Solving the Longest Simple Path Problem with Constraint-Based Techniques

Quang Dung PHAM[1], Yves DEVILLE[2]

[1] Hanoi University of Science and Technology
School of Information and Communication Technology, Hanoi, Vietnam
dungpq@soict.hut.edu.vn
[2] Université catholique de Louvain
ICTEAM Institute, B-1348 Louvain-la-Neuve, Belgium
yves.deville@uclouvain.be

Keywords: Constraint Programming, local search, longest path, graph algorithms, comparison of models

Abstract. The longest simple path problem on graphs arises in a variety of context, e.g., information retrieval, VLSI design, robot patrolling. Given an undirected weighted graph $G = (V,E)$, the problem consists of finding the longest simple path (i.e., no vertex occurs more than once) on G. We propose in this paper an exact and a tabu search algorithm for solving this problem. We show that our techniques give competitive results on different kinds of graphs, compared with recent genetic algorithms.

1 Introduction

In this paper we address the problem of finding the longest simple path on a given arbitrary undirected weighted graph. Given an undirected weighted graph $G = (V,E)$, each edge $e \in E$ is associated with a non negative weight $w(e)$. We denote $\mathscr{P}$ the set of simple paths in G (i.e., no vertex occurs more than once). The length (or cost) of a path $p \in \mathscr{P}$ is defined to be the sum of weights of all edges of p: $w(p)$=$\sum_{e \in p} w(e)$. The longest simple path problem (LPP) consists of finding a path $p \in \mathscr{P}$ such that $w(p)$ is maximal. Clearly, LPP is NP-hard because the problem of deciding whether or not there exists a Hamilton cycle in a given graph which is NP-complete can be reduced to this problem [4]. We consider in this paper only simple paths. Henceforth, we use the word "path" instead of "simple path" for short.

Motivation and Related works Wong et al. in [17] mentioned LPP on graphs in the context of information retrieval on peer-to-peer networks where the weights are associated with vertices. They also proposed a genetic algorithm for solving this problem. LPP has also been addressed in [13] for evaluating the worst-case packet delay of Switched Ethernet. The given Switched Ethernet is transformed into a delay computation model represented by a directed graph. On this particular directed acyclic graph, the longest path can be computed using dynamic programming. LPP also appears in the domain of

high-performance printed circuit board design in which one needs to find the longest path between two specified vertices on a rectangle grid routing [16]. In that paper, authors proposed a mixed integer linear programming model for solving LPP in a specific graph. In [11], the authors described LPP in the context of multi-robot patrolling and [10] proposed a genetic algorithm for solving LPP.

In [7], approximating algorithms for LPP (unweighted graph) were considered. The authors showed that no polynomial-time algorithm can find a constant factor approximation for the longest path problem unless **P=NP**. In [5], a heuristic algorithm is proposed for LPP, based on the strongly connected components of the graph.

LPP can also be reduced to the simple shortest path problem (SSPP), by using the same graph, but with edges weight negated. Notice that shortest path algorithms are not appropriate as the cycles of the graph have a negative weight. One should restrict the search to simple paths. The efficient polynomial algorithms for the shortest path problem [1] are thus not applicable for SSPP. The SSPP is often coupled with resource constraints (SSPPRC), such as in vehicle routing. This problem is however usually relaxed by allowing non simple paths, when used in a column generation context. Other methods, such as Lagrangean relaxation, are effective only when the costs are non-negative. An exact method for SSPPRC is however proposed in [3], extending the classical label correcting algorithm.

Constraint programming (CP) has never been directly used for solving LPP. Filtering techniques have been developed for the shortest path problem. In [14, 2], filtering techniques are presented in the context of graph variables. The cost-based filtering techniques do not handle negative cycles.

We consider LPP on arbitrary graphs and propose two constraint-based techniques for solving it: an exact algorithm based on constraint programming technique and a constraint-based local search algorithm. We compare these two techniques with the generic algorithm proposed in [10] on different classes of graphs: graphs extracted from real maps in the simulation package for multi-robot patrolling [11, 12] and ramdom planar graphs. Experimental results show the efficiency of the proposed approaches.

Outline The paper is organized as follows. Sections 2 and 3 respectively describe the proposed exact and tabu search algorithms for solving LPP. Experimental results are given in Section 4. Finally, Section 5 concludes the paper and draws some directions for future work.

2 The exact algorithm

2.1 CP model for solving longest path between two specified vertices

We first propose a CP model for finding the longest (simple) path from a source vertex s to a destination vertex t on a given graph.

Variables For modeling a path p from a vertex s to a vertex t on the given graph $G = (V, E)$, we use an array of variables x in which $x(v)$ represents the successor of v on the path from s to t. The domain of $x(v)$ denoted by $D(x(v))$ is defined to be the set of

adjacent vertices of v in G plus a specific value $\perp$: $D(x(v)) = \{u \in V \mid (u,v) \in E\} \cup \{\perp\}$. If v is not in the path, then $x(v) = \perp$. We use an array y in which $y(v)$ represents the length (i.e., the number of edges) of the subpath of p from s to v (v is in p). If v is not in p then $y(v)$ is undefined.

Constraints The constraints that x, y must satisfy in order to form a path from s to t are the following:

- C1: $x(v) = u \Rightarrow x(u) \neq \perp, \forall v \in V \setminus \{t\}, \forall u \in D(v) \setminus \{t, \perp\}$
- C2: $\sum_{v \in V}(x(v) = u) \leq 1, \forall u \in V \setminus \{s,t\}$
- C3: $\sum_{v \in V}(x(v) = t) = 1$
- C4: $x(v) \neq s, \forall v \in V$
- C5: $x(s) \neq \perp$
- C6: $x(t) = \perp$
- C7: $x(v) \neq \perp \Rightarrow y(x(v)) = y(v) + 1, \forall v \in V$
- C8: $y(s) = 0$

Constraint C1 ensures the continuation of the path until the destination t. It says that a successor v of a vertex u must have its successor if v is not the destination. Constraint C2 specifies that each vertex u of G cannot be the successor of more than one vertex. Constraints C3 and C4 say that the destination vertex must be the successor of some vertex and the source vertex cannot be the successor of any vertices. Constraint C5 says that the source vertex must have a successor and constraint C6 imposes, by convention, $\perp$ the successor of the destination. Constraints C7, C8 eliminate the sub-tour from the path.

