Complexity of the Critical Node Problem over trees

Marco Di Summa, Andrea Grosso

Dipartimento di Informatica, Università di Torino
Corso Svizzera, 185, 10149 Torino, Italy
e-mail: {disumma,grosso}@di.unito.it

Marco Locatelli

Dipartimento di Ingegneria Informatica, Università di Parma
Via G.P. Usberti, 181/A, 43124 Parma, Italy
e-mail: locatell@ce.unipr.it

Abstract

In this paper we deal with the Critical Node Problem (CNP), i.e., the problem of searching for a given number K of nodes in a graph G, whose removal minimizes the (weighted or unweighted) number of connections between pairs of nodes in the residual graph. In particular, we study the case where the physical network represented by graph G has a hierarchical organization, so that G is a tree. The $\mathcal{NP}$-completeness of this problem for general graphs has been already established (Arulselvan et al.). We study the subclass of CNP over trees, generalizing the objective function and constraints to take into account general nonnegative "costs" of node connections and "weights" for the nodes that are to be removed. We prove that CNP over trees is still $\mathcal{NP}$-complete when general connection costs are specified, while the cases where all connections have unit cost are solvable in polynomial time by dynamic programming approaches. For the case with nonnegative connection costs and unit node weights we propose an enumeration scheme whose time complexity is within a polynomial factor from $\mathcal{O}(1.618034^n)$, where n is the number of nodes of the tree. Results from computational experiments are reported for all the proposed algorithms.

Keywords: Complexity, Critical Node Problem, Multicut in Trees, Dynamic Programming

1 Introduction

Given an undirected graph $G(V, E)$ with $|V| = n$ nodes, the Critical Node Problem (CNP) calls for removing from G a subset of nodes $S \subseteq V$ in order to minimize some connectivity measure in the subgraph $G[V - S]$ induced by $V - S$, while a constraint on the size or "weight" of S has to be enforced. In a possible — and fairly general — formulation, a nonnegative connection cost c_{ij} is specified for each pair of distinct nodes $i, j \in V$, a weight $w_j \geq 0$ for each $j \in V$ and a bound K are given. Two nodes i, j are connected if a path exists between them. The optimal solution is required to

$$\begin{aligned} &\text{minimize } f(S) = \sum \{c_{ij} \colon i, j \text{ are connected in } G[V - S]\} \\ &\text{subject to } \sum_{j \in S} w_j \leq K. \end{aligned}$$

The problem has attracted some attention in recent years; especially the case where $c_{ij} = 1$ for all $i \neq j$, $w_j = 1$ for all $j \in V$ has been tackled in the literature. In such a case the problem amounts to removing at most $K \leq n$ nodes, minimizing the number of pairs connected in the residual graph. Applications of CNP considered in the literature include: fragmentation of *terrorist networks*, where a fixed number of persons has to be identified in such a way that their removal will result in the minimum communication between the remaining individuals (see [8]); *network immunization*, where a graph representing contacts between people is given, only a given maximum number of persons can be vaccinated, and we aim at minimizing the propagation of the virus (see [4, 10]); *transportation networks*, where identifying critical nodes, i.e., nodes whose failure would highly compromise the efficiency of the transportation, is quite important for a correct allocation of the resources (see [6]); *telecommunication networks*, when we want to prevent the spread of a virus or find some way to reduce as much as possible the communication within the network (see [5]).

Among recent works about the subject we mention that of Borgatti (see [2]), where different measures of fragmentation within a network, including the one employed in CNP, are introduced and a greedy heuristic is proposed, and the work by Arulselvan et al. (see [1]). In the latter work a detailed computational study is presented, including an integer linear programming model and a heuristic algorithm for the case with all unit weights and all unit costs. Also, the $\mathcal{NP}$-completeness of CNP on general graphs is proven. The proposed model is a binary linear programming problem with $O(n^2)$ variables and $O(n^3)$ constraints. The recognition version of CNP with unit connection costs and node weights is proven to be an $\mathcal{NP}$-complete problem through a reduction from the maximum independent set problem. Finally, the proposed heuristic performs a greedy construction starting from an independent set of nodes, followed by a refinement phase based on local search. The results and computation times of the heuristic are compared with those obtained by `CPLEX` applied to the proposed mathematical model of CNP.

In this paper we are interested in studying the complexity of a special case of CNP, namely the case where the graph G is a tree. This case is relevant if the (terrorist, communication, etc.) network to be fragmented has some kind of hierarchical organization, i.e., each entity in the network (node in the graph) directly communicates only with its "boss" (its parent node in the graph) and with its "subalterns" (its children nodes in the graph). This paper offers complexity results and algorithms for different versions of CNP on trees, depending on whether the costs c_{ij} and weights w_j are general nonnegative integers or all have unit values, as summarized in the following table.

c_{ij}	w_j	complexity
≥ 0	≥ 0	strongly $\mathcal{NP}$-complete
≥ 0	$= 1$	strongly $\mathcal{NP}$-complete
$= 1$	≥ 0	solvable in $\mathcal{O}(n^7)$
$= 1$	$= 1$	solvable in $\mathcal{O}(n^3K^2)$

While for general graphs the $\mathcal{NP}$-completeness of the general version of the CNP follows from the results of [1], it turns out that for the case of trees there is a gap between the complexities of the version with arbitrary nonnegative costs and that with unit costs.

In Section 2 we prove that the CNP over trees is strongly $\mathcal{NP}$-complete for general costs c_{ij}, even if $w_j = 1$ for all $j \in V$. In Section 3 we present a dynamic programming approach for the case with unit costs and unit node weights whose complexity is polynomial; in Section 4 we similarly deal with the case with unit costs and general node weights. In Section 5 we propose an enumeration scheme for the case with general costs and unit node weights, whose time complexity is within a polynomial factor from $\mathcal{O}(1.618034^n)$. In Section 6 we present and discuss the results of some computational experiments carried out with all the algorithms introduced in the paper.

2 Complexity of the general problem over trees

When the graph involved in the CNP is a tree $T(V, E)$, each pair of nodes is connected by exactly one path. In order to establish the complexity of the CNP over trees we need to introduce the following problem.

`MULTICUT IN TREES` **(MCT)** Given an undirected tree $T = (V, E)$, a collection H of pairs of nodes

$$H = \{\{u_1, v_1\}, \{u_2, v_2\}, \dots, \{u_t, v_t\}\}$$

and a weight w_e for each edge $e \in E$, determine a subset (multicut) $W \subseteq E$ with minimum total weight $\sum_{e \in W} w_e$, whose removal disconnects each pair of nodes in H.

In [7] it has been proven that MCT is $\mathcal{NP}$-complete even in the unweighted case, where $w_e = 1$ for all $e \in E$ and a minimum-cardinality subset W is required. The decision version of the unweighted-MCT asks whether there exists a multicut W with $|W| \leq B$, for a given bound B. We describe a polynomial reduction from the decision version of unweighted MCT to the decision version of CNP on trees. Given an instance of unweighted-MCT with $T = (V, E)$ with $|V| = n$, we define an instance of CNP that works on a tree $T' = (V', E')$ as follows.

- The vertex set of T' contains a copy of all the nodes of T plus a pair of nodes e_1, e_2 for each edge e of T:

  $$V' = V \cup \left[\bigcup_{e \in E} \{e_1, e_2\} \right]. \tag{1}$$

  For each edge $e = \{i, j\}$ of T, a 3-edges chain $i - e_1 - e_2 - j$ appears in T' (see Figure 1), i.e. the edge set E' is defined as

  $$E' = \bigcup_{e=\{i,j\} \in E} \{\{i, e_1\}, \{e_1, e_2\}, \{e_2, j\}\}. \tag{2}$$

  Note that removing edge e from T disconnects two nodes $i, j \in V$ if and only if removing one of nodes e_1, e_2 from T' disconnects i, j.

