# Hybrid constructive heuristics for the critical node problem

**Annals of Operations Research manuscript No.**
(will be inserted by the editor)

# Hybrid Constructive Heuristics for the Critical Node Problem

**Bernardetta Addis · Roberto Aringhieri · Andrea Grosso · Pierre Hosteins**

Received: date / Accepted: date

**Abstract** We consider the *Critical Node Problem*: given an undirected graph and an integer number $K$, at most $K$ nodes have to be deleted from the graph in order to minimize a connectivity measure in the residual graph. We combine the basic steps used in common greedy algorithms with some flavour of local search, in order to obtain simple hybrid heuristic algorithms. The obtained algorithms are shown to be effective, delivering improved performances (solution quality and speed) with respect to known greedy algorithms and other more sophisticated state of the art methods.

**Keywords** Critical Node Problem · Graph Fragmentation · Hybrid Heuristics

## 1 Introduction

The *Critical Node Problem (CNP)* is to determine a set of vertices in a graph whose deletion results in a graph having the minimum pairwise connectivity.

To the authors' knowledge, the origin of the problem can be traced back to the so-called *network interdiction problems* studied by Wollmer [25] and later by Wood [26], although these seminal papers focused on arc deletion. Due also to the renewed emphasis on security-related research in networks (see [13]) recently the attention has moved to node deletion problems. In particular, when the assessment of the robustness of communication network is considered (see [10, 9]), the deleted nodes in the solution of CNP, represent the critical node of the network. Further applications of CNP arise in different contexts. In [5] the CNP is stated in the context of detecting so-called "key players" in a relational network. In [3] and [18] contagion control via vaccination of

Work supported by a Google Focused Grant on Mathematical Programming, project "Exact and Heuristic Algorithms for Detecting Critical Nodes in Graphs"

B. Addis
LORIA (CNRS UMR 7503), Université de Lorraine , INRIA Nancy Grand Est
E-mail: bernardetta.addis@loria.fr

R. Aringhieri, A. Grosso, P. Hosteins
Dipartimento di Informatica, Università degli Studi di Torino
E-mail: {roberto.aringhieri,andrea.grosso,pierre.hosteins}@unito.it

a limited number of individuals is considered: the nodes of the graph are potentially infected individuals and the edges represent contacts occurring between them.

In this paper we consider the CNP formulated as follows. Given an undirected graph $G = (V, E)$ and an integer $K$, determine a subset of $K$ nodes $S \subseteq V$, such that the number of node pairs still connected in the induced subgraph $G[V \setminus S]$

$$f(S) = |\{i, j \in V \setminus S \colon i \text{ and } j \text{ are connected by a path in } G[V \setminus S]\}|$$

is as small as possible. The CNP is known to be NP-complete [3], polynomially solvable on trees [7] and other specially structured graphs via dynamic programming: graphs with bounded tree-width [1] and series-parallel graphs [16].

The CNP on general graphs has been tackled by branch and cut methods. A MILP model is proposed in [3]; a very large model relying on constraint separation is used in the branch and cut methods proposed in [8]; recently a compact MILP formulation has been proposed in [21].

A heuristic approach based on finding an indepent set, coupled with a 2-exchange local search phase is proposed in [3]. Metaheuristics — namely population-based incremental learning and simulated annealing — are studied and experimentally compared in [18]. An approximation algorithm based on a randomized rounding of the relaxed linear programming model is proposed in [19]. Negative results for approximation, outside of the probabilistic framework, are provided in [1]. For a broad literature review, including problems with different metrics about graph fragmentation, we refer to [13, 17, 22–24] and references within.

State of the art literature considers sets of benchmark instances whose size is up to few thousand nodes (but usually smaller). On the other side, instances coming from applications can be considerably larger. For this reason, computationally efficient algorithms capable to deal with such instances are still worth to investigate.

Our focus is on developing computationally efficient heuristics able to deal with large instances of CNP. Instead of adapting classical metaheuristics to the problem, we concentrate on ad-hoc greedy methods proposed for CNP, combining them in order to obtain new ad-hoc heuristics with some flavor of local search.

Simple constructive heuristics and their hybridization are discussed in Section 2. Computational tests are reported and discussed in Section 3.

## 2 The algorithms

Observe that deleting any $S \subseteq V$ that is a vertex cover for $G$ completely disconnects the graph, giving $f(S) = 0$ (only the complementary independent set survives). Since in general a vertex cover has more than $K$ nodes, such $S$ is infeasible for the CNP. Anyway it is possible to build a feasible solution by iteratively *adding back* to the graph one node at a time. Based on this rule, Arulselvan et al. [3] propose Algorithm 1, that we call *Greedy1*.

An efficient implementation of *Greedy1* (Algorithm 1) deals with the key step on line 3 by keeping trace of the connected components of $G[V \setminus S]$. It looks at all neighbours of all nodes in $S$ in order to determine the connected components that can be fused together and the dimension of the new component. If the connected components of the residual graph are stored, this requires at most $\mathcal{O}(|E|)$ operations; after the best node has been chosen, and added back to the graph (i.e. deleted from $S$), the other nodes in $V \setminus S$ are explored to update the component they now belong to. The

**Algorithm 1:** *Greedy1*

```
   Data: Graph: $G$, $K$
   Result: $S^*$
1  Set $S$ to a vertex cover of $G$;
2  while $|S| > K$ do
3      $i = \arg\min\{f(S \setminus \{i\}) - f(S)\}$;
4      $S := S \setminus \{i\}$;            // Added back to $G$ $\iff$ deleted from $S$
5  $S^* := S$;
```

complexity of identifying and/or merging components does not exceed $\mathcal{O}(|V| + |E|)$. These operations are repeated until $K$ nodes remain out of the graph, meaning less than $|V| - 1 - K$. The complexity of (heuristically, greedily) generating the initial vertex cover is bounded by $\mathcal{O}(|E|)$.