The objective function (i.e., the cost of the path) to be maximized is:

$$f(x,y) = \sum_{v \in V \setminus \{t\}} w(v, x(v))$$

where $w(u, \perp) = w(\perp, u) = 0, \forall u \in V$ by convention.

The implementation of the CP algorithm in `COMET` is given in Figure 1. Lines 2-7 initialize the domain `D[u]` for each vertex `u`. Variable `null_value` (line 2) represents the $\perp$ value encoded by 0. Line 8 creates a CP solver `cp`. Decision variables `x` and `y` are initialized in lines 9-10. The objective function to be maximized is stated in line 12. Constraint C1 is stated in lines 14-15. Lines 16-17 state the constraint C2. Lines 18-19, 20-21 respectively state constraints C3, C4, C5 and C6. Constraint C7 is stated in lines 22-23 and line 24 states the constraint C8.

The search is given in lines 26-40. The assignment strategy is described as follows. If the variable `x[v]` is assigned (or instantiated) by the value `u` at the given moment and `u` is not the destination vertex `t` then the next variable to be instantiated will be `x[u]`. Variable `z` keeps the vertex to be instantiated which is initialized by the source vertex `s` in line 26. This strategy ensures the partial assignment is always a partial path from the source vertex `s` to some vertex of the given graph. If `z` is the destination, then the remaining uninstantiated variables will be assigned to $\perp$ (lines 29-31). If the variable `x[z]` is bound to $\perp$, the search backtracks (lines 33-34). Otherwise, we try all vertices `u` of the domain of `x[z]` in the decreasing order of the weights of edges connecting `z` and the vertex `u` for assigning to `x[z]` (line 36-37). The next variable to be assigned will be `x[u]` (see line 38).

```
1   float search(int s, int t){
2     null_value = 0;
3     set{int} D[1..n]();
4     forall(u in 1..n){
5       forall(v in adj[u]) D[u].insert(v);
6       D[u].insert(null_value);
7     }
8     Solver<CP> cp();
9     var<CP>{int} x[u in 1..n](cp,D[u]);
10    var<CP>{int} y[1..n](cp,0..n-1);
11    maximize<cp> % OBJECTIVE FUNCTION
12      sum(v in 1..n)(w[v,x[v]])
13    subject to{ % CONSTRAINTS
14      forall(v in 1..n, u in adj[v]: v != t && u != t)
15        cp.post((x[v] == u) <= (x[u] != null_value));
16      forall(u in 1..n: u != s && u != t)
17        cp.post(sum(v in 1..n)(x[v] == u) <= 1);
18      cp.post(sum(v in 1..n)(x[v] == t) == 1);
19      forall(v in 1..n) cp.post((x[v] != s);
20      cp.post(x[s] != null_value);
21      cp.post(x[t] == null_value);
22      forall(v in 1..n: v != t)
23        cp.post((x[v] != null_value) ~> (y[x[v]] == y[v] + 1));
24      cp.post(y[s] == 0);
25    }using{ % SEARCH
26      Integer z(s);
27      while(!bound(x)){
28        int v = z;
29        if(v == t)
30          forall(u in x.rng(): !x[u].bound())
31            label(x[u],null_value);
32        else
33          if(x[v].getSize()==1 && x[v].getValue()==null_value)
34            cp.fail();
35          else
36            tryall<cp>(u in 1..n: x[v].memberOf(u)) by (-w[v,u]){
37              label(x[v],u);
38              z := u;
39            }
40      }
41    }
42    return sum(v in 1..n)(w[v,x[v]]);
43  }
```

Fig. 1. CP model for finding the longest simple path between two vertices in the given graph

Alternative model using global constraint The proposed model could be simplified by using the global constraint *alldifferent* [6]. The domain of $x(v)$ is $D(x(v)) = \{u \in V \mid (u,v) \in E\} \cup \{v\}$. When v is not in the path, then $x(v) = v$ (the value $\perp$ is not used) and $x(t) = s$ by convention. The constraints are the following:

- C9: $x(v) = u \Rightarrow x(u) \neq u, \forall v,u \in V \setminus \{t\} : v \neq u$
- C10: *alldifferent(x)*
- C11: $x(s) \neq s$
- C12: $x(t) = s$
- C13: $x(v) \neq v \Rightarrow y(x(v)) = y(v) + 1, \forall v \in V \setminus \{t\}$
- C14: $y(s) = 0$

The *alldifferent* global constraint C10 ensures that each vertex v in the solution ($x(v) \neq v$) is the successor of only one vertex. Although this model is simpler and uses a global

constraint, our experiments (presented in Section 4) show that the first model is much more efficient on large instances.

2.2 The exact algorithm

The solution to LPP can be found by iteratively calling the method `search(s,t)` in Figure 1 for all the pairs of source and destination $\langle$`s`, `t`$\rangle$ belonging to the same connected component of the graph, and taking the maximum resulting value.