- We assign the following costs to the connections between nodes $p, q \in V'$, where M is a constant strictly greater than $|H|$ (for example, $M = 1 + n(n-1)/2$ is enough):

  $$c_{pq} = \begin{cases} 1 & \text{if } p, q \in V \text{ and } \{p, q\} \in H; \\ M & \text{if } p, q \in V' - V \text{ and } p = e_1, q = e_2 \text{ for some } e \in E; \\ 0 & \text{in all other cases.} \end{cases} \tag{3}$$

- We assign $w_j = 1$ to each $j \in V'$ and require that no more than $K = B$ nodes are removed. The decision problem asks whether there exists a subset $S \subseteq V'$ such that the total cost of the surviving paths when S is removed is not greater than

$$K' = M(n - K - 1). \tag{4}$$

It can be easily seen that the construction is computable in polynomial time.

We prove the following

Proposition 2.1. *CNP on trees is $\mathcal{NP}$-complete.*

Proof. CNP on trees is easily seen to be a member of $\mathcal{NP}$. We prove

Unweighted-MCT $\propto$ CNP on trees.

Given an instance I of MCT, we build an instance I' of CNP according to (1)–(4).

Assume that I is a yes-instance. Then there exists a subset W with (no more than) K edges whose removal from T disconnects all pairs in H. In I', we can select the K nodes in S as follows:

select exactly one between nodes e_1, e_2 for each $e \in W$.

By removing the nodes in S from T', exactly K connections with cost M are broken, and no more than $(n-K-1)$ of them survive; also, all the unit-cost connections between pairs of nodes in H are broken. Hence the total cost of the surviving connections is at most $M(n-K-1)$.

Conversely, assume that I' is a yes-instance. Then there exists a subset $S \subseteq V'$ with (at most) K nodes such that removing S from T' leaves only a set of connected pairs with total cost $\leq K' = M(n-K-1)$. We observe that

(i) S must be composed only of nodes from $V' - V$, and

(ii) for each pair of additional nodes e_1, e_2 (with $e \in E$) at most one of them can belong to S.

Indeed, if (i) or (ii) is violated, at most $K-1$ connections with cost M can be removed and the total cost of the surviving connections would be at least $M(n-K) > K'$. With (i) and (ii) holding true, exactly K connections with cost M are broken; then the total cost of the surviving connections is at least $M(n-K-1)$, and not greater than $M(n-K-1) + |H|$. The lower bound is attained only if all the unit-cost connections between pairs of nodes in H are broken.

Hence we can remove in I a set W of at most K edges separating all pairs in H by setting

$$e \in W \iff \{e_1, e_2\} \cap S \neq \emptyset,$$

and I is a yes-instance. □

Remark 1. We note that the reduction does not exponentially inflate the numbers in Unweighted-MCT, which is not a number problem. This also establishes $\mathcal{NP}$-completeness in the strong sense for the general version of the CNP on trees, ruling out the possibility of finding a pseudopolynomial algorithm, unless $\mathcal{P} = \mathcal{NP}$. Because of the reduction involved, the proof of Proposition 2.1 also establishes the strong $\mathcal{NP}$-completeness for the special case with general connection costs c_{ij} and unit weights $w_j = 1$ for all $j \in V$.

Remark 2. The above reduction can be modified so as to show that the CNP over trees is $\mathcal{NP}$-complete even with unit weights w_j and 0-1 connection costs c_{ij}. To see this, given an instance I of Unweighted-MCT as above, we create an instance I' of the CNP by replacing each edge $e = \{i, j\}$ with a chain $i - e_0 - e_1 - \cdots - e_M - j$, where M is an integer strictly greater than $|H|$. Given two nodes $p, q \in V'$, the connection cost is

$$c_{pq} = \begin{cases} 1 & \text{if } p, q \in V \text{ and } \{p, q\} \in H; \\ 1 & \text{if } p = e_0, q = e_\ell \text{ for some } e \in E \text{ and } \ell > 0; \\ 0 & \text{in all other cases.} \end{cases}$$

The rest of the construction is as above. Note that there is an optimal solution to I' in which only nodes of the type e_0 for some $e \in E$ are removed. Now a proof similar to that of Proposition 2.1 applies.

Remark 3. We also note that the problem with general connection costs c_{ij} and node weights w_j remains $\mathcal{NP}$-complete, at least in the ordinary sense, on even very simple topologies, like the star graph. Figure 2 sketches a simple reduction PARTITION $\propto$ CNP for a star graph. Note that the star case is trivial when $w_j = 1$ for all $j \in V$. A simple reduction can also be worked out for the case where the graph is only a tree with maximum degree 2 (a path).

3 The unit-costs, unit-weights case on trees

In this section we illustrate a polynomial algorithm for solving CNP on trees when $c_{ij} = 1$ for all i, j and $w_j = 1$ for all $j \in V$. In this case the problem calls for minimizing the number of paths surviving in a tree $T(V, E)$ after having removed at most K nodes.

Given the tree $T(V, E)$ with $|V| = n$, let T_a be the subtree of T rooted at $a \in V$. If a is not a leaf of T, let $T_{a_1}, \ldots, T_{a_s}$ be the subtrees of T_a rooted at the children nodes $a_1, \ldots, a_s$ respectively, where s depends on a (see Figure 3). Let also $|T_a|$ be the number of nodes in T_a. In order to solve the problem by dynamic programming, we define the following functions.

$F_a(m, k)$ = minimum number of paths surviving in T_a when k nodes are removed from T_a and m nodes of T_a are still connected to a. Note that the number of nodes connected to some given $v \in V$ always counts v itself. Condition $m = 0$ indicates that a is removed from T_a. Furthermore, if it is not possible to remove k nodes from T_a so that m nodes of T_a are still connected to a, then we define $F_a(m, k) = \infty$.

$G_{a_i}(m, k)$ = number of paths surviving in the subtree $T_{a_{i,s}} = a + T_{a_i} + \cdots + T_{a_s}$ when k nodes are removed from $T_{a_{i,s}}$ and m nodes of the subtree are still connected to a. As above, $m = 0$ indicates that a is removed from $T_{a_{i,s}}$ and $G_{a_i}(m, k) = \infty$ if it is not possible to remove k nodes from $T_{a_{i,s}}$ so that m nodes of $T_{a_{i,s}}$ are still connected to a.

The values for F and G can be computed by traversing the tree in postorder (from leaves to root), by means of the following relations:

$$F_a(m, k) = G_{a_1}(m, k) \quad \text{for any non-leaf node } a \in V; \tag{5}$$

$$G_{a_i}(m, k) = \begin{cases} \min\{F_{a_i}(p, q) + G_{a_{i+1}}(0, k - q) : \\ \qquad p = 0, \ldots, |T_{a_i}|,\ q = 0, \ldots, k - 1\} & \text{if } m = 0\ (a \in S), \\ \\ \min\{F_{a_i}(p, q) + G_{a_{i+1}}(m - p, k - q) + p(m - p) : \\ \qquad p = 0, \ldots, m - 1,\ q = 0, \ldots, k\} & \text{if } m > 0\ (a \notin S), \end{cases} \tag{6}$$

for any non-leaf node $a \in V$ and $i < s$;

the initial conditions on each leaf a and on each rightmost subtree T_{a_s} are the following:

$$F_a(m,k) = \begin{cases} 0 & \text{if } (m=0, k=1, \text{ i.e. } a \in S) \text{ or } (m=1, k=0, \text{ i.e. } a \notin S), \\ \infty & \text{otherwise}, \end{cases} \tag{7}$$

$$G_{a_s}(m,k) = \begin{cases} \infty & \text{if } m=k=0, \\ \min\{F_{a_s}(p,k-1) : p=0,\dots,|T_{a_s}|\} & \text{if } m=0,\ k>0\ (a \in S), \\ F_{a_s}(m-1,k)+(m-1) & \text{if } m>0\ (a \notin S). \end{cases} \tag{8}$$

Equation (5) follows because $T_a = T_{a_{1,s}}$ for any non-leaf node $a \in V$.