*Greedy1* has the following fundamental weakness: if a node $k$ belongs to an optimal (or even only good) solution $S^*$ but it is not in the initial vertex cover, there will be no chance of bringing it into the solution. Consider for example the graph in Figure 1 with $K = 1$. The optimal solution is obviously $S^* = \{6\}$; if the minimal vertex cover computed at step 1 is $\{2, 3, 4, 5, 7, 8, 9, 10\}$, *Greedy1* is deemed to deliver the worst possible solution, leaving the graph with a single, large connected component.

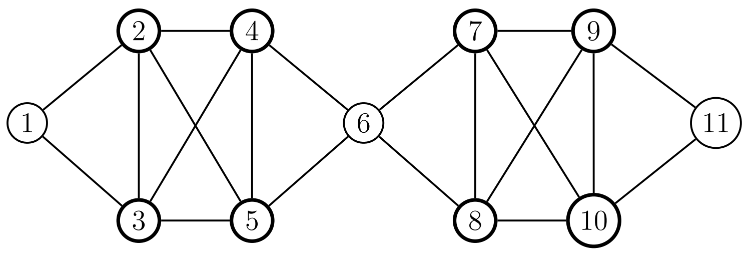


Fig. 1: Example

One might think about a completely specular approach, that — starting from the whole graph — sequentially applies *remove* operations, leading to *Greedy2* (see Algorithm 2).

**Algorithm 2:** *Greedy2*

```
   Data: Graph: $G$, $K$
   Result: $S^*$
1  $S = \{\emptyset\}$;
2  while $|S| < K$ do
3      $i = \arg\max\{f(S) - f(S \cup \{i\})\}$;
4      $S := S \cup \{i\}$;            // Added to $S$ $\iff$ removed from $G$.
5  $S^* = S$;
```

This algorithm may seem to make perfectly sense, but at a closer look, it reveals a striking weakness. Indeed, it results in a *completely random* choice of $i$, unless $G[V \setminus S]$

contains a so-called *articulation point,* i.e. a node whose removal results in splitting the graph into two or more connected components. For example, even if *Greedy2* easily handles the example of Figure 1, on the graph in Figure 2 with $K = 2$ it is likely to deliver the worst possible solution — a single large connected graph, whereas the optimal choice would obviously be to delete $\{6, 7\}$.

An efficient implementation of *Greedy2* is a bit more tricky than *Greedy1.* The search for an articulation point is done through a modified graph visit, tracking the presence of articulation points and the dimension of the connected components rooted at it (see [14] for details). In the search for $K$ nodes to delete one applies $K$ times an algorithm of complexity $\mathcal{O}(|V| + |E|)$. The recent work of [20] also proposes an efficient implementation of algorithm *Greedy2.*

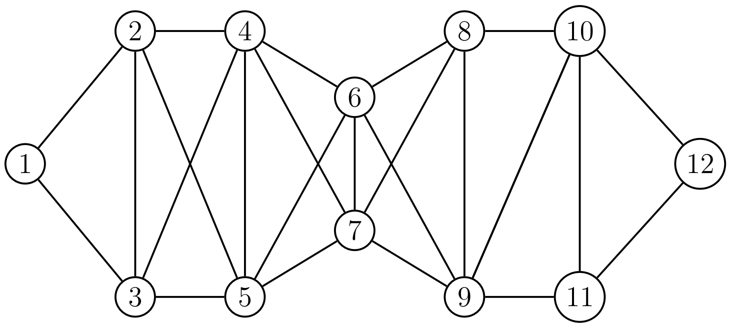


Fig. 2: Example

In order to take advantage both of the efficient add-back step as well as of the detection of articulation points, we developed a hybrid approach, alternating the application of the two basic greedy steps: *add-back* and *remove* (see lines 3-4 of *Greedy1* and *Greedy2*).

Furthermore, aiming at recovering from possibly wrong decisions, after finding a feasible solution, we allow the algorithm to move into the unfeasible region — with additional $\Delta_K^+$ *add-back* or $\Delta_K^-$ *remove* steps — before returning to a feasible solution. The exploration of the solution space benefits of a somewhat stronger perturbation than that achieved by the basic add-and drop move, still possibly preserving some good structure of the previously generated feasible solutions. The resulting algorithm, that we call *Greedy3* (see Algorithm 3), acquires some flavor of local search while retaining most of the simple structure of greedy algorithms. That's why we still call it "greedy".

*Greedy3* alternates stages where a number of *add-back* steps is applied with stages where the *remove* operation takes place. In each of these stages, a feasible solution is found (when exactly $K$ nodes are removed from the graph), and if it is better of the actual best solution found, it is saved as record. As a stopping criterion, a maximum number of generated feasible solutions $N_S$ are allowed to be evaluated, where $N_S$ is an exogenous parameter (set by the user).

