merged (line 2), and assign its size with 1 (line 3). Then we update the component configuration after putting node u back to the residual graph (line 4-6), that is, increasing the size of the merged component (line 5), and remove the information of the size of neighborhood component (line 6), updating pos for each neighbors of u in the residual graph (line 7). Similarly, we update I after the component $C_{pos(v)}$ is split by deleting the node v (line 9-13).

Evaluation: Given current candidate solution S, $f(S)$, component configuration $I = (\Delta, pos)$, and a move (u, v).

Algorithm 1: the Update procedure

Input : Graph G, a move (u, v), the current component configuration $I = (\Delta, pos)$.
Output: new $I' = (\Delta', pos')$.

1 $I' \leftarrow I$;
2 Let $pos(u)$ be a new component index;
3 $\delta_{pos(u)} \leftarrow 1$;
4 **foreach** $n \in N(u)$ **and** $\delta_{pos(n)} \in \Delta$ **do**
5 $\quad \delta_{pos(u)} \leftarrow \delta_{pos(u)} + \delta_{pos(n)}$;
6 $\quad \Delta \leftarrow \Delta \setminus \{\delta_{pos(n)}\}$;
7 $pos(w) \leftarrow pos(u), \forall w \in C_{pos(n)}$;
8 $\Delta \leftarrow \Delta \cup \{\delta_{pos(u)}\}$;
9 **foreach** $n \in N(v)$ **and** $\delta_{pos(n)} = \delta_{pos(v)}$ **do**
10 $\quad (pos(n), \delta_{pos(n)}) \leftarrow \text{DFS}(n)$;
11 $\quad pos(w) \leftarrow pos(n), \forall w \in C_{pos(n)}$;
12 $\quad \Delta \leftarrow \Delta \cup \{\delta_{pos(n)}\}$;
13 $\Delta \leftarrow \Delta \setminus \{\delta_{pos(v)}\}$;
14 **return** new component configuration $I' = (\Delta', pos')$.

The increment $h(u)$ of u can be computed easily by means of I, which is illustrated in formula (2).

$$h(u) = \langle \sum_{\delta \in \Delta'} \delta + 1 \rangle - \sum_{\delta \in \Delta'} \langle \delta \rangle \tag{2}$$

Where $\langle \cdot \rangle$ denotes a mapping from the component size to corresponding pairwise connectivity, *i.e.*, $\langle X \rangle = \frac{X(X-1)}{2}$, and $\Delta' = \{\delta_{pos(n)} \mid n \in N(u)\}$. Take the graph in Figure 1(b) as an example. If we put node 2 back into the residual graph, the size of new component $\{1, 2, 4, 5\}$ is the sum of the size of $\{1, 4\}$, $\{5\}$, and $\{2\}$ itself, which is four. Hence $h(u) = \langle 4 \rangle - (\langle 2 \rangle + \langle 1 \rangle) = 5$, meaning the increment of node u is 5. The decrement $g(v)$ of v requires only traversing the component $pos(v)$-th component instead of the whole graph with a modified *Connect* algorithm [Hopcroft and Tarjan, 1973]. Then the objective function $f(S')$ after move (u, v) can be directly computed with the formula below.

$$f(S') = f(S) + h(u) - g(v) \tag{3}$$

By bringing in the component configuration, IEM computes the pairwise connectivity efficiently during the search, and the correctness of IEM is not difficult to prove.

3.2 Computational Complexity

Now we analyse the computational complexity of IEM. For the step of **Initialization**, we need a traveral on the whole graph to obtain the initial component configuration, thus the complexity is $O(|V|+|E|)$. When IEM updates I after move (u, v), as described in **Update**, the complexity includes two parts, where the first part is to connect several components into one component (line 4-7) with complexity $O(|V_C| + |E_C|)$ (C is the component containing u), and the second part splits graph with v (line 9-12), which needs a traversal

through the component of v with a complexity $O(|V_{C'}| + |E_{C'}|)$ (C' is the component containing v). At **Evaluation** step, we compute increment $h(u)$ with only a complexity $O(D(G))$ by formula (2), where $D(G)$ is the maximum node degree in G. While for $g(v)$, we also need to traverse its component C_v, but fortunately it can be done simultaneously when updating the component configuration of v (line 4-7).