Recursion (6) can be interpreted as follows.

- Consider first the case $m=0$ (i.e., node a is removed from the subtree), which is illustrated in Figure 4 (where $i=2$ and $k=3$). Expression $F_{a_i}(p,q)$ gives the minimum number of paths that survive in T_{a_i} when q nodes are removed from T_{a_i} and p nodes of T_{a_i} are still connected to a_i (for instance, in the example in Figure 4, for $p=3$ and $q=1$ we have $F_{a_2}(3,1)=3$: this is achieved by removing node b). Since q nodes have been removed from T_{a_i}, exactly $k-q$ nodes (including a) must be removed from $T_{a_{i+1,s}}$. The minimum number of paths that survive in $T_{a_{i+1,s}}$ when $k-q$ nodes (including a) are removed from $T_{a_{i+1,s}}$ is given by $G_{a_{i+1}}(0,k-q)$ (in the example, $G_{a_3}(0,2)=5$, which is achieved by removing nodes a and c). Thus the expression $F_{a_i}(p,q)+G_{a_{i+1}}(0,k-q)$ gives the minimum number of paths that survive in $T_{a_{i,s}}$ when q nodes are removed from T_{a_i} (and the other $k-q$ nodes are removed from $T_{a_{i+1,s}}$) and p nodes of T_{a_i} are still connected to a_i (this value is 8 in the example). By taking the minimum over $p=0,\dots,|T_{a_i}|$ and $q=0,\dots,k-1$, we find the value of $G_{a_i}(0,k)$.

- Consider now the case $m>0$, which is illustrated in Figure 5 (where $m=7$, $i=2$ and $k=2$). As above, expression $F_{a_i}(p,q)$ gives the minimum number of paths that survive in T_{a_i} when q nodes are removed from T_{a_i} and p nodes of T_{a_i} are still connected to a_i (for instance, in the example in Figure 5, for $p=3$ and $q=1$ we have $F_{a_2}(3,1)=3$: this is achieved by removing node b). Since q nodes have been removed from T_{a_i}, exactly $k-q$ nodes must be removed from $T_{a_{i+1,s}}$; and since p nodes of T_{a_i} are still connected to a_i and thus to a, exactly $m-p$

nodes of $T_{a_{i+1,s}}$ must remain connected to a. The minimum number of paths that survive in $T_{a_{i+1,s}}$ when $k-q$ nodes are removed from $T_{a_{i+1,s}}$ and $m-p$ nodes of $T_{a_{i+1,s}}$ are still connected to a is given by $G_{a_{i+1}}(m-p,k-q)$ (in the example, $G_{a_3}(4,1)=9$, which is achieved by removing node a_3). Thus the expression $F_{a_i}(p,q)+G_{a_{i+1}}(m-p,k-q)$ gives the minimum number of paths that survive in T_{a_i} or $T_{a_{i+1,s}}$ when q nodes are removed from T_{a_i} (and the other $k-q$ nodes are removed from $T_{a_{i+1,s}}$) and p nodes of T_{a_i} are still connected to a_i, while $m-p$ nodes of $T_{a_{i+1,s}}$ are still connected to a. Now we have to add the paths connecting nodes of T_{a_i} to nodes of $T_{a_{i+1,s}}$, i.e. $p(m-p)$ paths (12 paths in the example). This gives expression $F_{a_i}(p,q)+G_{a_{i+1}}(m-p,k-q)+p(m-p)$ of recursion (6) (whose value is 24 in the example). By taking the minimum over $p=0,\dots,m-1$ and $q=0,\dots,k$, we find the value of $G_{a_i}(m,k)$.

Given a leaf $a \in V$, equation (7) says that

- if a is removed ($k=1$), then no node ($m=0$) and no path survive in T_a (which becomes empty);
- if a is not removed ($k=0$), then node a survives ($m=1$), and the number of paths is again 0.

For a justification of (8), recall that $T_{a_{s,s}} = a + T_{a_s}$.

Now, assuming that T is rooted at node 1, the optimal value for the problem is given by

$$\text{OPT} = \min\{F_1(m,K) : m = 0,\dots,n\}, \tag{9}$$

and the optimal solution is recovered by backtracking.

The following proposition formally establishes that Unweighted-CNP on trees belongs to the class $\mathcal{P}$.

Proposition 3.1. *The recursion* (5)–(8) *can be computed in* $\mathcal{O}(n^3K^2)$ *time.*

Proof. For each of the n nodes there are at most $(n+1)\cdot(K+1)$ values of F and G to compute; for each (m,k) pair, $m \in \{0,\dots,n\}$, $k \in \{0,\dots,K\}$, the heaviest computation is that of equation (6) that requires at most $\mathcal{O}(nK)$ steps. Hence a time bound of $\mathcal{O}(n^3K^2)$ follows. □

We remark that this dynamic programming approach can be easily extended to the case of a forest: just add a dummy node 0 along with edges between node 0 and the root of each connected component of the forest, and then apply the above recursion to the resulting tree, with the additional condition that node 0 must be removed (this is accomplished by modifying (9) into OPT $= F_0(0, K+1)$).

4 The case with unit costs and arbitrary node weights

Let $w_j \geq 0$ be arbitrary weights assigned to the nodes $j \in V$. The CNP problem in this case amounts to finding a subset S of nodes with total weight $\sum_{j \in S} w_j$ not exceeding a given K such that the number of surviving paths after having removed the node set S is minimized.

This special case can be solved by a dynamic programming algorithm formulated in the same spirit of that in Section 3. The recursion uses two parameters m and k representing respectively the number of nodes connected to the root of a subtree and the number of paths surviving in the same subtree.

Keeping the notation for subtrees introduced in Section 3, we define the following functions.

- $F_a(m,k)$ is the minimum total weight of the nodes to be removed from the subtree T_a in order to have node a connected to exactly m nodes (including a itself) and k paths surviving in T_a.
- $G_{a_i}(m,k)$ is the minimum total weight of the nodes different from a to be removed from the subtree $T_{a_{i,s}} = a + T_{a_i} + T_{a_{i+1}} + \cdots + T_{a_s}$ in order to have a connected to m nodes of $T_{a_{i,s}}$ and k paths surviving in $T_{a_{i,s}}$.