In many cases, especially in the context of multi-robot patrolling [11], the given graph is sparse and contains many *bridges* (an edge of a graph is called bridge if the removal of this edge from the graph increases the number of connected components of the graph). In these cases, the given graph could be preprocessed by computing its bridges, and its *bridge-blocks* (the connected components of the graph without its bridges). Intuitively, the longest path will be a sequence of paths within different bridge-blocks. It is thus possible to only call the method `search(s,t)` for pairs $\langle$`s`, `t`$\rangle$ belonging to the same bridge-block, and combine these results to obtain the solution.

Before describing our algorithm, we give some definitions and notations. Given a graph H, we denote V_H and E_H respectively the set of vertices and edges of H. Given a set of vertices $S \subseteq V$, we denote $G(S)$ the subgraph induced by S in which $V_{G(S)} = S$ and $(u,v) \in E_{G(S)} \Leftrightarrow (u,v) \in E, \forall u,v \in S$. In a rooted tree T, $T(v)$ denotes the set of descendant vertices of v in T, including v.

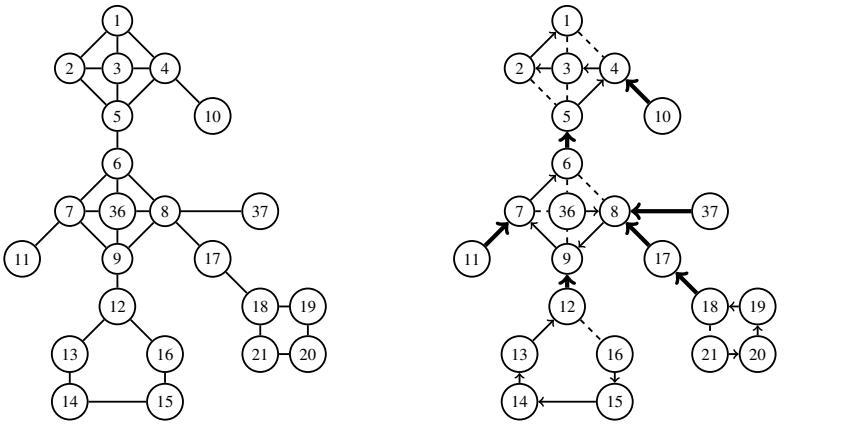
Without loss of generality, we assume that the graph G is connected. The computation of bridges and bridge-blocks can be realized with a variant of the $O(|V|+|E|)$ depth-first search algorithm $DFS(r)$ of [15]. The variant algorithm we developed requires to fix a root vertex r of V_G and produces the following results:

- The set $B(G)$ of bridges of G;
- A spanning tree T of G, rooted at r, in which $f(v)$ is the father of a vertex v (by convention $f(r) = r$); the spanning tree will be composed of k spanning subtrees (one per each bridge-block), connected by the bridges;
- The set $\{S_1, \ldots, S_k\}$, where S_i is the set of vertices of bridge-blocks of G;
- The set R of roots of the different subtrees $T(S_i)$
  $R = \{r_i | r_i \text{ is the root of } T(S_i), 1 \leq i \leq k\}$;
- $Exit(r_i)$, the set of exit vertices of $T(S_i)$, which are vertices in S_i connected to (the root of) some other bridge-blocks in T plus the vertex r_i
  $Exit(r_i) = \{v \in S_i | \exists r_j \in R : f(r_j) = v\} \cup \{r_i\}$;
- $ChRoot(v)$, the children vertices of v in T that are outside the bridge-blocks of v; by definition of bridge blocks, such children are necessarily members of R
  $ChRoot(v) = \{r_i \in R | f(r_i) = v\}$
- $NextRoot(r_i)$ is the set of root vertices of bridge-blocks connected to S_i
  $NextRoot(r_i) = \cup_{v \in S_i} ChRoot(v)$

Figure 2 illustrates the result of $DFS(1)$; we give some of the results: $S_1 = \{1,2,3,4,5\}$, $S_2 = \{6,7,8,9,36\}$, $S_3 = \{17\}$, $S_4 = \{18,19,20,21\}$, $S_5 = \{37\}$, $S_6 = \{12,13,14,15,16\}$, $S_7 = \{11\}$, $S_8 = \{10\}$. $R = \{1,6,17,18,37,12,11,10\}$, $NextRoot(1) = \{6,10\}$, $NextRoot(6) = \{11,12,17,37\}$, $NextRoot(17) = \{18\}$. The other *NextRoot* values are $\emptyset$.

$B(G) = \{(6,5),(10,4),(11,7),(12,9),(17,8),(18,17),(37,8)\}$. $ChRoot(1) = \{\}$, $ChRoot(5) = \{6\}$, $ChRoot(8) = \{17,37\}$, etc. $Exit(1) = \{1,4,5\}$, $Exit(6) = \{6,7,8,9\}$, etc. We now denote:

- $L(r_i)$: the cost of the longest path in $G(T(r_i))$.
- $d(u,v)$: the weight of the longest path between u and v in $G(S_i)$, with $u,v \in S_i$. By convention, $d(u,u) = 0$.
- $H(r_i)$: the cost of the longest path in $G(T(r_i))$ starting from r_i.
- $h(v)$: the cost of the longest path in $G(T(v))$ starting from v, but without edge in $G(S_i)$, with $v \in S_i$.



a. The graph G b. The rooted spanning tree T (solid arcs are in T, bold edges are bridges)

Fig. 2. Illustrating bridges-blocks

These last two quantities can be computed recursively as follows:

$$h(v) = max_{r_i \in ChRoot(v)} H(r_i) + w(v,r_i)$$
$$H(r_i) = max_{v \in S_i} d(r_i,v) + h(v)$$

We have $h(v) = 0$ when $ChRoot(v) = \emptyset$.