We conclude the time complexity of IEM and previous evaluation method (*i.e.*, by DFS) in Table 1. The 2nd-4rd column report the step of move (u, v), time complexity, and space complexity, respectively. $|V_C| + |E_C|$ indicates the sum of the sizes of components around u and v. Initially (*#move*=0), both of origin method and IEM compute objective function from scratch. With the optimization of the solution, the size of the biggest component decreases and the computational complexity of IEM gets smaller (this is also shown in an experiment in the next section). Intuitively, IEM don't traverse the component that is not changed after move by taking the advantage of component configuration. The space complexity of IEM is linear to the size of component configuration, which is only $O(|V|)$ for $I = (\Delta, pos)$.

Method	*#move*	Time complexity	Space complexity
Origin	$n \in N$	$O(\|V\| + \|E\|)$	$O(1)$
IEM	0	$O(\|V\| + \|E\|)$	$O(\|V\|)$
	$n > 0$	$O(\|V_C\| + \|E_C\| + D(G))$	

Table 1: Comparison of complexity.

4 Applications of IEM

In this section, we evaluate the effectiveness of IEM on exploitation by applying IEM to two important greedy operations, *i.e.*, $Swap$ and $Add\text{-}Remove$. Then we evaluate the new operations on standard benchmarks for CNP. Finally, we discuss the generalization of IEM.

4.1 New Swap Operation

We mentioned that $SwapN$ is an improved version of $Swap$. Now we further improve $SwapN$ operation with IEM to get a faster operation, denoted as $SwapN+$. The pseudocode of $SwapN+$ is outlined in Algorithm 2.

In the beginning, we compute the decrement $g(v)$ for each node $v \in V \setminus S$ by a modified depth-first search process, namely *Connect* (line 1), then we initialize (u^*, v^*) to denote the best move (line 2). There is a loop to traverse all nodes in the current candidate solution (lines 3-8). In each loop, $SwapN+$ searches the best move for u, denoted as (u, w). If this move is better than the best move found so far within the loop, which means this move leads to a more decrement than increment, we update it (line 9-10). After the loop, $SwapN+$ computes the best neighbor (line 11) and its objective function value in an incremental way (line 12), then $SwapN+$ updates the component configuration (line 13), which is the key procedure for $SwapN+$. Finally, $SwapN+$ returns the best neighbor S^*, $f(S^*)$, and $I' = (\Delta', pos')$ (line 14).

In each loop, the increment $h(u)$ caused by reintroducing node u to the residual graph can be computed with the formula (2) (line 4). While the computation for best $g(v)$

Algorithm 2: the $SwapN+$ operation

Input : Graph G, current solution S, $f(S)$, and the component configuration $I = (\Delta, pos)$.

Output: Best neighbor S^*, $f(S^*)$, and $I' = (\Delta', pos')$.

1 $\mathcal{G}$: $\{g(v) \mid v \in V \setminus S\} \leftarrow$ *Connect*($G[V \setminus S]$);
2 Let (u^*,v^*) be an empty move;
3 **foreach** $u \in S$ **do**
4 compute increment $h(u)$ with formula (2);
5 select best node v' in the component $C_{pos(u)}$ with biggest decrement $g(v')$ by *Connect*($C_{pos(u)}$);
6 $V^* \leftarrow \{v \mid g(v) > g(v'), pos(v) \neq pos(u)\}$, $\forall g(v) \in \mathcal{G}$;
7 **if** $V^* = \emptyset$ **then** $w \leftarrow v'$;
8 **else** $w \leftarrow$ arg max $\{g(v)\}$, for $v \in V^*$;
9 **if** $g(v^*) - h(u^*) < g(w) - h(u)$ **then**
10 $(u^*, v^*) \leftarrow (u,w)$;
11 $S^* \leftarrow (S \setminus \{u^*\}) \cup \{v^*\}$;
12 $f(S^*) \leftarrow f(S) + h(u^*) - g(v^*)$;
13 $I' \leftarrow$ Update(G, (u^*, v^*), I);
14 **return** the best neighbor S^*, $f(S^*)$, and new I'.