As a matter of fact *Greedy3* is considerably less myiopic than the other two, since it has indeed chances of undoing wrong choices taken at early iterations. Experimental comparisons show that a single run of *Greedy3* is considerably more effective than a multistart application of the basic greedy algorithms.

```
Algorithm 3: Greedy3
Data: Graph: $G$, $K$
$N_S$: the number of feasible solutions that we are allowed to generate
Result: $S^*$
1  $S :=$ Vertex Cover($G$);
2  $n := 0$, $S^* := \{\emptyset\}$, $f(S^*) := \infty$ ;
3  repeat
4      while $|S| > K$ do
5          $i = \arg\min\{f(S\setminus\{i\}) - f(S)\}$;
6          $S := S \setminus \{i\}$;
7      if $f(S) \leq f(S^*)$ then
8          $S^* := S$;
9      $n := n + 1$;
10     while $|S| > K - \Delta_K^-$ do
11         $i = \arg\min\{f(S\setminus\{i\}) - f(S)\}$;
12         $S := S \setminus \{i\}$;
13     while $|S| < K$ do
14         $i = \arg\max\{f(S) - f(S \cup \{i\})\}$;
15         $S := S \cup \{i\}$;
16     if $f(S) \leq f(S^*)$ then
17         $S^* := S$;
18     $n := n + 1$;
19     while $|S| < K + \Delta_K^+$ do
20         $i = \arg\max\{f(S) - f(S \cup \{i\})\}$;
21         $S := S \cup \{i\}$;
22 until $n > N_S$;
```

```
Algorithm 4: Greedy4
Data: Graph: $G$, $K$
$N_S$: the number of feasible solutions that we are allowed to generate
Result: $S^*$
1  $S := \emptyset$;
2  $n := 0$, $S^* := \{\emptyset\}$, $f(S^*) := \infty$;
3  repeat
4      while $|S| < K$ do
5          $i = \arg\max\{f(S) - f(S \cup \{i\})\}$;
6          $S := S \cup \{i\}$;
7      if $f(S) \leq f(S^*)$ then
8          $S^* := S$;
9      $n := n + 1$;
10     while $|S| < K + \Delta_K^+$ do
11         $i = \arg\max\{f(S) - f(S \cup \{i\})\}$;
12         $S := S \cup \{i\}$;
13     while $|S| > K$ do
14         $i = \arg\min\{f(S\setminus\{i\}) - f(S)\}$;
15         $S := S \setminus \{i\}$;
16     if $f(S) \leq f(S^*)$ then
17         $S^* := S$;
18     $n := n + 1$;
19     while $|S| > K - \Delta_K^-$ do
20         $i = \arg\min\{f(S\setminus\{i\}) - f(S)\}$;
21         $S := S \setminus \{i\}$;
22 until $n > N_S$;
```

The procedure we use for extracting a vertex cover is a modified version of the greedy heuristic which selects the node with highest degree, adds it to the cover, deletes all adjacent edges, and then repeats until the graph is empty (see, e.g., [15]). Our modifications are concerned with an initial shuffling of the nodes in such a way to consider them in random order instead of by decreasing order of node degree, and a pre-processing phase to take out the nodes of degree 1.

Following the same logic we also define *Greedy4* (see Algorithm 4) that starts, similarly to *Greedy2*, with a *remove* stage.

***Dynamically restarting the search.*** Computational experience showed that our mechanism of sequentially adding and removing $\Delta_K^{\pm}$ nodes around the feasible solution, implemented in *Greedy3* and *Greedy4*, is not always sufficient to avoid being trapped in "bad" solutions. Therefore, when a solution does not improve after a given number of iterations $\mathcal{I}$, we allow the algorithm to perform a complete restart (i.e., generating a new vertex cover from scratch). We kept the same stopping criterion, allowing $N_S$ feasible solutions to be evaluated. This new algorithm is called *Greedy3d* (that is *Greedy3* with dynamical restart). Performances improve, at the modest cost of handling the additional parameter $\mathcal{I}$. A similar modification was applied to *Greedy4*, leading to a dynamically restarted procedure *Greedy4d*.

## 3 Computational results

In order to experimentally evaluate the algorithms we relied on two test beds. The first set is built on the basis of the 16 graphs used in [18]. They are divided into 4 types of graphs with different structures, and with number of nodes ranging from 250 to 5000 (see Table 1).

We got several instances of CNP by choosing various values for the parameter $K$ from the set $\{5, 10, 20, 30, 50, 75, 100, 150, 200, 300, 500, 1000, 1500, 3000\}$ (always keeping $K < |V|$); we decided however to delete each instance for which we can certify a solution with 0 value, regarding such instances as easy ones. Therefore we got a set of 159 instances — we label it "set 1" in the following.

| | Barabasi-Albert (BA) | Erdos-Renyi (ER) | Forest-Fire (FF) | Watts-Strogatz (WS) |
|---|---|---|---|---|
| $\|V\|$,$\|E\|$,$K$ | 500,499,50 | 235,350,50 | 250,514,50 | 250,1246,70 |
| | 1000,999,75 | 466,700,80 | 500,828,110 | 500,1496,125 |
| | 2500,2499,100 | 941,1400,140 | 1000,1817,150 | 1000,4996,200 |
| | 5000,4999,150 | 2344,3500,200 | 2000,3413,200 | 1500,4498,265 |

Table 1: Sizes of the graphs from [18] from which set 1 is derived.