Using the above values, we can now compute $L(r_i)$, the cost of the longest path in $G(T(r_i))$, with $r_i \in R$. We have $L(r_i) = max\{L_0(r_i), L_1(r_i), L_2(r_i)\}$, where $L_0(r_i)$ (resp. $L_1(r_i)$, $L_2(r_i)$) is the cost of the longest path in $G(T(r_i))$ containing no (resp. exactly one, at least 2) vertex of S_i. In the computation of $L_1(r_i)$, we should consider the path with exactly one vertex in S_i. Each of these (non empty) paths contains a vertex $v \in S_i$ with a child outside S_i (i.e., $ChRoot(v) \neq \emptyset$). When v has a unique child outside S_i, we should only consider the longest path starting from this child vertex, and extend it with the bridge relating v to this vertex. When v has at least two children outside S_i, we should take the two longest paths starting from these children and relate these paths with the two corresponding bridges. These values can then be computed as follows:

- $L_0(r_i) = max_{r_j \in NextRoot(r_i)} L(r_j)$

– $L_1(r_i) = max_{v \in S_i} l_1(v)$ where

$$l_1(v) = \begin{cases} 0 & \text{if } ChRoot(v) = \emptyset \\ H(r_j) + w(r_j, v) & \text{if } ChRoot(v) = \{r_j\} \\ max_{r_k < r_j \in ChRoot(v)} H(r_k) + w(r_k, v) + w(v, r_j) + H(r_j) & \text{otherwise} \end{cases}$$

– $L_2(r_i) = max_{u \neq v \in S_i} h(u) + d(u, v) + h(v)$

Exploiting lower bounds of the longest path From the above equations, we can deduce that the longest path of the graph either joins two non-exit vertices of the same bridge-block, or the longest path contains bridges and contains at least one exit vertex of every visited bridge-block. This yields a lower bound f^* of the optimal solution. Then, for each pair u, v of non-exit vertices in the same bridge-block, we apply the CP search method $search(u, v, f^*)$ to find the longest path between u and v, using f^* as lower bound. The $search(u, v, f^*)$ method is identical to $search(u, v)$ except that it also posts the constraint that the objective function is bounded by f^*.

The algorithm is composed of the sequential calls to the recursive methods ComputeFirstStep(r) and ComputeSecondStep(r), depicted in Algorithms 1 and 2, where r is the root vertex of the spanning tree T. Algorithm 1 computes the longest path on G containing some exit vertices and bridges. The variable f^* stores the cost of this path. The algorithm 2 completes the search by considering pairs of vertices without exit vertices, using f^* as lower bound. The search is performed only if the cost W of the bridge-block $G(S_i)$ is greater than f^* (see lines 3–4) because the cost of the any longest path on S_i is always less then or equals to W. The data structure $L(r_i)$, $d(u, v)$, $H(r_i)$, $h(v)$ and f^* are global. At the end of the computation, f^* will be the cost of the optimal solution. The algorithm can easily be extended to also return the optimal path.

Two other optimizations have also been considered. The first one is to efficiently compute an upper bound of the cost of the longest path from u to v, using a flow-based linear programming model, in order to avoid an exact computation of the longest path between these two vertices when the upper bound is lower or equal to f^* (f^* is the current lower bound of the solution). The second optimization uses the longest path P between two vertices u and v to compute lower bounds for the longest paths between pairs of vertices appearing in P. None of these optimizations significantly improved the algorithms on the considered benchmarks. They were thus not integrated in the algorithm.

3 The tabu search algorithm

We propose in this section a tabu search algorithm for solving LPP using the constraint-based local search framework `LS(Graph)` [8, 9]. In the local search approach, we maintain a dynamic path on the given graph, i.e., path that can be changed over time. At each step, we change the current path for generating another path with respect to some criteria until the termination condition is reached. In other words, we first define, for each path, a set of neighboring paths. Then, we start the search with some initial

Algorithm 1: ComputeFirstStep(r_i)

```
foreach r_j ∈ NextRoot(r_i) do
    ComputeFirstStep(r_j);
foreach u ∈ Exit(r_i), v ∈ S_i : u ≠ v do
    d(u,v) ← search(u,v); d(v,u) ← d(u,v);
foreach v ∈ S_i do
    h(v) ← max_{r_j ∈ ChRoot(v)} H(r_j) + w(v,r_j);
H(r_i) ← max_{v∈S_i} d(r_i,v) + h(v); L_0 = max_{r_j ∈ NextRoot(r_i)} L(r_j);
L_1 ← 0;
foreach v ∈ S_i : ChRoot(v) ≠ ∅ do
    if ChRoot(v) = {r_j} then l_1 ← H(r_j) + w(r_j,v);
    else
        l_1 ← max_{r_k < r_j ∈ ChRoot(v)} H(r_k) + w(r_k,v) + w(v,r_j) + H(r_j);
    L_1 ← max{L_1, l_1};
L_2 ← max_{u≠v∈S_i} h(u) + d(u,v) + h(v);
L(r_i) ← max{L_0, L_1, L_2};
f* ← max{f*, L(r_i)};
```

Algorithm 2: ComputeSecondStep(r_i)

```
foreach r_j ∈ NextRoot(r_i) do
    ComputeSecondStep(r_j);
W ← cost of G(S_i);
if W > f* then
    foreach u ∈ S_i \ Exit(r_i), v ∈ S_i \ Exit(r_i) : u ≠ v do
        d(u,v) ← search(u,v,f*); f* ← max{f*, d(u,v)};
```

path, and we iteratively replace the current path by an appropriate neighboring path of its neighborhood. The action of replacing a current path by a neighboring path is called a local move.