We compute the values for F and G recursively for all $a \in V$, $m = 0, \ldots, n$, $k = 0, \ldots, n(n-1)/2$, as follows. Assume $F_a(m,k) = \infty$, $G_a(m,k) = \infty$ if $m < 0$ or $k < 0$.

$$F_a(m,k) = G_{a_1}(m,k) \qquad \text{for all non-leaf nodes } a \in V \tag{10}$$

$$G_{a_i}(m,k) = \begin{cases} \min\{F_{a_i}(p,q) + G_{a_{i+1}}(0, k-q), \\ \qquad p = 0, \ldots, |T_{a_i}|,\ q = 0, \ldots, k\} & \text{if } m = 0, \\ \min\{F_{a_i}(p,q) + G_{a_{i+1}}[m-p, k-q-p(m-p)], \\ \qquad p = 0, \ldots, m,\ q = 0, .., k\} & \text{if } m > 0. \end{cases} \tag{11}$$

Equation (11) is written for all non-leaf nodes $a_i \in V$ with $i < s$ (non-rightmost subtrees). For each rightmost subtree T_{a_s} we specify the initial condition

$$G_{a_s}(m,k) = \begin{cases} w_a + \min\{F_{a_s}(p,k),\, p = 0,\dots,|T_{a_s}|\} & \text{if } m = 0, \\ F_{a_s}(m-1, k-m+1) & \text{if } m > 0, \end{cases} \tag{12}$$

and, for every leaf a:

$$F_a(m,k) = \begin{cases} w_a & \text{if } m = 0,\, k = 0, \\ 0 & \text{if } m = 1,\, k = 0, \\ \infty & \text{in all other cases.} \end{cases} \tag{13}$$

The optimal value, assuming the tree is rooted at node 1, is given by

$$\text{OPT} = \min\{k\colon F_1(m,k) \le K, m = 0,\dots,n,\, k = 0,\dots,n(n-1)/2\}. \tag{14}$$

The optimal solution is recovered by backtracking.

The arguments justifying equations (10)–(13) are analogous to those presented for equations (5)–(8) in the previous section, and we illustrate them concisely.

For equation (10), note that $T_a = T_{a_{1,s}}$.

For equation (11) note that if $m = 0$ (i.e. a is removed), a path surviving in $T_{a_{i,s}} = T_{a_i} + T_{a_{i+1,s}}$ either belongs entirely to T_{a_i} or to $T_{a_{i+1},s}$, hence if q paths belong to T_{a_i} exactly $k - q$ belong to $T_{a_{i+1},s}$. In the latter tree, no node will be connected to a, whereas at most $|T_{a_i}|$ nodes can be connected to a_i. Hence by definition of F and G the minimum total weight of the nodes removed from $T_{a_{i,s}}$ will be $G_{a_i}(m,k) = \min_{p,q}\{F_{a_i}(p,q) + G_{a_{i+1}}(0, k-q)\}$. On the other hand, if a is not removed and it is connected to $m > 0$ nodes of T_a, a path in $T_{a_{i,s}}$ can be either completely contained in one of T_{a_i}, $T_{a_{i+1,s}}$, or partially contained in both subtrees because it passes through a. If a_i is connected to p nodes of T_{a_i} and a is connected to $m - p$ nodes in $T_{a_{i+1,s}}$, the paths passing through a are exactly $p(m-p)$. If q paths survive in T_{a_i}, $k - q - p(m-p)$ paths survive in $T_{a_{i+1,s}}$ and, by definition of F and G, $G_{a_i}(m,k) = \min_{p,q}\{F_{a_i}(p,q) + G_{a_{i+1}}[m-p, k-q-p(m-p)]\}$.

The initial condition (12) takes into account that, if node a is removed ($m = 0$), the k surviving paths of $T_{a_{s,s}} = a + T_{a_s}$ must belong entirely to T_{a_s}, hence the minimum possible weight for the nodes removed from $T_{a_{s,s}}$ will be $G_{a_s}(0,k) = \min_p\{F_{a_s}(p,k)\} + w_a$. On the other hand if node a is not removed ($m > 0$), in T_{a_s} we must have $m - 1$ nodes connected to a_s and $k - (m-1)$

surviving paths, since $m-1$ paths connect a to $m-1$ other nodes in $T_{a_{s},s}$; thus $G_{a_s}(m,k) = F_{a_s}(m-1, k-m+1)$ follows.

The initial condition (13) trivially handles the case of a one-node tree: remove the single node a (case $m = 0, k = 0$), or keep it ($m = 1, k = 0$); all other combinations of m and k are infeasible and are considered to have an infinite weight.

Proposition 4.1. *The recursion* (10)–(13) *can be computed in* $\mathcal{O}(n^7)$ *time.*

Proof. For each node $a \in V$ there are at most $n+1 = \mathcal{O}(n)$ values for m and $n(n-1)/2+1 = \mathcal{O}(n^2)$ values for k; this gives $\mathcal{O}(n^4)$ values $F_a(\cdot,\cdot)$ and $G_{a_i}(\cdot,\cdot)$ to compute. The heaviest computation lies in equation (11), where $\mathcal{O}(n)$ values are possible for p and $\mathcal{O}(n^2)$ for q. Hence in the worst case a number of operations bounded by $\mathcal{O}(n) \cdot \mathcal{O}(n^2) \cdot \mathcal{O}(n^4) = \mathcal{O}(n^7)$ is required. □

5 An enumeration scheme for the case with unit node weights

When different costs $c_{ij} \geq 0$ are specified for each pair of nodes $i \neq j$ and unit weights $w_j = 1$ are given for all nodes $j \in V$, the CNP problem is strongly $\mathcal{NP}$-hard (see Section 2), hence not even a pseudopolynomial algorithm is likely to exist. In this section we consider a superpolynomial algorithm for such a case.

Following a standard notation, let $\mathcal{O}^*(\alpha^n)$ denote a time complexity of type $\mathcal{O}(\alpha^n \text{poly}(n))$. A brute-force enumeration of all K-nodes subsets of V is viable when K is very small (or large), but in the general case the trivial enumeration algorithm requires $\mathcal{O}^*(2^n)$ time. A better enumeration scheme can be developed as follows. First of all we remark the following dominance result.

Property 5.1. *For a tree with more than two nodes, there exists an optimal solution where no leaf is removed.*

For an immediate proof, note that since the nodes have unit weights, every path broken by removing a leaf can be broken by leaving the leaf in the tree and removing the leaf's parent, instead. Hence we only consider solutions where no leaf is removed.

A tree $T(V,E)$ with depth ≤ 1 is a star with one (a single node), two (an edge) or n nodes, and an optimal solution is trivially computed by the following rule:

if $K > 0$ remove the center of the star, obtaining a solution with cost zero ;
if $K = 0$ no node can be removed, and the optimal value is $\sum\{c_{pq} \colon p, q \in V, p \neq q\}$.

Consider an instance of the problem with given $T(V, E)$, K, $\{c_{ij}\}$; if T has depth > 1 we select a node j such that all its neighbors except one are leaves and perform a branch on it. Let Γ_j be the set of leaves adjacent to j. We compute the optimal value $f^*(T, K)$ as

$$f^*(T,K) = \min \begin{cases} f^*(T', K-1) & (j \text{ is removed}) \\ \sum_{p,q \in \{j\} \cup \Gamma_j} c_{pq} + f^*(T'', K) & (j \text{ stays in the tree}). \end{cases}$$

The trees T' and T'' are defined as follows.

(i) If we remove node j, we must solve the residual problem of removing at most $K - 1$ nodes from the tree T' obtained by pruning j and all $i \in \Gamma_j$ from T.

(ii) If we do not remove j from T, we pay a constant term $\sum_{p,q \in \{j\} \cup \Gamma_j} c_{pq}$ in the objective function and then we must solve an equivalent residual problem on a tree T'', where the nodes $\{j\} \cup \Gamma_j$ are shrunk into a single node j' and the costs are given by

$$\begin{aligned} c''_{pq} &= c_{pq} & &\text{if } p, q \notin \{j\} \cup \Gamma_j, \\ c''_{pj'} &= c_{pj} + \sum_{q \in \Gamma_j} c_{pq} & &\text{for all } p \notin \{j\} \cup \Gamma_j. \end{aligned}$$

Note that, if node j is not removed from T, every path linking p and $q \in \Gamma_j$ can be broken if and only if the path linking p and j' in T'' is broken.