In order to obtain larger instances we downloaded example graphs from The Stanford Large Network Dataset Collection (SNAP)[1]— mainly graphs from the Internet peer-to-peer networks category and the Collaboration networks category. These instances represent respectively 8 networks of computers connected to the internet and exchanging files, plus 5 networks of physicists collaborating around the world and linked by their published works on the website `www.arxiv.org`. The size of such graphs is reported in Table 2.

Given the large number of nodes and the substantial difference between the graphs' dimensions (up to a factor of 6), we choose to select values of $K$ as fractions of the number of nodes $|V|$. In practice we choose five values of $K$ for each graph. The two types of graphs do not have the same density, therefore, in order to obtain interesting instances, we set the values for $K$ as:

$$\text{peer-to-peer:} \quad K \in \{0.01|V|, 0.05|V|, 0.1|V|, 0.15|V|, 0.2|V|\}.$$

$$\text{scientific collaborations:} \quad K \in \{0.1|V|, 0.2|V|, 0.3|V|, 0.4|V|, 0.5|V|\}.$$

[1] `http://snap.stanford.edu/data/`

Larger values of $K$ easily lead to completely disconnected graphs with an optimum that is easily detected by all the algorithms. The total number of obtained instances in the test bed is 65 — we call it "set 2" in what follows.

Peer to peer

| Graph name | $\|V\|$ | $\|E\|$ |
|---|---|---|
| p2p1 | 10876 | 39994 |
| p2p2 | 8846 | 31893 |
| p2p3 | 8717 | 31525 |
| p2p4 | 6301 | 20777 |
| p2p5 | 8114 | 26013 |
| p2p6 | 26518 | 65369 |
| p2p7 | 22687 | 54705 |
| p2p8 | 36682 | 88328 |

Scientific collaboration

| Graph name | $\|V\|$ | $\|E\|$ |
|---|---|---|
| grqc | 5242 | 14484 |
| hepth | 9877 | 25973 |
| hepph | 12008 | 118489 |
| astroph | 18772 | 198050 |
| condmat | 23133 | 93439 |

Table 2: Graphs taken from SNAP from which set 2 is derived.

We compare the relative performances of the various algorithms by means of *performance profiles*. The performance profile for each algorithm $A$ on a collection of instances $T$ is the function defined by

$$p_A(n) = \frac{|\{I \in T : A(I) \leq 2^n \mathrm{BEST}(I)\}|}{|T|}$$

where $A(I)$ is the result of algorithm $A$ on instance $I$ and $\mathrm{BEST}(I)$ is the best result obtained in the test campaign (with all the considered algorithm) for instance $I$; the point $(n, p(n))$ on the curve denotes the fraction of tested instances for which algorithm $A$ delivered a solution with a relative error [2] less then or equal to $2^n - 1$ with respect to the best result found. Note that the curve works with a logarithmic scale; particularly, the relative error obtained by the algorithm grows exponentially as a function of $n$:

| $n$ | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathrm{Err}_{\%}$ | 0 | 7.2 | 14.9 | 23.1 | 32.0 | 41.4 | 51.6 | 62.5 | 74.1 | 86.6 | 100.0 |

When plotting the curves for two algorithms $A$, $A'$, algorithm $A$ can be considered better than $A'$ if the performance profile of $A$ lies at north-west of the profile of $A'$. See [11] for a detailed introduction.

All tests were performed on a server equipped with two AMD Opteron 8425HE processors, 2.1 GHz clock and 16 GB RAM running Linux, the code is developped in C++.

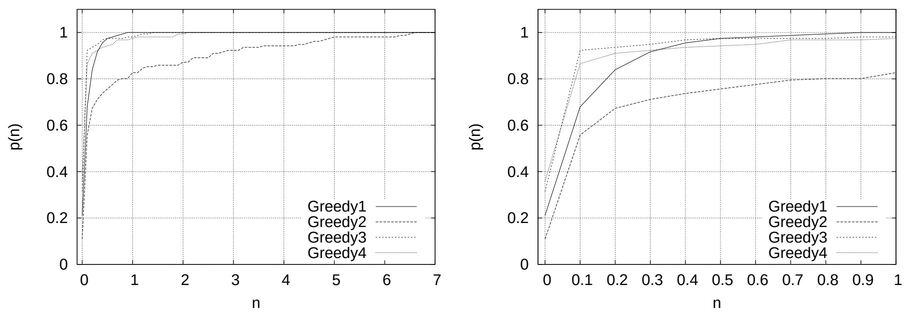
As for the algorithms' parameters we set values for $\varDelta_K^{\pm}$ and $\mathcal{I}$ after a few preliminary experiments. We chose $\varDelta_K^{\pm} = K/2$ for all the new algorithms and $\mathcal{I} = 5$ for the dynamic restarted versions. This choice gave satisfactory results without asking for an excessive effort in calibration.

[2] Calculated as
$$\frac{A(I) - \mathrm{BEST}(I)}{\mathrm{BEST}(I)}$$

***Comparing the basic approaches.*** We tested *Greedy3* and *Greedy4* all instances in set 1, allowing a single run on each instance, fixing for both $N_S = 60$.