3.1 The modeling

We apply the modeling approach in [9] in which a specific object encapsulating a rooted spanning tree variable of the given graph and a source vertex variable will be used for representing a path variable and its neighborhood. More precisely, we maintain a object $VarPath(G,tr,s)$ where G is the given graph, tr a rooted spanning tree variable (of G) and s is source vertex variable. The graph G is fixed while tr and s are variables that can be changed over time. At any moment, s is a vertex of G, tr is a rooted spanning tree and $VarPath(G,tr,s)$ induces a unique path in tr from s to the root of tr. We also use $VarPath(G,tr,s)$ to denote this path. Given a path $p = VarPath(G,tr,s)$, the neighborhood of p is generated by changing tr or s. There are four kinds of neighborhoods detailed in [9]:

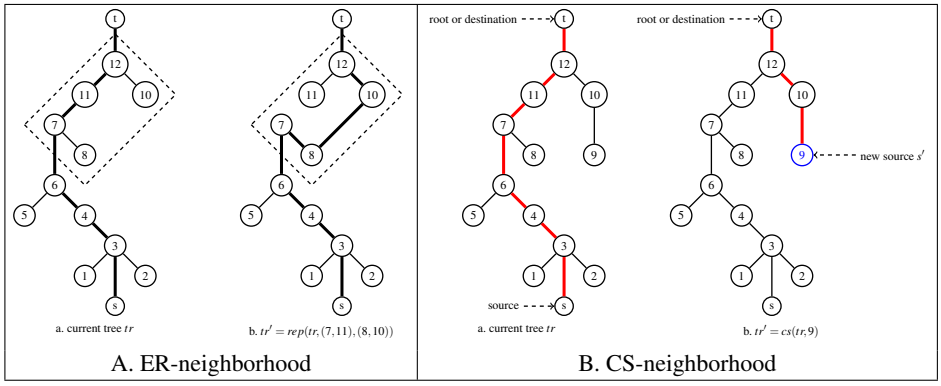


Fig. 3. Illustrating ER-neighborhood and CS-neighborhood

- ER-neighborhood based on edges replacement of tr. Figure 3A illustrates this neighborhood in which the edge (7,11) of the current tree (in Figure 3Aa) is replaced by the edge (8,10), the resulting tree is given in Figure 3Ab. The current path induced from the current tree (in Figure 3a) is $\langle s,3,4,6,7,11,12,t\rangle$ and the neighboring path induced from the neighboring tree (in Figure 3b) is $\langle s,3,4,6,7,8,10,12,t\rangle$.
- CS-neighborhood based on changing the source vertex s. Figure 3B illustrates this neighborhood in which the current source vertex is replaced by the new source vertex 9. The current path induced from the current tree (in Figure 3Ba) is $\langle s,3,4,6,7,-11,12,t\rangle$ and the neighboring path (in Figure 3Bb) is $\langle 9,10,12,t\rangle$.
- CD-neighborhood based on changing the root of tr. Figure 4A illustrates this neighborhood in which the root of tr is replaced by the new root 5. The current path induced from the current tree (in Figure 4Aa) is $\langle s,3,4,6,7,11,12,t\rangle$ and the neighboring path (in Figure 4Ab) is $\langle s,3,4,6,5\rangle$.
- The fourth neighborhood exploited in this tabu search is the general neighborhood $\mathcal{N}_2$ (see [9] for details). The main idea of this neighborhood is the following. Given a rooted spanning tree tr which induces the path p, the neighborhood $\mathcal{N}_2(tr)$ is the set of rooted spanning tree tr' which induce the path p' and the path p' can be created by replacing a subpath of p by a new path having exactly 2 edges. Figure 4B illustrates this neighborhood in which the current rooted spanning tree tr induces the current path $p = \langle s,3,4,6,7,11,12,t\rangle$ and on right hand side is a neighboring rooted spanning tree which induces the path $p' = \langle s,3,4,8,11,12,t\rangle$ where p' is created by replacing the subpath $\langle 4,6,7,11\rangle$ of p by the new path $\langle 4,8,11\rangle$ having 2 edges. In practice, the local move in this neighborhood is performed as follows. Given a current rooted spanning tree tr which induces the current path p, the neighboring path p' is obtained by replacing a subpath of p by a new path $\langle v,x,u\rangle$ with $x \notin p, u,v \in p$ and $(v,x),(x,u) \in E$, the spanning tree tr' is generated randomly such that it induces the path p'.

The three first neighborhoods feature the diversification of the tabu search where the neighboring solutions are quite different from the current solution, while the fourth neighborhood allows the search to explore solutions close to the current solution.

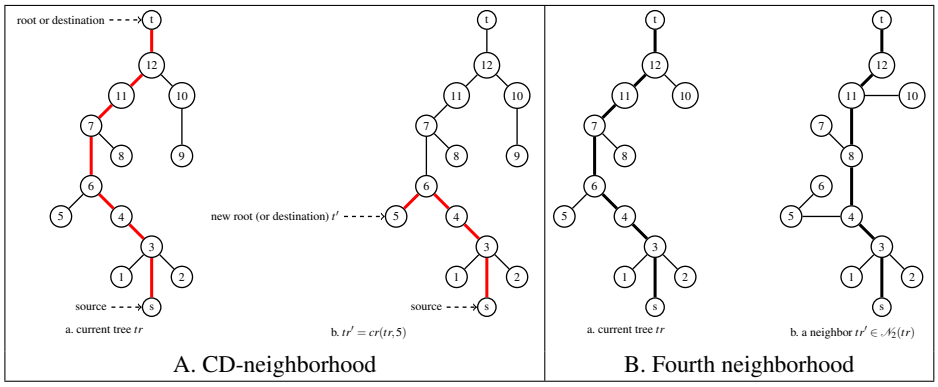


Fig. 4. Illustrating CD-neighborhood and fourth neighborhood

Our tabu search uses *short-term memory* to escape local minima and to avoid revisiting solutions. The tabu lists store attributes of the local moves instead of the whole solutions. The check whether a solution p is tabu is replaced by the check whether the attributes of the local move from the current solution to p is tabu. The first tabu list consists of two lists for storing two edges (an edge to be removed and an edge to be added) of the local move in the first neighborhood. The second tabu list stores the new selected source of the local move in the CS-neighborhood and the third tabu list stores the new selected root of the local move in the CD-neighborhood. The fourth tabu list stores the new introduced vertex (e.g., vertex 8 in Figure 4Bb) in the fourth neighborhood.