is much more complex (line 5-8). The intuition behind this computation is to choose the biggest decrement value $g(w)$ among $\mathcal{G}$ and $g(v')$, where $\mathcal{G}$ indicates the "global optimum" when $u \in S$ (biggest decrement of all nodes in the whole residual graph) and $g(v')$ is "local optimum" (biggest decrement in a single component). Since $\mathcal{G}$ is obtained when the node u has not been put back, the value $g(v)$ in $\mathcal{G}$ may become invalide after u is put back. Therefore, we consider both of "global optimum" and "local optimum".

For example, in Figure 1(b), $\mathcal{G} = \{g(1) = 1, g(3) = 6, g(4) = 1, g(5) = 3, g(6) = 3, g(8) = 3\}$. If $u = 7$, meaning to put node 7 back, then $h(u) = \langle 5 \rangle - \langle 4 \rangle = 4$. The best removed node of 'local optimum' is $v' = 3$ and $g(v') = 7$. Because we compute $g(v')$ by assuming put node 7 back, the previous computed values $g(3)$, $g(5)$, $g(6)$, and $g(8)$ become invalid. By comparing $g(v')$ with $g(1)$ and $g(4)$, we get the best removed node is 3.

With the method described above, we iteratively update the best move (u^*, v^*) until the end of the loop. Compared with $SwapN$, $SwapN+$ doesn't need to traverse the whole graph for each step of the loop, resulting in a more efficient greedy operation. Moreover, we improve $Add\text{-}Remove$ (AR for short) with *Neighborhood* N_1 [Aringhieri *et al.*, 2016b], named ARN and its IEM version $ARN+$. Due to the limit of space, we will not go into details of AR but directly show the complexity comparison in the following table.

Table 2 illustrates the complexity of origin $Swap$ operation, $SwapN$ and $SwapN+$, where $C = (V_C, E_C)$ is the largest component during the search and $D(G)$ is the maximum node degree in G. The time complexity of $SwapN+$ is obtained by replacing $(|V| + |E|)$ in $SwapN$ with the time

complexity of IEM, which is $(|V_C| + |E_C| + D(G))$, along with only one DFS (with complexity $O(|V| + |E|)$) out of loop to compute $\mathcal{G}$ in line 1. For CNP, the average size of all components is usually small, especially when the candidate solution is close to the optimum, thus $SwapN+$ has lower complexity than $SwapN$. While $SwapN+$ will make little difference if the average size of the component is close to the whole graph G, meaning that the graph is dense. In this case, the complexity of $SwapN+$ is $O(K(|V_C| + |E_C| + D(G)))$.

Operations	Time Complexity
$Swap$	$O(K(\|V\| - K)(\|V\| + \|E\|))$
$SwapN$	$O(K(\|V\| + \|E\|))$
$SwapN+$	$O(\|V\| + \|E\| + K(\|V_C\| + \|E_C\| + D(G)))$
AR	$O(\|V\|(\|V\| + \|E\|))$
ARN	$O(K(\|V\| + \|E\|))$
$ARN+$	$O(\|V\| + \|E\| + K \times D(G))$

Table 2: Comparison results in greedy operations.

4.2 Experiments on Greedy Operations

To evaluate the effectiveness of $SwapN+$ and $ARN+$, we conduct an experiment by applying $SwapN$, $SwapN+$, ARN, and $ARN+$ into the same generic local search algorithm for CNP, which is based on the Algorithm 4 of [Aringhieri *et al.*, 2016b], resulting in four corresponding algorithms. All of four algorithms begin with a random initial solution, then continuously move to next candidate solution with each operation until reaching the local optimum. If one algorithm stucks in local optimum, it will restart with a new random solution. We run four algorithms on linux machine with 3.60 GHz Intel Core i7 and 8GB RAM. Timeout is set to 30 minutes for each algorithm.