In order to fairly compare *Greedy3* and *Greedy4* against *Greedy1* and *Greedy2*, we ran *Greedy1* and *Greedy2* on a multistart basis, allowing 60 runs for each algorithm and keeping the best solution delivered.

The overall results are presented in the performance profile shown in Figure 3.



(a) complete curves (b) initial part of the same curves

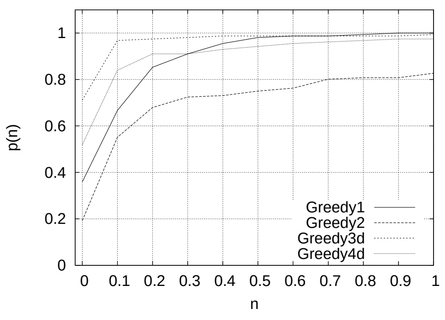
Fig. 3: Performance profiles for *Greedy1*, *Greedy2*, *Greedy3*, *Greedy4* . Tests performed on set 1.

Algorithm *Greedy2* exhibits by far the worse behavior, being strongly outperformed by all the others. *Greedy3* and *Greedy4* offer the best performances with *Greedy3* being slightly better. Although *Greedy1* performances seem not too far from *Greedy3* and *Greedy4*, a relevant difference is remarkable on the first part of the profiles, meaning that *Greedy3* and *Greedy4* are actually significantly better at delivering best solutions, with *Greedy1* being only able to deliver a best solution in 20% of the tests, while *Greedy3* and *Greedy4* deliver the best solution in approximately 35% of the tests. Furthermore, if we look more into detail (see Figure3b), we can observe that accepting a relative error smaller then 10%, *Greedy3* and *Greedy4* are able to "solve" 90% of the instances, while *Greedy1* only around 70%.

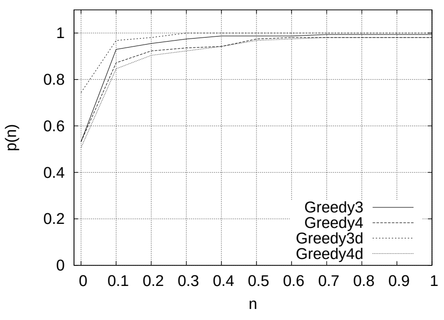
We also ran the same tests with all the algorithms being stopped after the same amount of running time. Each instance was solved by *Greedy3* with $N_S = 60$, keeping track of the running time. The same amount of CPU time was allotted to the multistart application of the other algorithms. We got completely similar results, with also a slightly improved performance record for *Greedy3* and *Greedy4*.

***Dynamic restart.*** We tested *Greedy3d* and *Greedy4d* with a single run on each instance of set 1, with $N_S = 60$; *Greedy1* and *Greedy2* were tested in a multistart fashion as described above over 60 runs. The results over set 1 are summarized by the performance profiles in Figure 4.

The profiles show that *Greedy1* and *Greedy2* are strongly outperformed by *Greedy3d* and *Greedy4d*, while also *Greedy3* and *Greedy4* are outperformed by *Greedy3d*. Hence adding the dynamic restart to *Greedy3* proves to be fruitful. It is interesting to note that adding the dynamic restart to *Greedy4* is not as beneficial as in the *Greedy3* case.



(a) *Greedy1*, *Greedy2*, *Greedy3d*, *Greedy4d*



(b) *Greedy3*, *Greedy4*, *Greedy3d*, *Greedy4d*

Fig. 4: Performance profiles on set 1.

***Comparison with previous approaches.*** Since we used graphs generated in previous works on the CNP [18], we will compare the results of our algorithms *Greedy3d* and *Greedy4d* with those delivered by the metaheuristic (**S**imulated **A**nnealing and **P**opulation-**B**ased **I**ncremental **L**earning) algorithms presented in [18]. Results for these two algorithms are presented in the cited paper for 30 runs, with information on the best and worst results as well as the average result for the objective. We ran a single execution of our algorithms with $N_S = 30$. We compare our results with the best results found by SA and PBIL[3]. The results are summarized in Table 3. We warn the reader that the tests for SA and PBIL reported in [18] ran on a different machine (Intel i7-2600 K, 3.4 GHz clock, 8 GB RAM). The cpu times reported in Table 3 for SA and PBIL are the cumulative times for 30 runs, with the objective function value being the best value delivered over all such runs. It is interesting to observe how *Greedy3d* and *Greedy4d* find the best results in all cases except one, often with a large gap compared to SA and PBIL. We do not aim at a fine-grained comparison, but the large differences in the solution quality and computation times cannot be explained only by the different machines. *Greedy3d* and *Greedy4d* can then outperform metaheuristics like SA and PBIL, that should perform extensive exploration of the solution space. Hence we take the figures of Table 3 as an indication of the effectiveness of our algorithms.