The tabu search algorithm depicted in Algorithm 3 explores the four neighborhoods described above together with a restart schema. Variable s (line 1) represents the current solution. Function f is the cost function of the paths. S^* (line 1) stores the best solution found so far and the restart-best solution s^* stores the best solution found from each restart. The restart is performed (lines 3–4) when the restart-best solution is not improved for a given *maxStable* continuous iterations. Variable *nic* counts the number of continuous iterations in which the restart-best solution is not improved. *nic* increases by one when the current solution is not better than s^* (see lines 14–15). Otherwise, it is set by one (lines 12-13). The initial solution is generated (line 1) using the schema for generating a set of paths of the initial population in the genetic algorithm of [10]. Lines 10–11 update the best solution (if any).

The core of the tabu search is the local move. Lines 6–7 explore four kinds of neighborhood ($\tau_1, \tau_2, \tau_3, \tau_4$ are four corresponding tabu lists). For each neighborhood, it chooses a first improving neighbor in order to avoid computation overhead. The tabu search then selects the best one among the four selected neighbors (line 8). Line 9 stores attributes of the local move taken into the corresponding tabu list.

The procedure of selecting a first improving neighbor is described in Algorithm 4. All accepted neighbors, i.e., neighbor that is not tabu or is better than the best solution found so far S^*, are scanned (lines 2–3). If all the accepted neighbors are not better than the current solution, then the algorithm selects the best one. Otherwise, the first

neighbor that is better than the current solution is selected and the algorithm terminates (lines 6–7). The tabu search is implemented using the `LS(Graph)` framework [9].

Algorithm 3: TabuSearch

```
1  s ← generateInitialSolution(); S* ← s; s* ← s; nic ← 1;
2  while getTime() < timeLimit do
3      if nic < maxStable then
4          s ← generateInitialSolution(); s* ← s; nic ← 1;
5      else
6          foreach i ∈ 1..4 do
7              s_i ← selectFirstImprovingNeighbor(N_i, s, S*, τ_i);
8          s ← argMax_{i=1}^{4} f(s_i);
9          τ is the tabu list corresponding to s; τ ← τ :: s;
10         if f(s) > f(S*) then
11             S* ← s;
12         if f(s) > f(s*) then
13             s* ← s; nic ← 1;
14         else
15             nic ← nic + 1;
16 return S*;
```

4 Experiments

We compare our exact and tabu search algorithms with the genetic algorithms (GA) of [10]. In that paper, four versions GANP, GAIP, GABPP, and GAMM of the genetic algorithm were tested on only one sparse graph. We re-implemented all these four versions in the `COMET` programming language. The best version will be used for comparing with our exact and tabu search algorithms.

Instances We experiments the algorithms on three classes of instances. The first instances are taken from the Multi-Robot patrolling simulation package [12]. These graphs were extracted from real maps with a specialized image processing package. They are sparse graphs and most of them are tree-like graphs. The second class instances are random planar graphs in which each graph has many small-size bridge-blocks (the number of vertices of each bridge-block is 10). The third class of instances are random planar graphs where each graph has only one bridge-block. All the instances are available (becool.info.ucl.ac.be).

Settings The length of the tabu list is set to 50 and the *maxStable* is set to 50. The tabu search and the genetic algorithms GA are executed 20 times for each instance. The time

Algorithm 4: selectFirstImprovingNeighbor(N, s, S^*, τ)

Input: N is a neighborhood, s is the current solution, S^* is the best solution found so far, and τ is the corresponding tabu list of N

Output: A first improving neighbor (if any) or the best non-tabu neighbor if no improving neighbor exists

```
1 eval ← −∞;
2 foreach s_i ∈ N(s) do
3     if ¬isTabu(s_i, τ) ∨ f(s_i) > f(S*) then
4         if f(s_i) > eval then
5             eval ← f(s_i); s̄ ← s_i;
6             if f(s_i) > f(s) then
7                 break;
8 return s̄;
```

limit for each execution is 30 minutes. In the genetic algorithm, the maximal number[3] of solutions maintained in each population is set to 200 and the percentage of the best solution which are saved for the next generation is set to 5%. The experiments were performed on XEN virtual machines with 1 core of a CPU Intel Core2 Quad Q6600 @2.40GHz and 1GB of RAM.

Results The results are presented in Table 1. Columns 2-3 are the number of vertices and edges of the given graph. Column 4 presents the optimal objective values which are found by exact algorithms. Columns 5-7 respectively show the execution times of our exact algorithms without exploiting lower bounds, exploiting lower bounds and of our alternative model using the *alldifferent* constraint (with lower bound exploited). Columns 8-11 respectively present the average, the minimal, the maximal of the objective values of the best solution found, and the average of time for finding the best solution in 20 executions of our tabu search algorithm for each instance. The same information of the genetic algorithm is presented in columns 12-15. All the times are reported in seconds.

All exact algorithms find the optimal solutions in the two first classes of instances. The use of lower bounds is especially effective in the second class of instances, with a speedup around 3. Notice the spectacular speedup on the instance Grelha-4 in the first class of instances. We observe that exploiting lower bound yields significant improvement (see comparison between columns 5 and 6). Moreover, the alternative model using the *alldifferent* global constraint does not improve the performance. It is even much less efficient on the second class of instances. The global constraint might give more pruning than the first CP model but the pruning is time-consuming.