Table 3 presents the comparison results on those operations. The 1st column indicates two popular benchmarks for CNP, which are Synthetic benchmark set [Ventresca, 2012] with 16 instances and Real-world benchmark set [Aringhieri *et al.*, 2016a] with 26 instances. The 2nd-3rd (resp. 5th-6th) columns report the number of iterations of $SwapN$ and $SwapN+$ (resp. ARN and $ARN+$). The column 4 and 7 indicates the percentage of promotion on the number of iteratations improved by IEM when compared with $SwapN$ and $ARN+$, respectively. In this table, The results show that IEM dramaticlly improves the speed of iterations except one instance ('astroph') on $Swap$ operation. The reason is that the complexity of $SwapN+$ is not better than $SwapN$ when the graph is dense and the quality of current candidate solution is bad, which means the graph is still connected after removing K nodes randomly. However, it can be solved easily by initializing the candidate solution with a *vertex cover* for a well developed algorithm.

4.3 Discussions on IEM

Essentially, the main contribution of IEM is to speed up computing objective function, hence, it can be directly used in many local search algorithms, *e.g.*, the state-of-the-art algorithm, named MACNP [Zhou *et al.*, 2018].

Besides, the idea of IEM can be generalized to many variants of CNDP. Recall that CNDP is a class of problems

Instance	$SwapN$	$SwapN+$	$r(\%)$	ARN	$ARN+$	$r(\%)$
BA500	2354961	12045646	**412**	9884803	115423335	**1068**
BA1000	394975	5358276	**1257**	3139508	30548651	**873**
BA2500	97552	1571078	**1511**	644693	10994273	**1605**
BA5000	30711	855233	**2685**	177194	5479938	**2993**
ER235	2670521	4619664	**73**	12600767	154235620	**1124**
ER466	756549	1017988	**35**	3988013	36974789	**827**
ER941	93375	170834	**83**	1721909	12816408	**644**
ER2344	13750	21353	**55**	150771	2217001	**1370**
FF250	1941061	6540028	**237**	19976115	125905719	**530**
FF500	469623	5293437	**1027**	3840637	62425631	**1525**
FF1000	107054	646575	**504**	1318338	19639388	**1390**
FF2000	53206	194193	**265**	276358	9233735	**3241**
WS250	294506	428755	**46**	4599399	19357836	**321**
WS500	138297	173045	**25**	1791512	10601070	**492**
WS1000	7495	8069	**8**	453195	1551581	**242**
WS1500	5109	6706	**31**	195775	1586618	**710**
Bovine	62336488	82270817	**32**	225450249	240661909	**7**
Circuit	3047339	6242722	**105**	18324596	88362803	**382**
E.Coli	4469988	8724364	**95**	21363723	51461413	**141**
USAir97	404767	714375	**76**	8459496	24101060	**185**
HumanDis	548757	2771335	**405**	5175371	39819422	**669**
TrainsRome	5208927	28361856	**444**	41935093	150339452	**259**
EU_flights	2171	2337	**8**	109403	273729	**150**
openflights	6437	14091	**119**	437980	4145663	**847**
yeast	40000	278030	**595**	371973	11835872	**3082**
Ham1000	40844	43629	**7**	722435	5548467	**668**
Ham2000	5802	5923	**2**	161859	1192399	**637**
Ham3000a	1733	2390	**38**	70091	539796	**670**
Ham3000b	1728	1902	**10**	64246	514915	**701**
Ham3000c	1734	3345	**93**	65790	539024	**719**
Ham3000d	1728	3340	**93**	68221	563669	**726**
Ham3000e	1729	3334	**93**	68384	529632	**674**
Ham4000	744	936	**26**	36725	560771	**1427**
Ham5000	387	700	**81**	20955	210134	**903**
powergrid	5831	40500	**595**	34155	4015521	**11657**
OClinks	2931	5379	**84**	125293	635792	**407**
facebook	63	92	**46**	9629	33471	**248**
grqc	1117	1662	**49**	26218	2678589	**10117**
hepth	71	112	**58**	6238	234933	**3666**
hepph	7	12	**71**	2156	22562	**946**
astroph	2	2	0	694	3395	**389**
condmat	3	4	**33**	809	22454	**2676**

Table 3: Comparison results on the number of iterations.

to identify the critical nodes in a network, aiming to evaluate the robustness of the whole network. For a given candidate solution S, the objective function of S is often related to all nodes in the graph. For instance, *MaxNum* is a variant of CNDP whose goal is to degrades the number of components in the residual graph after deleting K nodes. By using IEM, such objective value can be computed easier than CNP.