Refer to Figure 6, where we branch on node 6 as an example. When we dive into branch (i), the residual problem has at most $n-2$ nodes — at least node j and a leaf are pruned; when we dive into branch (ii) the residual problem has at most $n - 1$ nodes — at least node j and a leaf are shrunk. Both T' and T'' are obtained from T in at most $\mathcal{O}(n^2)$ time. Then, the running time $t(n)$ for n nodes satisfies the recursive relation

$$t(n) \leq t(n-1) + t(n-2) + \mathcal{O}(n^2).$$

Using standard techniques for handling univariate recursions (see [9] among many books on the subject) we get $t(n) \approx \mathcal{O}^*(1.618034^n)$, where the basis for the exponential is the largest real root of the associated algebraic equation $\alpha^2 - \alpha - 1 = 0$.

6 Computational experiments

Here we test the algorithms discussed in the previous sections by performing experiments on randomly generated instances of the CNP (in its various versions). In all cases the trees are generated using Broder's algorithm for the Uniform Spanning Tree problem [3]: this guarantees that each time the tree is chosen at random with uniform distribution among all trees having n nodes.

For the dynamic programming algorithms, we performed tests on an AMD Athlon 1.5GHz PC with 512 MB RAM. In order to save time, we tested the superpolynomial algorithm of Section 5 on a faster Xeon 2.33 GHz processor with 8 GB RAM.

6.1 Unit costs and unit node weights

We tested the $\mathcal{O}(K^2n^3)$ dynamic programming algorithm — called DP$_1$ in what follows — of Section 3 on randomly generated trees for number of nodes ranging from 10 to 200; the parameter K has been set to values $\lfloor \alpha n/2 \rfloor$ for $\alpha = 0.2, 0.4, 0.6, 0.8, 1.0$. For each value of n, ten random trees were generated. Table 1 reports the average and maximum running times for $n = 50, \ldots, 200$. In order to save some CPU time, the algorithm is implemented so that the F and G matrices in general are not completely filled; instead the value of each state is computed and stored only when it is actually required to compute other states. Anyway we remark that in most cases 70 to 90% of the state space has to be exploited. Table 2 highlights the quadratic dependence of the running time on K and the cubic dependence on n.

6.2 Unit costs and general node weights

We tested the $\mathcal{O}(n^7)$ algorithm — called DP$_2$ in what follows — of Section 4 on the same randomly generated trees used for testing DP$_1$. Integer weights w_j for the nodes were randomly generated from the uniform distribution $[1, 100]$. The parameter K was fixed at $K = \lfloor \alpha W/2 \rfloor$ where $W = \sum_{i=1}^{n} w_j$, and $\alpha = 0.2, 0.4, 0.6, 0.8, 1.0$. As in the previous case, the algorithm is implemented so that the value of each state is computed only when it is actually needed. Despite the higher worst-case complexity, the algorithm handles quite easily all instances up to $n = 100$ — Table 3 summarizes the CPU times obtained for $n = 50, \ldots, 100$. Note that applying equation (14) when searching for the optimum, if the values of $k = 0, \ldots, n(n-1)/2$ are scanned in increasing order, the search terminates as soon as a value $F_1(m, k) \leq K$ is found. This offers a valuable speedup, especially on instances where low values of objective function can be reached — that is, when large values of K are involved. Because of the heavy $\mathcal{O}(n^4)$ memory requirements, we could not test instances with $n > 100$; indeed on the instances of Table 3 the applicability of DP$_2$ seems limited more by the space complexity rather than by the running time. Anyway we note that usually less than 10% of the state space had to be exploited in these tests, hence as far as this type of instances is considered, one can implement the F and G matrices by means of sparse structures (for example, linked lists) in order to attack larger examples.

We expect things go worse for DP$_2$ with a narrower distribution of the node weights and tighter values of K. Hence we tested the algorithm on the extremal case with all-unit weights $w_j = 1$, on the same instances used for DP$_1$; Table 4 reports the performance of DP$_2$ on such instances. Although with still acceptable CPU times, the performance of DP$_2$ deteriorates quickly as n grows from 50 to 100, especially for low values of K. The loss of performance is better seen when considering very small values of K, such as $K = 2, 3, \dots, 6$ — that are below $\lfloor \alpha W/2 \rfloor$ in most cases, hence do not show up in Tables 3–4. Table 5 clearly shows that for these values of K a substantial effort is required by DP$_2$; from 60 to 85% of the state space had to be computed. We suspended the tests for the $n = 100$, $K = 2$ case because times higher than 2,000 seconds were required in most cases. Note however that for such small values of K even an enumerative approach could be viable. Each of the instances tested for Table 5 is easily handled by DP$_1$ in less than half a second. However by comparing Tables 1 and 4, DP$_2$ is the winner for instances with large enough K — see the columns with $\alpha \geq 0.6$.

6.3 General costs and unit node weights

We present here the results of computational experiments testing the superpolynomial recursive algorithm of Section 5.

We used a few devices to reduce the number of subtrees analyzed by the algorithm. First of all, whenever K becomes equal to zero, clearly we do not continue the recursion on the current subtree and we just return the total weight of the remaining connections. We also exploited the following easy observations.

Observation 6.1. *The optimal solution to the CNP is zero whenever the tree admits a vertex cover with at most K nodes.*

Thus, whenever our recursive algorithm produces a new subtree T to analyze, we can compute a minimum vertex cover C of T. If $|C| \leq K$, an optimal solution for T is given by C and we do not need to continue the recursion on this subtree.

Note that a minimum vertex cover C of a tree T can be computed by means of the following well-known algorithm: select a node j such that all its neighbors except one are leaves; include j in C; prune j and iterate. This procedure can be implemented to run in linear time.

The observation below is easy to prove.

Observation 6.2. *Let τ be the cardinality of a minimum vertex cover of a tree T and let j be a node of T such that all its neighbors except one are leaves.*

(i) *If j is pruned from T (giving a tree T'), then the cardinality of a minimum vertex cover of T' is $\tau - 1$.*

(ii) *If the nodes in $j \cup \Gamma_j$ are shrunk into a single node (giving a tree T''), then the cardinality of a minimum vertex cover of T'' is either τ or $\tau - 1$.*

By the above observation, we do not need to compute a minimum vertex cover for every subtree generated throughout the recursion, but just for some of them. More specifically, assume that we know that the cardinality of a minimum vertex cover for some subtree T is $\tau > K$, and let j be the node on which the branching described in Section 5 is performed.

(i) When T' is obtained by pruning node j, the value of K has to be decreased by 1. Then, since the cardinality of a minimum vertex cover of T' is $\tau - 1$, there is no hope of finding a vertex cover with at most $K - 1$ nodes.

(ii) When T'' is obtained by shrinking the nodes in $j \cup \Gamma_j$ into a single node, the value of K does not change. Then, since the cardinality of a minimum vertex cover of T' is at least $\tau - 1$, it is possible to find a vertex cover with K nodes only if $\tau - K = 1$.

It follows that: (i) there is no need to compute a minimum vertex cover after pruning a node; (ii) if we have computed a vertex cover C for some subtree T, with $|C| > K$, there is no need to compute a minimum vertex cover before $|C| - K$ shrinking operations have been performed.

We tested the algorithm (with the speed-up techniques described above) on instances of the CNP with unit weights on nodes, where n takes all even values between 20 and 60 and K takes all even values between 0 and $n/2$. (Note that for $K \geq n/2$, there is always a vertex cover with K nodes, thus the algorithm would immediately return an optimal solution to the problem.) For each choice of n and K, ten instances were generated and solved to optimality. The weights on the edges are randomly generated integer numbers with uniform distribution in $[0, 100]$. The average computation times for each family of instances are listed in Table 6.