The results are somewhat surprising since a well tailored metaheuristic with sufficient running time would be expected to outperform a greedy-based approach. A possible explanation could reside in the fact that evaluating the cost of a local search move has a priori the complexity of a DFS on the graph, making the local search part of an algorithm very slow. A way around this situation is provided in a companion work [2] in the form of a Variable Neighbourhood Search (VNS) algorithm. Such a metaheuristic is able to provide better quality solution than algorithms *Greedy3d* and *Greedy4d* described in this work, however the two types of algoritm have different scopes: while the VNS is able to identify potentially better solutions, it is much slower and therefore more limited on large graphs. In order to fairly compare *Greedy3d* and *Greedy4d* with such a VNS algorithm, we use a maximum running time for the VNS which is equal to the maximum running time between *Greedy3d* and *Greedy4d*. We use

[3] the values of $K$ for the FF graphs are different from those printed in [18] — they have been corrected in [12].

| Graph | $K$ | SA | | PBIL | | *Greedy3d* | | *Greedy4d* | |
|---|---|---|---|---|---|---|---|---|---|
| | | Obj. Val | Time | Obj. Val | Time | Obj. Val | Time | Obj. Val | Time |
| ER235 | 50 | 7700 | 1140 | 6700 | 2250 | 315 | 1 | **313** | <1 |
| ER466 | 80 | 48627 | 3300 | 44255 | 5490 | **1938** | 1 | 1993 | 1 |
| ER941 | 140 | 234479 | 1083 | 229576 | 14070 | **8106** | 3 | 8419 | 3 |
| ER2344 | 200 | 2011122 | 57930 | 2009132 | 65130 | 1118785 | 12 | **1112685** | 10 |
| BA500 | 50 | 997 | 1980 | 892 | 3780 | **195** | <1 | **195** | 1 |
| BA1000 | 75 | 3770 | 5160 | 3057 | 7920 | **559** | 2 | **559** | 1 |
| BA2500 | 100 | 31171 | 25200 | 28044 | 35340 | **3722** | 9 | **3722** | 6 |
| BA5000 | 150 | 170998 | 94620 | 146753 | 105450 | **10196** | 27 | **10196** | 18 |
| WS250 | 70 | 14251 | 2100 | 13786 | 4050 | 11694 | <1 | **11401** | 1 |
| WS500 | 125 | 54201 | 5190 | 53779 | 7890 | **4818** | 2 | 11981 | 2 |
| WS1000 | 200 | 311700 | 16440 | **308596** | 20280 | 316416 | 9 | 318003 | 8 |
| WS1500 | 265 | 717369 | 54480 | 703241 | 61920 | **157621** | 15 | 243190 | 11 |
| FF250 | 50 | 1841 | 1110 | 1386 | 2640 | 199 | 1 | **197** | 0 |
| FF500 | 110 | 2397 | 4680 | 1904 | 6690 | **262** | 1 | 264 | 1 |
| FF1000 | 150 | 92800 | 123000 | 59594 | 15270 | 1288 | 4 | **1271** | 4 |
| FF2000 | 200 | 387248 | 51690 | 256905 | 58830 | 4647 | 12 | **4592** | 11 |

Table 3: Results of the algorithms SA and PBIL from [18], *Greedy3d* and *Greedy4d* presented here, on the graphs introduced by [18]. The best of four results is displayed in bold font.

a VNS with First Improvement strategy, which is the most competitive version of the algorithm presented in [2].

Another type of algorithm which has been developed for dealing with critical node problems are greedy heuristics based on the so-called centrality measures: the principle is to delete the node having the highest value with respect to a given centrality measure, update the centrality values and repeat the operation until $K$ nodes are deleted. In order assess the quality of our results, we also compute solutions using two centrality-based greedy algorithms: one called $G^{cent}_{deg}$ where the centrality values are the degree of each node in the graph and one called $G^{cent}_{bet}$, based on the betweenness centrality of the nodes (see [6] for definition and computation). The centrality values are recomputed at each step, which makes $G^{cent}_{bet}$ extremely slow because betweenness centrality is computed through a procedure of complexity $\mathcal{O}(|V||E|)$ [6]. In order to maintain a fair comparison, these greedy algorithms are restarted until 30 feasible solutions have been constructed and only the best result is reported, while the time value represents the total running time over 30 runs.

The results from the comparisons are displayed in Table 4. Results in boldface are reported for the algorithm reaching the best solution. Results in *italic* font are reported in order to highlight cases where a competitor delivers better results that our *Greedy* algorithms. VNS manages to find better solutions than both *Greedy3d* and *Greedy4d* in only 4 instances over 16 (ER466, ER941, WS500). This is a hint that, although the VNS framework is most promising when sufficient running time is available, both types of algorithm are actually complementary. As far as the comparison with centrality-based algorithms is concerned, it can be noted that good quality results are achieved by $G^{cent}_{bet}$ but at the price of an extremely high computational time. While $G^{cent}_{deg}$ is unable to give satisfying results compared to *Greedy3d* and *Greedy4d*, we see that $G^{cent}_{bet}$ provides good quality results for the densest graphs of the Watts-Strogatz type, which is probably due to the difficulty of greedy rules to distinguish between the different nodes at each step. Nonetheless, the very large running times for recomputing the centrality values make the use of $G^{cent}_{bet}$ somewhat unrealistic when it comes to real-world instances.