5 Experiments

In this section, we adopt two popular benchmarks, *i.e.*, Synthetic benchmark set and Real-world benchmark set, which are mentioned before. Then we evaluate the effectiveness of IEM by applying it to the state-of-the-art algorithm.

Since the source code of the state-of-the-art algorithm MACNP is not available, we reimplemented their algorithm completely based on their paper [Zhou *et al.*, 2018] and another version equipped with IEM. Although IEM improves the results of MACNP, its performance is still not comparable with the results attained by running their binary code[2]. Considering the study of particle swarm optimization algorithms for CNP is an interesting research direction that has not been investigated before [Lalou *et al.*, 2018], we implemented PSOCNP to solve CNP based on a discrete particle swarm optimization algorithm [Pan *et al.*, 2008]. Based on IEM, we implemented an improved version of PSOCNP, named

[2]Available at http://www.info.univ-angers.fr/pub/hao/cnps.html

Instance	$\|V\|$	$\|E\|$	K	MACNP				PSOCNP				PSOCNP+IEM			
				f_{best}	f_{avg}	t_{avg}	$\#s$	f_{best}	f_{avg}	t_{avg}	$\#s$	f_{best}	f_{avg}	t_{avg}	$\#s$
BA500	500	499	50	**195**	**195.0**	$<$ **0.1**	30	**195**	**195.0**	0.5	30	**195**	**195.0**	$<$ **0.1**	30
BA1000	1000	999	75	**558**	**558.0**	0.3	30	**558**	**558.0**	8.5	30	**558**	**558.0**	**0.1**	30
BA2500	2500	2499	100	**3704**	**3704.0**	0.6	30	**3704**	**3704.0**	45.9	30	**3704**	**3704.0**	**0.5**	30
BA5000	5000	4999	150	**10196**	**10196.0**	3.3	30	**10196**	**10196.0**	248.9	30	**10196**	**10196.0**	**0.8**	30
ER235	235	350	50	**295**	**295.0**	2.9	30	**295**	**295.0**	30.1	30	**295**	**295.0**	**0.2**	30
ER466	466	700	80	**1524**	**1524.0**	21.6	30	**1524**	1526.6	264.6	30	**1524**	**1524.0**	**3.6**	30
ER941	941	1400	140	**5012**	5021.2	571.1	7	5020	5058.1	1502.9	3	5014	**5016.2**	1516.8	17
ER2344	2344	3500	200	***902128**	**923349.0**	1643.1	1	1071287	1112443.4	2619.9	1	926437	939504.8	1843.0	1
FF250	250	514	50	**194**	**194.0**	$<$ **0.1**	30	**194**	**194.0**	0.4	30	**194**	**194.0**	$<$ **0.1**	30
FF500	500	828	110	**257**	**257.0**	**0.2**	30	**257**	**257.0**	20.5	30	**257**	**257.0**	**0.2**	30
FF1000	1000	1817	150	**1260**	**1260.0**	55.1	29	**1260**	**1260.0**	305.2	30	**1260**	**1260.0**	**1.1**	**30**
FF2000	2000	3413	200	**4545**	4546.4	329.3	12	**4545**	4545.6	3116.3	15	**4545**	**4545.0**	**7.4**	**30**
WS250	250	1246	70	**3083**	3132.5	1448.3	10	3085	3184.5	1513.3	3	**3083**	**3108.2**	**519.0**	**19**
WS500	500	1496	125	**2072**	2080.8	1285.7	9	2082	2088.8	636.7	3	**2072**	**2080.1**	**1019.5**	**11**
WS1000	1000	4996	200	124351	149798.0	2752.5	1	147961	157987.3	3461.1	1	**115011**	**136453.0**	2592.4	1
WS1500	1500	4498	265	**13098**	13192.2	2269.5	2	15508	16130.0	2908.4	1	**13098**	**13142.3**	**1102.8**	**12**
Bovine	121	190	3	**268**	**268.0**	$<$ **0.1**	30	**268**	**268.0**	$<$ **0.1**	30	**268**	**268.0**	$<$ **0.1**	30