The results indicate that for fixed K the computation time grows quickly as n increases. For fixed n, the worst cases seem to be those with K close to $n/4$. This can be explained as follows: on the one hand, the smaller K, the smaller number of pruning operations are needed to have $K = 0$, and thus the smaller number of subtrees must be analyzed; on the other hand, when K approaches $n/2$, it is more likely that there is a vertex cover of T with cardinality not too larger than K, and thus a small number of shrinking operations is needed to obtain a subtree where a vertex cover with at most K nodes exists.

Without implementing the speed-up devices based on vertex covers, the average computation time would be much higher. In particular, many hours would be needed to solve instances with n large and K close to $n/2$, whereas these instances are solved almost instantaneously when exploiting vertex covers. As to the instances with K close to $n/4$ (which are the most time-consuming ones when using the speed-up devices), without speed-ups the average computation time would be more than the double.

7 Conclusions and future research

In this paper we studied the complexity of a subclass of the Critical Node Problem (i.e., the problem of removing K nodes in a graph G in order to maximize disconnectivity), namely the subclass of CNP over trees. $\mathcal{NP}$-completeness of CNP for general graphs was proven in [1]. On the one hand we proved that the general version of CNP over trees with different connection costs c_{ij} is still (strongly) $\mathcal{NP}$-complete through a reduction from `MULTICUT IN TREES`, and on the other hand we showed that the cases with $c_{ij} = 1$ for all pairs $i \neq j$ and unit or arbitrary node weights, are solvable in polynomial time through dynamic programming approaches. For the version with general costs and unit node weights, an enumeration scheme with time complexity cheaper than $\mathcal{O}^*(2^n)$ was proposed. We reported and discussed some computational experiments for all the algorithms presented in the paper.

Future research on this topic may involve the derivation of algorithms capable of tackling instances of the CNP on more general graphs. An integer linear programming model for the general CNP has been given in [5] (see also Section 1). An alternative model is based on the obvious fact that in order to disconnect two nodes i, j, it is necessary to remove at least one node for every path connecting i and j. Using a binary variable x_v to indicate whether node v is removed from the graph, and a variable y_{ij} to indicate whether nodes i, j are disconnected, and denoting by $\mathcal{P}(i,j)$ the set of paths between nodes i and j, the resulting formulation is the following:

$$
\begin{aligned}
\max \ & \textstyle\sum_{i,j\in V} y_{ij} && (15)\\
\text{s.t.} \ & \textstyle\sum_{v\in P} x_v \geq y_{ij}, \quad i,j \in V,\ P \in \mathcal{P}(i,j), && (16)\\
& \textstyle\sum_{v\in V} x_v \leq K, && (17)\\
& x_v, y_{ij} \in \{0,1\}, \quad v,i,j \in V. && (18)
\end{aligned}
$$

(The extension to the weighted cases is immediate). While the number of variables is polynomial, the number of constraints can be exponential in general. However, for the case of the tree, the above model constitutes an integer linear programming formulation of the CNP with $\mathcal{O}(n^2)$ variables and constraints, as each pair of nodes is linked by a unique path — the structure of the model in this case resembles that of a set-covering problem.

For the case of a general graph, we are currently studying model (15)–(18) from a polyhedral viewpoint. In particular, we are developing several families of cutting planes and testing their effectiveness when used in a branch-and-cut framework.

References

[1] Arulselvan A., Commander C.W., Elefteriadou L., Pardalos P.M., Detecting critical nodes in sparse graphs, *Computers & Operations Research*, 36, 2193–2200 (2009)

[2] Borgatti S.P., Identifying sets of key players in a network, *Computational and Mathematical Organization Theory*, 12:21–34 (2006)

[3] Broder A., Generating random spanning trees, *30th Annual Symposium on Foundations of Computer Science*, 442–447 (1989)

[4] Cohen R., Ben Avraham D., Havlin S., Efficient immunization strategies for computer networks and populations, *Physical Review Letters*, 91: 247901–247905 (2003)

[5] Commander C.W., Pardalos P.M., Ryabchenko V., Uryasev S., Zrazhevsky G., The wireless network jamming problem, *Journal of Combinatorial Optimization*, 14:481–498 (2007)

[6] Elefteriadou L., Highway capacity, in Kutz M. (ed.) *Handbook of transportation engineering*, New York, McGraw-Hill, chapter 8 (2004)

[7] Garg N., Vazirani V.V., Yannakakis M., Primal-dual approximation algorithms for integral flow and multicut in trees, *Algorithmica*, 18, 3–30 (1997)

[8] Krebs V., Uncloaking terrorist networks, *First Monday*, 7 (2002)

[9] Rosen K., Discrete mathematics and its applications, McGraw-Hill (1999).

[10] Zhou T., Fu Z.-Q., Wang B.-H., Epidemic dynamics on complex networks, *Progress in Natural Science*, 16:452–457 (2006)

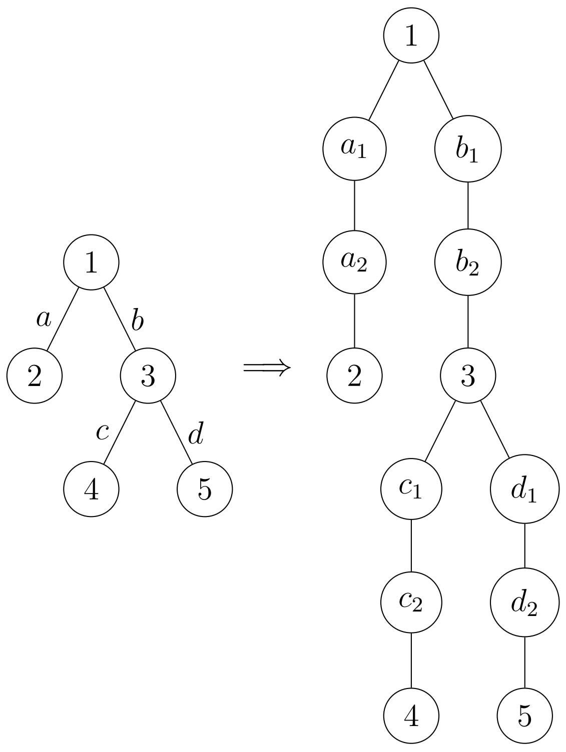
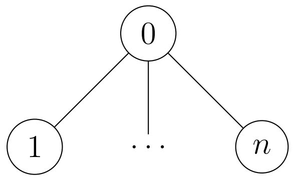


Figure 1: Reduction Unweighted-MCT $\propto$ CNP on trees: insertion of additional nodes. For every edge, we add two additional nodes (e.g., we add nodes a_1, a_2 for edge a). Note that a pair of nodes i, j is disconnected in T by removing, e.g., edge a if and only if the same pair is disconnected in T' by removing one of the two nodes a_1, a_2

$$\text{PARTITION} \quad \begin{cases} \text{Given } A = \{a_1, a_2, \ldots, a_n\},\ \displaystyle\sum_{j=1}^{n} a_j = 2B \\ \quad \text{find } S \subseteq A \text{ such that } \displaystyle\sum_{a_j \in S} a_j = B. \end{cases}$$

CNP



$$\begin{array}{ll} c_{0j} = a_j & j = 1, \ldots, n \\ c_{ij} = 0 & i, j = 1, \ldots, n \\ w_j = a_j & j = 1, \ldots, n \\ w_0 = B + 1 & \\ K = B & \end{array}$$

Figure 2: Reduction PARTITION $\propto$ CNP for a star graph.