| Graph | $K$ | *Greedy3d* | | *Greedy4d* | | VNS | | $G_{deg}^{cent}$ | | $G_{bet}^{cent}$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Obj. Val | Time | Obj. Val | Time | Obj. Val | Time | Obj. Val | Time | Obj. Val | Time |
| ER235 | 50 | 315 | 1 | **313** | <1 | 316 | 1 | 499 | 1 | 378 | 194 |
| ER466 | 80 | 1938 | 1 | 1993 | 1 | **1924** | 1 | 4636 | 9 | 2118 | 2167 |
| ER941 | 140 | 8106 | 3 | 8419 | 3 | **7687** | 3 | 70871 | 59 | 8582 | 26698 |
| ER2344 | 200 | 1118785 | 12 | **1112685** | 10 | 1323124 | 12 | 1811785 | 526 | 2063566 | 876024 |
| BA500 | 50 | **195** | <1 | **195** | 1 | 196 | 1 | 197 | 5 | 199 | 283 |
| BA1000 | 75 | **559** | 2 | **559** | 1 | 559 | 2 | 578 | 34 | 559 | 1922 |
| BA2500 | 100 | **3722** | 9 | **3722** | 6 | 3722 | 9 | 4153 | 305 | 3726 | 20444 |
| BA5000 | 150 | **10196** | 27 | **10196** | 18 | 10222 | 27 | 12626 | 1812 | 10216 | 151750 |
| WS250 | 70 | 11694 | <1 | 11401 | 1 | 12613 | 1 | 16110 | 2 | **9529** | 633 |
| WS500 | 125 | 4818 | 2 | 11981 | 2 | *3154* | 2 | 67189 | 15 | **2786** | 4100 |
| WS1000 | 200 | 316416 | 9 | 318003 | 8 | *310117* | 9 | 319600 | 85 | **203340** | 114473 |
| WS1500 | 265 | 157621 | 15 | 243190 | 11 | 161844 | 15 | 761995 | 239 | **19511** | 233341 |
| FF250 | 50 | 199 | 1 | **197** | 0 | 199 | 1 | 241 | 2 | 198 | 143 |
| FF500 | 110 | **262** | 1 | 264 | 1 | 269 | 1 | 291 | 15 | 264 | 889 |
| FF1000 | 150 | 1288 | 4 | **1271** | 4 | 1330 | 4 | 1679 | 88 | 1289 | 10555 |
| FF2000 | 200 | 4647 | 12 | **4592** | 11 | 4691 | 12 | 7660 | 478 | 4676 | 62427 |

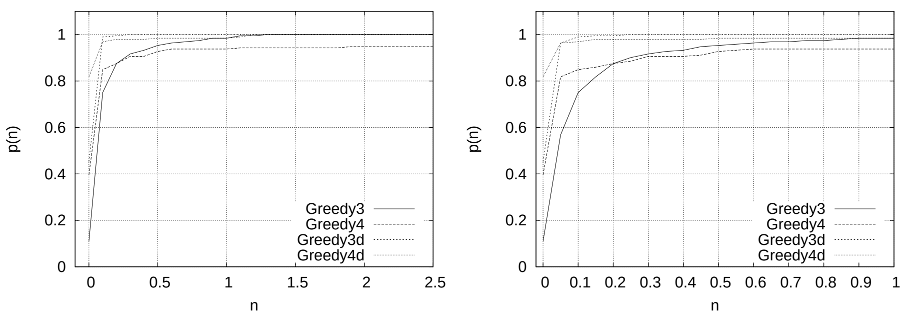
Table 4: Results of the greedy algorithms based on degree and betweenness centrality, together with results from a VNS algorithm, on the graphs introduced by [18].

***Results on large real-world instances.*** Figure 5 contains performance profiles drawn for the testing of *Greedy3*, *Greedy4*, *Greedy3d* and *Greedy4d* on the instance set named set 2. Algorithms *Greedy3d* and *Greedy4d* appear to strongly dominate *Greedy3* and *Greedy4*. Also, for this test set *Greedy4d* shows much better performances than those obtained on set 1. Hence on set 2 *Greedy4* strongly benefits from the addition of the dynamic restart feature. By examining the behavior of the algorithms on some instances, we explain this phenomenon as follows. Our algorithms, although incorporating some flavor of neighborhood search through perturbating the solutions, they are still — and this was in the scope of the research — greedy-like, in the sense that the neighborhood exploration is kept limited. The first feasible solution generated by *Greedy4* is basically obtained by an application of *Greedy2*, which has proved to offer really poor performances. A very bad solution generated at the first stage is unlikely to be so strongly improved in successive stages. The dynamic restart offers a chance to restart the search from a (possibly, *very*) different solution, especially on a large graph. On the other hand, *Greedy3* benefits from a first stage that applies the logic of *Greedy1*, which is much more effective than *Greedy2*, hence the dynamic restart has a still a beneficial but somehow milder impact.

On the large instances of set 2, *Greedy3d* and *Greedy4d* offer the best performances. *Greedy4d* offered a substantially larger number of best solutions found on set 2 — on more than 80% of the tests, while *Greedy3d* only his the best results in 45% on the instances. Nevertheless, if we accept a relative error up to 3% both algorithms offer such precision on 96% of the instances, while for the remaining instances *Greedy4d* can have a relative error of over 100% while *Greedy3d* remains within a relative error of 20%.