Circuit	252	399	25	**2099**	**2099.0**	0.2	30	**2099**	**2099.0**	2.5	30	**2099**	**2099.0**	**0.1**	30
E.coli	328	456	15	**806**	**806.0**	$<$ **0.1**	30	**806**	**806.0**	0.1	30	**806**	**806.0**	$<$ **0.1**	30
USAir97	332	2126	33	**4336**	**4336.0**	411.2	30	**4336**	**4336.0**	118.9	30	**4336**	**4336.0**	**6.8**	30
HumanDi	516	1188	52	**1115**	**1115.0**	1.0	30	**1115**	**1115.0**	1.6	30	**1115**	**1115.0**	$<$ **0.1**	30
TreniR	255	272	26	**918**	**918.0**	**0.6**	30	**918**	**918.0**	2.2	30	**918**	**918.0**	0.8	30
EU_fli	1191	31610	119	350762	353697.0	416.4	14	351600	356217.0	3555.1	1	**348268**	**348787.8**	1968.5	18
openfli	1858	13900	186	28700	28700.0	1544.4	6	30173	32061.8	2613.3	1	**26842**	**26874.0**	1486.7	8
yeast	2018	2705	202	**1412**	1412.1	43.6	29	**1412**	**1412.0**	1078.2	**30**	**1412**	**1412.0**	**1.8**	**30**
H1000	1000	1998	100	**307355**	**311191.0**	1693.6	1	317374	320663.0	3322.0	1	310513	314699.5	957.8	1
H2000	2000	3996	200	**1256624**	1280064.0	3432.1	1	1366250	1378233.4	2638.5	1	1257009	**1273894.8**	3489.8	1
H3000a	3000	5999	300	**2881235**	2940212.4	2458.9	1	3339955	3390975.5	3532.7	1	2885914	**2909354.5**	3537.5	1
H3000b	3000	5997	300	**2879996**	2936228.7	3213.7	1	3334815	3379214.3	3579.7	1	2906794	**2924863.8**	3471.6	1
H3000c	3000	5996	300	**2856998**	2923482.1	3405.4	1	3250475	3373023.5	3380.0	1	2874324	**2914392.6**	3491.1	1
H3000d	3000	5993	300	**2881780**	2939566.7	2580.7	1	3278288	3349591.8	3571.4	1	2911738	**2930687.3**	3159.0	1
H3000e	3000	5996	300	**2895843**	2943541.0	3087.4	1	3324660	3392942.5	3593.2	1	2921898	**2941841.0**	3253.3	1
H4000	4000	7997	400	**5188610**	5352623.7	3246.6	1	6140794	6182121.0	3557.5	1	5286161	**5334647.0**	3415.6	1
H5000	5000	9999	500	8434117	8574214.7	3057.3	1	9664681	9702098.0	1429.3	1	**8399531**	**8485434.3**	3489.1	1
powergr	4941	6594	494	**15868**	15926.9	459.2	1	16162	16365.4	952.2	1	15878	**15894.0**	496.4	1
Oclinks	1899	13838	190	**614467**	615980.8	**1240.5**	**8**	622237	624889.9	3040.5	1	**614467**	**615307.6**	2419.3	7
faceboo	4039	88234	404	**698823**	**773608.5**	3440.1	1	2855818	3761080.3	3170.7	1	1217441	1503048.5	3595.5	1
grgc	5242	14484	524	13611	13642.0	2982.6	1	13877	13944.6	3383.1	1	**13609**	**13617.0**	2099.1	1
hepth	9877	25973	988	106851	108232.8	3351.6	1	10801681	14952435.0	3588.0	1	***105527**	**106073.1**	3548.5	1
hepph	12008	118489	1201	**10012131**	**10596695.2**	3444.4	1	41113046	43234228.0	3578.1	1	11294736	12271670.0	3400.0	1
astroph	18772	198050	1877	**60785097**	**62155874.6**	3351.2	1	122564685	123382768.0	3589.5	1	63348601	64934640.0	3550.4	1
condmat	23133	93439	2313	**9047266**	**9827883.0**	1893.8	1	152091300	157963360.0	3599.0	1	13165175	14492496.0	3528.4	1
$X(Y)$	*	*	*	*	*	*	*	**0(17)**	**2(14)**	**1(1)**	*	**6(21)**	**21(15)**	**14(5)**	*

Table 4: Comparison between PSOCNP, PSOCNP+IEM, and the state-of-the-art algorithm on synthetic and real-world benchmarks.