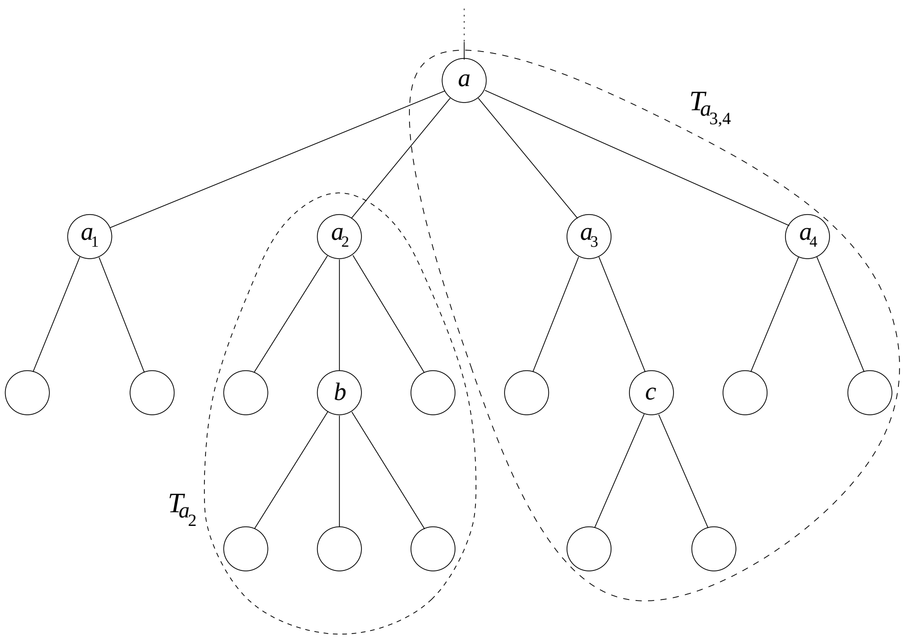


Figure 3: Example of a subtree T_a, where node a has four children (i.e. $s = 4$). The subtrees T_{a_2} and $T_{a_{3,4}}$ are shown.

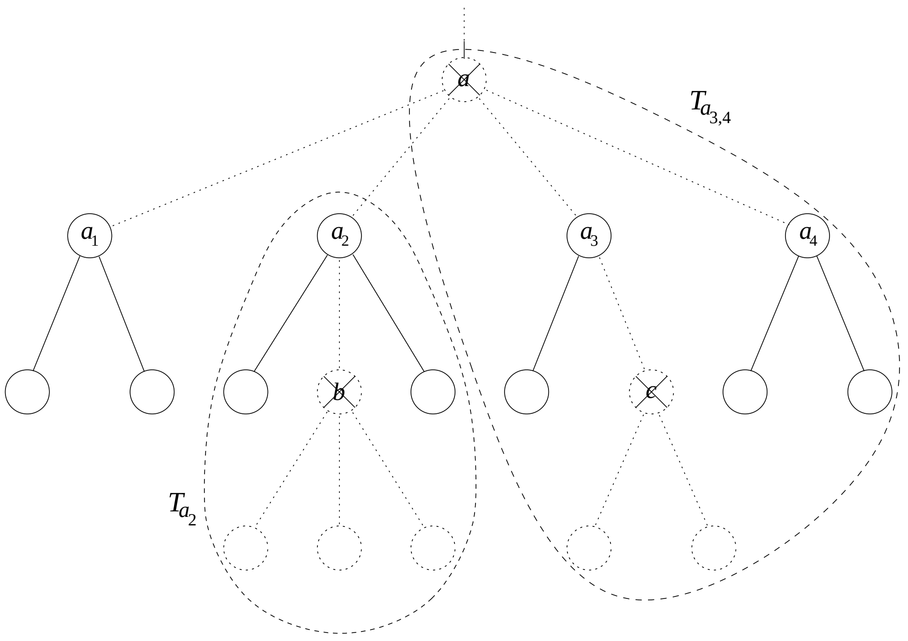


Figure 4: Application of recursion (6) to the subtree of Figure 3 for $m = 0$, $i = 2$ and $k = 3$.

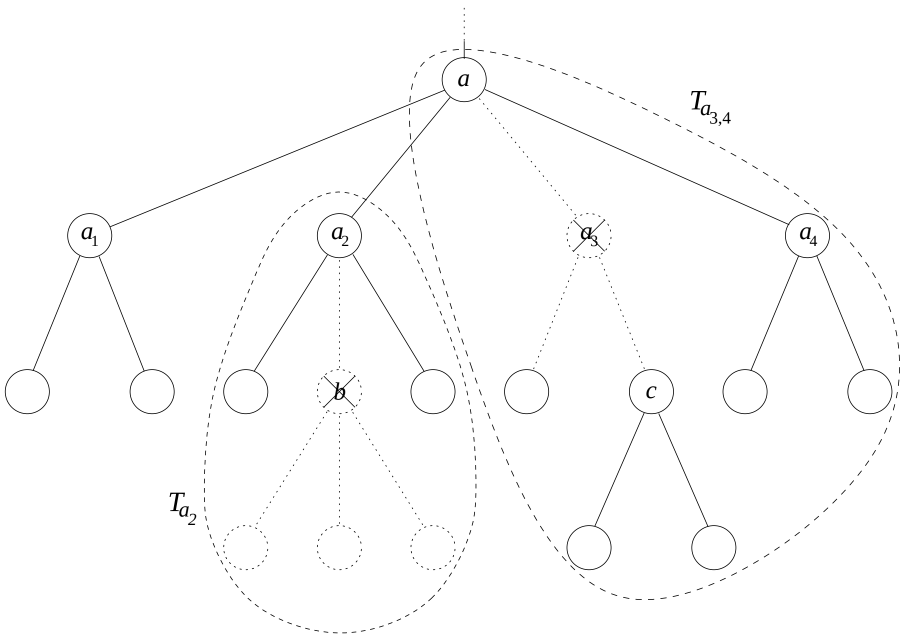
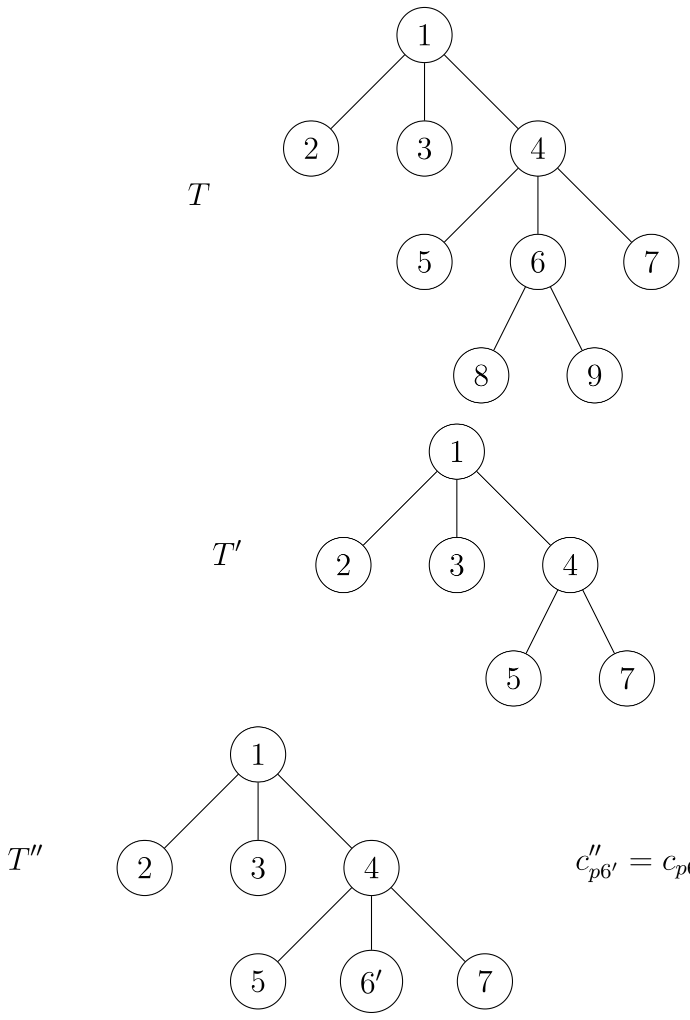


Figure 5: Application of recursion (6) to the subtree of Figure 3 for $m = 7$, $i = 2$ and $k = 2$.