On the first class of instances, the tabu search and genetic algorithm find optimal solutions in all executions but they cannot prove the optimality. In average, our tabu search

[3] In some small instances, the total number of distinct paths of the given graph cannot reach 200.

Instances	$\sharp V$	$\sharp E$	Exact algorithms				tabu search				GA[10]			
			f^*	without LB t(s)	with LB t(s)	LB-alldiff t(s)	f_avg	f_min	f_max	t (s)	f_avg	f_min	f_max	t (s)
1r-5-map	12	11	443	0.32	0.39	0.32	443	443	443	0.00	443	443	443	0.09
Cumberland_2ndfloor	66	66	1272	0.38	0.39	0.38	1272	1272	1272	0.12	1272	1272	1272	0.65
Grelha4	25	40	1824	603.69	42.82	58.85	1824	1824	1824	4.33	1824	1824	1824	0.14
MIT_Infinite_Corridor30	121	126	3282	22.30	21.77	26.95	3282	3282	3282	166.36	3282	3282	3282	56.27
Maze	20	19	3469	0.36	0.33	0.40	3469	3469	3469	0.00	3469	3469	3469	0.09
broughton	135	135	1556	0.36	0.40	0.39	1556	1556	1556	0.09	1556	1556	1556	4.90
example	32	37	1092	3.12	2.58	2.98	1092	1092	1092	6.49	1092	1092	1092	1.36
fire	74	73	1060	0.34	0.47	0.44	1060	1060	1060	0.04	1060	1060	1060	0.82
forth_gridok	70	70	952	0.41	0.47	0.43	952	952	952	0.04	952	952	952	1.04
gates_first_floor	206	209	1318	12.65	15.57	15.70	1318	1318	1318	19.90	1318	1318	1318	41.35
hamilton_map	12	20	1580	1.37	1.10	0.94	1580	1580	1580	1.52	1580	1580	1580	2.09
isr_floor0	59	59	3181	3.94	4.30	3.76	3181	3181	3181	8.28	3181	3181	3181	0.90
isr_floor1	78	77	1810	0.30	0.41	0.36	1810	1810	1810	0.05	1810	1810	1810	0.99
largeexp3	41	40	1244	0.34	0.33	0.34	1244	1244	1244	0.01	1244	1244	1244	0.16
map	268	270	1273	1.48	1.58	1.70	1273	1273	1273	33.60	1273	1273	1273	12.90
quad	56	58	1036	0.47	0.48	0.47	1036	1036	1036	0.03	1036	1036	1036	0.25
strongly_connected	45	58	2166	1.28	0.76	0.82	2166	2166	2166	48.60	2166	2166	2166	3.28
planar-n100-m216	100	216	189.97	12.27	4.00	5.86	187.99	186.76	189.97	1050.45	188.41	185.93	189.97	1001.98
planar-n200-m434	200	434	447.00	60.48	21.32	35.76	431.89	426.05	438.60	921.83	404.10	377.84	427.01	1244.38
planar-n300-m655	300	655	392.49	167.91	54.73	96.48	376.24	371.03	383.49	813.98	347.40	331.20	364.35	1226.88
planar-n400-m870	400	870	510.19	363.32	121.15	210.23	481.37	452.14	500.72	1027.40	440.31	417.58	468.41	1119.04
planar-n500-m1089	500	1089	695.84	672.55	235.73	411.10	658.73	615.69	683.40	821.02	563.53	505.64	600.73	1359.98
planar-n600-m1301	600	1301	511.93	1130.77	364.77	619.12	473.37	445.18	490.29	837.44	430.11	402.50	464.16	1143.41
planar-n700-m1526	700	1526	512.32	1723.34	569.15	971.69	472.03	432.58	495.03	1267.33	442.70	423.89	459.58	1164.36
planar-n800-m1747	800	1747	655.38	2519.05	825.12	1530.89	594.08	540.55	634.26	781.01	494.55	467.67	536.96	1225.81
planar-n900-m1959	900	1959	670.13	3578.66	1161.53	2102.26	605.10	517.06	648.43	1175.67	527.84	446.74	567.88	1387.18
planar-n1000-m2177	1000	2177	598.42	4848.99	1649.42	2877.46	545.54	482.65	577.60	1153.83	490.19	439.78	530.98	1289.74
planar-n100-m285	100	285	-	-	-	-	1528.36	1508.55	1554.40	979.92	1415.89	1347.40	1489.86	873.10
planar-n150-m432	150	432	-	-	-	-	2560.63	2537.62	2605.69	894.29	2292.77	2165.32	2411.80	1375.68
planar-n200-m583	200	583	-	-	-	-	3952.31	3902.61	4017.22	1200.74	3384.08	3223.69	3565.75	1223.30
planar-n250-m731	250	731	-	-	-	-	5383.19	5334.40	5544.25	1252.84	4620.66	4314.07	5006.28	1355.14
planar-n300-m880	300	880	-	-	-	-	12563.32	12270.60	12905.10	823.21	8667.38	7432.96	9814.50	1194.17
planar-n350-m1031	350	1031	-	-	-	-	8542.16	8462.25	8645.68	1117.53	6892.87	6509.63	7237.82	1522.40
planar-n400-m1182	400	1182	-	-	-	-	10433.72	10179.30	10697.10	1215.00	8079.01	7598.70	8669.67	1469.28
planar-n450-m1329	450	1329	-	-	-	-	12359.00	12118.20	12632.20	1178.73	9291.49	8719.86	10117.80	1329.50
planar-n500-m1477	500	1477	-	-	-	-	13816.64	13574.80	14173.00	1076.90	10133.63	9384.45	10881.20	1304.58

Table 1. Experimental results

algorithm finds optimal solutions faster than the genetic algorithm for 11 of the 17 instances. Our exact algorithm finds optimal solutions and proves their optimality very fast (in less than 5 seconds) except the instances Grelha4, MIT_Infinite_Corridor30, and gates_first_floor. The reason is that in these three instances, the bridge-blocks are large. Thus, the number of pairs source-destination over which the CP algorithm (see Figure 1) performs is high. This makes the computation time of the exact algorithm relatively high on these instances.