PSOCNP+IEM. Both of them are implemented in C++ and compiled by g++ with '-O3' option.

All the experiments were carried out on the same platform, which is a Linux machine equipped with an Intel i7-9800x processor with 3.6 GHz and 32 GB RAM. We rerun the binary code of MACNP. Each algorithm was tried 30 times and 3600 seconds for each trial. All the results are shown in Table 4. In this table, we use f_{best}, f_{avg}, t_{avg}, and $\#s$ to indicate the best objective value, the average objective value, the average time in seconds to attain the f_{best}, and the number of successful trials to attain f_{best}, respectively. At the last row, $X(Y)$ stands for X instances outperformed by the algorithm compared with MACNP and Y instances on which both MACNP and the algorithm achieved the same quality of the solution. The symbol '$\star$' in instance "ER2344" and "hepth" are two new found upper bounds.

In **bold** we present the best results. PSOCNP+IEM outperforms PSOCNP in all instances, which means it attained better f_{best} and f_{avg} in 27 instances and achieved the same quality solution but faster in another 15 instances. Although the results of PSOCNP is not comparable with MACNP, we delightedly found PSOCNP+IEM spent less time to attain f_{best} in 14 instances compared with MACNP, and it significantly improves the quality of f_{avg} in 21 hard instances. Thanks to IEM, the PSOCNP has been improved to be a competitive algorithm.

6 Conclusions and Future Work

This paper focused on the computation of the objective function for CNP. We introduced a new mechanism, called IEM, to compute the objective function incrementally in low complexity. To evaluate the effectiveness of IEM, we compared $Swap$ operation equipped with IEM with the previous one, leading to a faster exploitation process. Besides, we use a simple PSOCNP and its IEM version to compare with the state-of-the-art algorithm MACNP. The experimental results show that the IEM significantly improves the performance of PSOCNP.

In the future, we plan to further study the variants of CNDP with IEM, and to improve the current PSOCNP algorithm. Moreover, it is interesting to seek for more efficient evaluation methods to compute the objective function.

References

[Addis *et al.*, 2016] Bernardetta Addis, Roberto Aringhieri, Andrea Grosso, and Pierre Hosteins. Hybrid constructive heuristics for the critical node problem. *Annals of Operations Research*, 238(1-2):637–649, 2016.

[Aringhieri *et al.*, 2015] Roberto Aringhieri, Andrea Grosso, Pierre Hosteins, and Rosario Scatamacchia. VNS solutions for the Critical Node Problem. *Electronic Notes in Discrete Mathematics*, 47:37–44, 2015.

[Aringhieri *et al.*, 2016a] Roberto Aringhieri, Andrea Grosso, Pierre Hosteins, and Rosario Scatamacchia. A general Evolutionary Framework for different classes of Critical Node Problems. *Engineering Applications of Artificial Intelligence*, 55:128–145, 2016.

[Aringhieri *et al.*, 2016b] Roberto Aringhieri, Andrea Grosso, Pierre Hosteins, and Rosario Scatamacchia. Local search metaheuristics for the critical node problem. *Networks*, 67(3):209–221, 2016.

[Arulselvan *et al.*, 2007] ASHWIN Arulselvan, Clayton W Commander, Panos M Pardalos, and OLEG Shylo. Managing network risk via critical node identification. *Risk management in telecommunication networks*, 2007.

[Arulselvan *et al.*, 2009] Ashwin Arulselvan, Clayton W. Commander, Lily Elefteriadou, and Panos M. Pardalos. Detecting critical nodes in sparse graphs. *Computers & Operations Research*, 36(7):2193–2200, 2009.