## 4 Conclusion

Building on top of basic greedy algorithms, we proposed “hybrid” heuristic algorithms in order to tackle the Critical Node Problem by combining the two most basic greedy rules. While more straightforward greedy algorithms start from the original graph or a completely fragmented graph and respectively delete or add back the nodes one by one,



(a) complete curves (b) initial part of the same curves

Fig. 5: Performance profiles obtained on set 2, for *Greedy3*, *Greedy4*, *Greedy3d*, *Greedy4d*.

we chose to alternatively add and delete nodes around a feasible solution, in order to get out of local minima. The results are clearly in favour of our new approach, especially when allowing for a dynamic restarting of the algorithm after a certain amount of non-improving iterations. We further validated these results by applying the algorithms on larger graphs representing peer-to-peer networks or scientific collaborations. The results are encouraging and can be taken as a hint that greedy methods for the CNP should be applied to practical applications such as computational biology, as suggested by [4], or any field that requires to find maximal fragmentation of large networks. One possible step forward would be the inclusion of global information on the graph at hand in order to help minimize the bad random choices of the greedy algorithms.

## References

1. Addis, B., Di Summa, M., Grosso, A.: Removing critical nodes from a graph: complexity results and polynomial algorithms for the case of bounded treewidth. Discrete Applied Mathematics **16-17**, 2349–2360 (2013)
2. Aringhieri, R., Grosso, A., Hosteins, P., Scatamacchia, R.: VNS solutions for the critical node problem. Electronic Notes in Discrete Mathematics **47**, 37–44 (2015). Proceedings of the VNS'14 conference
3. Arulselvan, A., Commander, C.W., Elefteriadou, L., Pardalos, P.M.: Detecting critical nodes in sparse graphs. Computers & Operations Research **36**, 2193–2200 (2009)
4. Boginski, V., Commander, C.W.: Identifying critical nodes in protein-protein interaction networks. In: S. Butenko, W.A. Chaovalitwongse, P.M. Pardalos (eds.) Clustering Challenges in Biological Networks, pp. 153–168. World Scientific Publishing (2009)
5. Borgatti, S.P.: Identifying sets of key players in a network. Computational and Mathematical Organization Theory **12**, 21–34 (2006)
6. Brandes, U.: A faster algorithm for betweenness centrality. Journal of Mathematical Sociology **25**, 163–177 (2001)
7. Di Summa, M., Grosso, A., Locatelli, M.: The critical node problem over trees. Computers and Operations Research **38**, 1766–1774 (2011)
8. Di Summa, M., Grosso, A., Locatelli, M.: Branch and cut algorithms for detecting critical nodes in undirected graphs. Computational Optimization and Applications **53**, 649–680 (2012)
9. Dinh, T., Xuan, Y., Thai, M., Pardalos, P., Znati, T.: On new approaches of assessing network vulnerability: Hardness and approximation. IEEE/ACM Transactions on Networking **20**, 609–619 (2012)

10. Dinh, T.N., Thai, M.T.: Precise structural vulnerability assessment via mathematical programming. In: MILCOM 2011 – 2011 IEEE Military Communications Conference, pp. 1351–1356. IEEE (2011)
11. Dolan, E., Moré, J.: Benchmarking optimization software with performance profiles. Mathematical Programming **91**(2), 201–13 (2002)
12. Edalatmanesh, M.: Heuristics for the critical node detection problem in large complex networks. Ph.D. thesis, Faculty of Mathematics and Science, Brock University, St. Catharines, Ontario (2013)
13. Golden, B.L., Shier, D.R. (eds.): Network Interdiction Applications and Extensions (2014). Virtual issue on Networks.
14. Hopcroft, J., Tarjan, R.: Algorithm 447: Efficient algorithms for graph manipulation. Communications of the ACM **16**(6), 372–378 (1973)
15. Papadimitriou, C., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, NJ (1982)
16. Shen, S., Smith, J.: Polynomial-time algorithms for solving a class of critical node problems on trees and series-parallel graphs. Networks **60**(2), 103–119 (2012). DOI 10.1002/net.20464
17. Shen, S., Smith, J., Goli, R.: Exact interdiction models and algorithms for disconnecting networks via node deletions. Discrete Optimization **9**, 172–88 (2012)
18. Ventresca, M.: Global search algorithms using a combinatorial unranking-based problem representation for the critical node detection problem. Computers & Operations Research **39**, 2763–2775 (2012)
19. Ventresca, M., Aleman, D.: A derandomized approximation algorithm for the critical node detection problem. Computers and Operations Research **43**, 261–270 (2014)
20. Ventresca, M., Aleman, D.: Efficiently identifying critical nodes in large complex networks. Computational Social Networks **2**(1), 6 (2015). DOI 10.1186/s40649-015-0010-y
21. Veremyev, A., Boginski, V., Pasiliao, E.: Exact identification of critical nodes in sparse networks via new compact formulations. Optimization Letters **8**, 1245–1259 (2014)
22. Veremyev, A., Prokopyev, O., Pasiliao, E.: An integer programming framework for critical elements detection in graphs. Journal of Combinatorial Optimization **28**, 233–273 (2014)
23. Veremyev, A., Prokopyev, O., Pasiliao, E.: Critical nodes for distance-based connectivity and related problems in graphs. Networks **66**, 170–195 (2015)
24. Walteros, J., Pardalos, P.: Selected topics in critical element detection. In: N.J. Daras (ed.) Applications of Mathematics and Informatics in Military Science, *Springer Optimization and Its Applications*, vol. 71, pp. 9–26. Springer New York (2012). DOI 10.1007/978-1-4614-4109-0_2
25. Wollmer, R.: Removing arcs from a network. Operations Research **12**, 934–940 (1964)
26. Wood, R.K.: Deterministic network interdiction. Mathematical and Computer Modelling **17**, 1–18 (1993)