On the second class of instances (random planar graphs with different bridge-blocks), our tabu search algorithm is always better than the genetic algorithm. The tabu search and genetic algorithms do not find optimal solutions while our exact algorithm is able to find and prove such optimal solutions. More over, in all instances except the last one (graph with 1000 vertices and 2177 edges), the execution time of the exact algorithm is better than the average execution time of the tabu search and of the genetic algorithms.

The results of the third class of instances (random planar graphs with a single bridge-block) are presented at the bottom of Table 1. These instances are not adapted to the exact algorithm as bridge-blocks cannot be exploited. The exact algorithms cannot terminate within 30 minutes. The tabu search gives much better results than the genetic algorithm. On all the instances, the worst solution found by the tabu search in 20 executions is always better than the best solution found by the genetic algorithm. The genetic algorithm converges early and did not escape the low-quality local optimum. The average execution time of our tabu search algorithm is better than the average execution time of the genetic algorithm in all instances except the first one (graph with 100 vertices and 285 edges).

In summary, our exact algorithm is able to find optimal solutions on large graphs composed of many bridge-blocks whereas local search algorithms cannot find the optimal solution. The exact algorithm is not applicable on large graphs with a single bridge-block. Our tabu search algorithm is shown to be better than the state of the art genetic algorithm [10].

5 Conclusion

In this paper, we proposed an exact algorithm, based on Constraint Programming (CP), and a tabu search algorithm, using constraint-based local search (CBLS), for solving the longest simple path problem (LPP), an NP-hard problem. The exact algorithm exploits the bridge-blocks of the graph. The local search algorithm is constructed on top of our LS(Graph) CBLS framework [9].

Experimental results showed that the exact algorithm is relevant for small graphs as well as for large graphs with different bridge-blocks. Exploiting the lower bounds improves the algorithm. Our tabu search algorithm finds optimal solutions only on small instances, but without proof of optimality. The tabu search algorithm is particularly well adapted on large instances with a small number of bridge-blocks, where our exact algorithm is not adapted. Except on some specific small instances, our tabu search algorithm is always better than the state of the art genetic algorithm [10], both in quality of solutions than in time efficiency.

As future work, we would like to develop specific search strategies for our exact algorithm in order to improve its efficiency on graphs with few bridge-blocks. We will study how to handle bridge-blocks dynamically. The bridge-blocks will not only be computed on the initial graph, but will also be computed incrementally on the graph induced by the domain of the variables. We also intend to develop hybrid techniques combining CP, integer programming and local search.

Acknowledgments We would like to thank the reviewers for their helpful comments and suggestions. This research is partially supported by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy) and the FRFC project 2.4504.10 of the Belgian FNRS (National Fund for Scientific Research). We also thank David Portugal for his support about some instances from the simulation package [12].

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network flows: theory, algorithms, and applications. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)
2. Dooms, G.: The CP(Graph) Computation Domains in Constraint Programming. Ph.D. thesis, UCLouvain, Belgium (2006)
3. Feillet, D., Dejax, P., Gendreau, M., Gueguen, C.: An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. Networks pp. 216–229 (2004)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability: A guide to the Theory of NP-Completeness. W. H. Freeman, 1st ed. (1979)
5. Gondran, M., Minoux, M.: Graphes et algorithmes, 3e édition revue et augmentée. Collection de la Direction des Études et Recherches d'Électricité de France, Volume 37, Eyrolles (1995)
6. van Hoeve, W.J.: The alldifferent constraint: A survey. In: Sixth Annual Workshop of the ERCIM Working Group on Constraints (2001)
7. Karger, D., Motwani, R., Ramkumar, G.: On approximating the longest path in a graph. Algorithmica 18, 421–432 (1993)
8. Pham, Q.D., Deville, Y., Van Hentenryck, P.: Constraint-based local search for constrained optimum paths problems. In: Proceedings of CPAIOR'2010. pp. 267–281 (2010)
9. Pham, Q.D.: LS(Graph): A constraint-based local search framework for constrained optimum tree and path problems on graphs. Ph.D. thesis, UCLouvain, Belgium (2011)
10. Portugal, D., Antunes, C.H., Rocha, R.: A study of genetic algorithms for approximating the longest path in generic graphs. In: Proceedings of SMC 2010. pp. 2539–2544 (2010)
11. Portugal, D., Rocha, R.: Msp algorithm: Multi-robot patrolling based on territory allocation using balanced graph partitioning. In: Proceedings of SAC 2010. pp. 1271–1276 (2010)
12. Portugal, D.: Multi-robot patrolling simulation package. http://paloma.isr.uc.pt/ davidbsp/
13. Schmidt, K., Schmidt, E.G.: A longest-path problem for evaluating the worst-case packet delay of switched ethernet. In: Proceedings of SIES'2010. pp. 205–208 (2010)
14. Sellmann, M., Gellermann, T., Wright, R.: Cost-based filtering for shorter path constraints. In: Proceedings CP'03. pp. 694–708 (2003)
15. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (1972)
16. Tseng, I.L., Chen, H.W., Lee, C.I.: Obstacle-aware longest-path routing with parallel milp solvers. In: Proceedings of WCECS-ICCS, Vol. 2. pp. 827–831 (2010)
17. Wong, W.Y., Lau, T.P., King, I.: Information retrieval in p2p networks using genetic algorithm. In: Proceedings of the 14th Int. World Wide Web Conference, Special interest tracks and posters. pp. 922–923 (2005)