	$\alpha = 0.2$		$\alpha = 0.4$		$\alpha = 0.6$		$\alpha = 0.8$		$\alpha = 1.0$	
n	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$
50	0.0	0.0	0.1	0.1	0.1	0.7	0.1	0.2	0.2	0.2
60	0.1	0.1	0.1	0.1	0.2	0.3	0.3	0.4	0.3	0.5
70	0.1	0.1	0.2	0.3	0.4	0.5	0.7	0.9	0.1	1.1
80	0.2	0.2	0.5	0.6	1.0	1.1	1.6	1.8	2.1	2.7
90	0.3	0.4	0.9	1.1	1.7	2.1	2.9	3.6	3.3	4.5
100	0.5	0.6	1.6	1.8	3.2	3.5	5.4	6.0	6.7	8.8
150	4.0	5.0	11.6	14.7	23.1	29.2	38.4	48.1	52.1	62.8
200	15.4	18.7	47.9	58.6	97.7	119.9	164.3	201	166.0	255.1

Table 1: Performance of DP$_1$ for $c_{ij} = 1$, $w_j = 1$. CPU time in seconds.



$$c''_{p6'} = c_{p6} + c_{p8} + c_{p9}.$$

Figure 6: Equivalent trees obtained when removing node 6 or not removing node 6.

	$K = 10$		$K = 20$		$K = 30$		$K = 40$		$K = 50$	
n	T_{avg}	T_{max}	T_{avg}	T_{max}	T_{avg}	T_{max}	T_{avg}	T_{max}	T_{avg}	T_{max}
100	0.5	0.6	1.6	1.8	3.2	3.5	5.4	6.0	6.7	8.8
150	2.2	3.0	6.0	7.8	11.6	14.7	18.8	23.8	27.7	34.9
200	5.5	6.5	15.4	18.7	29.5	35.8	48.0	58.6	70.6	86.6

Table 2: Performance of DP_1 for several values of K, n. CPU time in seconds.

	$\alpha = 0.2$		$\alpha = 0.4$		$\alpha = 0.6$		$\alpha = 0.8$		$\alpha = 1.0$	
n	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$
50	0.3	0.9	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
60	0.4	0.7	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1
70	1.0	1.6	0.3	0.3	0.2	0.3	0.2	0.2	0.2	0.2
80	2.3	4.5	0.6	0.7	0.4	0.4	0.4	0.4	0.4	0.4
90	3.1	7.1	0.9	1.1	0.7	0.7	0.6	0.6	0.6	0.7
100	5.2	7.8	1.4	1.9	1.1	1.2	1.0	1.0	1.0	1.1

Table 3: Performance of DP_2 for $c_{ij} = 1$, w_j uniformly distributed in $[1, 100]$. CPU time in seconds.

	$\alpha = 0.2$		$\alpha = 0.4$		$\alpha = 0.6$		$\alpha = 0.8$		$\alpha = 1.0$	
n	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$
50	0.9	1.3	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
60	2.2	4.8	0.3	0.4	0.1	0.2	0.1	0.1	0.1	0.1
70	4.3	6.7	0.5	0.8	0.3	0.3	0.2	0.3	0.2	0.2
80	8.0	12.2	1.0	1.5	0.5	0.6	0.2	0.5	0.4	0.4
90	14.2	20.8	1.7	2.3	0.8	0.9	0.6	0.7	0.6	0.7
100	23.6	39.4	2.8	4.2	1.3	1.6	1.1	1.2	1.0	1.1

Table 4: Performance of DP_2 for $c_{ij} = 1$, $w_j = 1$. CPU time in seconds.

	$K = 2$		$K = 3$		$K = 4$		$K = 5$		$K = 6$	
n	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$	T_{avg}	$T_{\max}$
50	11.2	15.9	4.2	6.1	1.8	2.8	0.9	1.3	0.5	0.8
60	46.0	67.7	16.7	29.3	7.1	13.9	3.6	7.6	2.2	4.8
70	139.3	247.9	49.3	90.2	22.6	38.6	11.7	14.6	6.7	9.8
80	383.1	466.1	145.4	191.9	67.3	108.6	36.9	56.8	21.3	31.9
90	871.1	1300.7	353.6	640.6	140.5	125.4	76.5	125.4	44.2	65.1
100	–	–	818.4	1171.4	379.9	627.8	202.1	312.5	123.1	196.7

Table 5: Performance of DP_2 for $c_{ij} = 1$, $w_j = 1$, with very small K. CPU time in seconds.

	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
20	0.0	0.0	0.0	0.0	0.0	0.0										
22	0.0	0.0	0.0	0.0	0.0	0.0										
24	0.0	0.0	0.0	0.0	0.0	0.0	0.0									
26	0.0	0.0	0.0	0.0	0.0	0.0	0.0									
28	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0								
30	0.0	0.0	0.0	0.1	0.1	0.0	0.0	0.0								
32	0.0	0.0	0.0	0.1	0.2	0.0	0.0	0.0	0.0							
34	0.0	0.0	0.0	0.3	0.4	0.2	0.0	0.0	0.0							
36	0.0	0.0	0.1	0.4	0.6	0.7	0.1	0.2	0.0	0.0						
38	0.0	0.0	0.1	0.6	1.8	1.0	0.9	0.1	0.0	0.0						
40	0.0	0.0	0.1	0.9	3.5	4.3	1.6	0.2	0.0	0.0	0.0					
42	0.0	0.0	0.2	1.7	4.2	15.0	5.7	1.3	0.2	0.0	0.0					
44	0.0	0.0	0.2	2.2	10.9	21.6	21.0	6.9	0.6	0.0	0.0	0.0				
46	0.0	0.0	0.3	2.4	12.8	48.8	38.1	17.7	1.2	0.2	0.0	0.0				
48	0.0	0.0	0.4	4.7	20.0	116.9	52.0	49.7	4.5	0.5	0.0	0.0	0.0			
50	0.0	0.0	0.5	7.4	36.2	178.8	140.8	104.9	27.8	2.3	0.1	0.0	0.0			
52	0.0	0.0	0.6	10.4	79.2	269.3	329.5	266.7	124.3	14.5	2.0	0.0	0.0	0.0		
54	0.0	0.0	0.6	11.9	81.0	287.8	1114.4	607.7	475.6	146.0	16.4	0.2	0.0	0.0		
56	0.0	0.0	0.9	15.4	132.8	649.6	1098.8	2114.1	1000.2	105.4	16.5	2.0	0.1	0.0	0.0	
58	0.0	0.0	1.0	25.9	274.5	1213.2	2810.1	3181.9	4188.7	521.1	62.7	24.2	0.2	0.0	0.0	0.0
60	0.0	0.0	1.2	32.2	275.4	1499.6	5479.1	10361.7	7139.1	3613.9	512.2	17.5	1.4	0.1	0.0	0.0

Table 6: Performance of the algorithm of Section 5. Row headings indicate the value of n, column headings indicate the value of K. Average CPU time in seconds.