[Boginski and Commander, 2009] Vladimir Boginski and Clayton W Commander. Identifying critical nodes in protein-protein interaction networks. In *Clustering challenges in biological networks*, pages 153–167. 2009.

[Cormen *et al.*, 2009] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. 2009.

[Di Summa *et al.*, 2011] Marco Di Summa, Andrea Grosso, and Marco Locatelli. Complexity of the critical node problem over trees. *Computers & Operations Research*, 38(12):1766–1774, 2011.

[Di Summa *et al.*, 2012] Marco Di Summa, Andrea Grosso, and Marco Locatelli. Branch and cut algorithms for detecting critical nodes in undirected graphs. *Computational Optimization and Applications*, 53(3):649–680, 2012.

[Fan and Pardalos, 2010] Neng Fan and Panos M Pardalos. Robust optimization of graph partitioning and critical node detection in analyzing networks. In *International Conference on Combinatorial Optimization and Applications*, pages 170–183, 2010.

[Hopcroft and Tarjan, 1973] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. 16(6):7, 1973.

[Lalou *et al.*, 2018] Mohammed Lalou, Mohammed Amin Tahraoui, and Hamamache Kheddouci. The Critical Node Detection Problem in networks: A survey. *Computer Science Review*, 28:92–117, 2018.

[Leskovec *et al.*, 2007] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. In *SIGKDD13*, pages 420–429, 2007.

[Pan *et al.*, 2008] Quan-Ke Pan, M Fatih Tasgetiren, and Yun-Chia Liang. A discrete particle swarm optimization algorithm for the no-wait flowshop scheduling problem. *Computers & Operations Research*, 35(9):2807–2839, 2008.

[Shen *et al.*, 2013] Yilin Shen, Nam P Nguyen, Ying Xuan, and My T Thai. On the discovery of critical links and nodes for assessing network vulnerability. *IEEE/ACM Transactions on Networking*, 21(3):963–973, 2013.

[Tomaino *et al.*, 2012] Vera Tomaino, Ashwin Arulselvan, Pierangelo Veltri, and Panos M Pardalos. Studying connectivity properties in human protein–protein interaction network in cancer pathway. In *Data Mining for Biomarker Discovery*, pages 187–197. 2012.

[Ventresca and Aleman, 2014a] Mario Ventresca and Dionne Aleman. A derandomized approximation algorithm for the critical node detection problem. *Computers & Operations Research*, 43:261–270, 2014.

[Ventresca and Aleman, 2014b] Mario Ventresca and Dionne Aleman. A Fast Greedy Algorithm for the Critical Node Detection Problem. In Zhao Zhang, Lidong Wu, Wen Xu, and Ding-Zhu Du, editors, *Combinatorial Optimization and Applications*, volume 8881, pages 603–612. Cham, 2014.

[Ventresca and Aleman, 2014c] Mario Ventresca and Dionne Aleman. A region growing algorithm for detecting critical nodes. In *International Conference on Combinatorial Optimization and Applications*, pages 593–602, 2014.

[Ventresca, 2012] Mario Ventresca. Global search algorithms using a combinatorial unranking-based problem representation for the critical node detection problem. *Computers & Operations Research*, 39(11):2763–2775, 2012.

[Veremyev *et al.*, 2014a] Alexander Veremyev, Vladimir Boginski, and Eduardo L Pasiliao. Exact identification of critical nodes in sparse networks via new compact formulations. *Optimization Letters*, 8(4):1245–1259, 2014.

[Veremyev *et al.*, 2014b] Alexander Veremyev, Oleg A Prokopyev, and Eduardo L Pasiliao. An integer programming framework for critical elements detection in graphs. *Journal of Combinatorial Optimization*, 28(1):233–273, 2014.

[Zhou and Hao, 2017] Yangming Zhou and Jin-Kao Hao. A fast heuristic algorithm for the critical node problem. In *GECCO17*, pages 121–122, Berlin, Germany, 2017.

[Zhou *et al.*, 2018] Yangming Zhou, Jin-Kao Hao, and Fred Glover. Memetic search for identifying critical nodes in sparse graphs. *IEEE Transactions on Cybernetics*, pages 1–14, 2018.