CEJOR manuscript No.
(will be inserted by the editor)

Algorithmic upper bounds for graph geodetic number

Ahmad T. Anaqreh · **Boglárka G.-Tóth** ·
Tamás Vinkó

the date of receipt and acceptance should be inserted later

Abstract Graph theoretical problems based on shortest paths are at the core of research due to their theoretical importance and applicability. This paper deals with the geodetic number which is a global measure for simple connected graphs and it belongs to the path covering problems: what is the minimal-cardinality set of vertices, such that all shortest paths between its elements cover every vertex of the graph. Inspired by the exact 0-1 integer linear programming formalism from the recent literature, we propose new method to obtain upper bounds for the geodetic number in an algorithmic way. The efficiency of these algorithms are demonstrated on a collection of structurally different graphs.

Keywords geodetic number · integer linear programming · upper bound · greedy heuristic

1 Introduction

Path covering problems play an important role both in theory and applications mostly by reasons of their straight interpretability. Complex notions of cover are possible, one of the most relevant set of such problems is involving shortest paths. The graph geodetic number, which belongs to this problem set, was introduced in [9]. There might be several application fields for the problem, perhaps the most straight-forward one is given in [10] which poses it as a social network problem. It turned out that calculating the geodetic number is an NP-hard problem for general graphs [3]. As for many similar graph theoretical problems, an integer linear programming (ILP) formulation is possible and such a model was given in a recent paper by Hansen and van Omme [8], containing also the first computational experiments on a set of random graphs of moderate size. Motivated by these results, this paper empirically investigates upper bound algorithms, which, according to our experiments provide results

Ahmad T. Anaqreh, Boglárka G.-Tóth, Tamás Vinkó
Institute of Informatics, University of Szeged, Hungary
E-mail: {ahmad, boglarka, tvinko}@inf.u-szeged.hu

of small gap on the same set of random graphs using relatively low computation time even on graphs with 150 nodes, as well as on real-world graphs of large scale.

In the rest of this section the definition of the geodetic number problem is given, followed by the 0-1 linear programming formalism from [8]. Then, in Section 2 the algorithmic descriptions of the two proposed upper bound procedures are given. Finally, we report our extensive computational experiments in Section 3 to demonstrate the efficiency of these algorithms.

Problem description. A simple connected graph is denoted by $G = (V,E)$, where V is the set of vertices and E is the set of edges. Assume that $n = |V|$ and $m = |E|$. Given $i, j \in V$, the set $I[i, j]$ contains all $k \in V$ which lies on any shortest paths (*geodetics*) between i and j. The union of all $I[i, j]$ for all $i, j \in S \subseteq V$ is denoted by $I[S]$, which is called *geodetic closure* of $S \subseteq V$. Formally

$$I[S] := \{k \in V : \exists i, j \in S, k \in I[i, j]\}.$$

The *geodetic set* is a set S for which $V = I[S]$. The *geodetic number* of G is

$$g(G) := \min\{|S| : S \subseteq V \text{ and } I[S] = V\}.$$

A 0-1 integer linear programming model. In [8] a binary integer linear programming model has been proposed which is as follows.We denote the length of the shortest path between u and v by $d(u,v)$ for every $u,v \in V$. For each node $k \in V$ define the set

$$P_k := \{(i, j) \in V \times V \mid d(i,k) + d(k, j) = d(i, j)\}.$$

Set P_k hence contains all pairs of vertices for which a shortest path is going through node k.

The 0-1 LP model is

$$\min \sum_{k=1}^{n} x_k \tag{1}$$

subject to

$$1 - x_k \leq \sum_{(i,j) \in P_k} y_{ij} \qquad \forall\, k, i, j \in V, i < j \tag{2}$$

$$y_{ij} \leq x_i \qquad \forall\, i, j \in V, i < j \tag{3}$$

$$y_{ij} \leq x_j \qquad \forall\, i, j \in V, i < j \tag{4}$$

$$x_i + x_j - 1 \leq y_{ij} \qquad \forall\, i, j \in V, i < j \tag{5}$$

and the variables are all binary

$$x_i \in \{0,1\} \quad \forall i \in V$$
$$y_{ij} \in \{0,1\} \quad \forall i, j \in V, i < j$$

The value of variable x_k indicates if vertex k belongs to the set S or not. Thus, the sum of these binary variables needs to be minimized. The auxiliary variables y_{ij} denote the bilinear term $x_i x_j$, so (2) means x_k can be 0 only if there are $i, j \in S$ such that k is on their shortest path. McCormick conditions (3)−(5) describe the $y_{ij} = x_i x_j$ correspondence.

2 Upper bounds

The trivial upper bound for the geodetic number is $g(G) \leq n$, which is tight for the complete graph. Chartrand *et al.* [5] prove that $g(G) \leq n-d+1$, where d is the length of the longest shortest path (called *diameter*) of G. Other upper bounds are also given [4, 11, 12] but these are concerning specific graph structures.

In this section we aim at deriving upper bounds for general graphs in algorithmic ways. Two upper bound algorithms are given. The first one uses Floyd's algorithm, while the second algorithm is based on Dijkstra's algorithm.

2.1 Greedy algorithm

The main idea of the algorithm is to generate a geodetic set in a greedy fashion: in every step choose that vertex i to be included in S which makes the largest increase in $I[S]$. Namely, where the cardinality $\mid I[S \cup \{i\}] \setminus I[S] \mid$ is maximal.

Initialization. The initialization part is shown in Algorithm 1. In the lines $4-5$ all-pair shortest paths are calculated, which are then used in lines $6-8$ where the sets I_{ij} are defined. At this point these sets are essentially the same as $I[i,j]$ which got defined already in Section 1, i.e., containing all nodes which lie on any shortest paths between nodes i and j. We use different notation here on purpose, as the sets I_{ij} are subject to change later in our algorithm. In line 9 the nodes with degree at most one are put into (the initially empty) geodetic set S. The degree-one node is the node connected by one edge in the graph, these nodes must be part of geodetic set S based on the fact that degree-one nodes are always part of the geodetic set, see [9].

The geodetic closure $I[S]$ gets initialized for S in lines $10-11$. Note that this set could be empty here. Finally, in lines $12-13$ all of the already covered vertices from the sets I_{ij} are removed.

Algorithm 1: Greedy algorithm - Initialization

1 **Function** `GreedyInit`
2 $S=\emptyset, I[S]=\emptyset, I_{ij}=\emptyset \quad \forall i,j \in V$
3 $d_{ij}=1 \quad \forall (i,j) \in E,\ d_{ij}=\infty \quad \forall (i,j) \notin E$
4 **for** $\forall k \in V, i \in V, j \in V$ **do** `// distance calculation`
5 $\quad d_{ij}=\min\{d_{ij}, d_{ik}+d_{jk}\}$ `// by Floyd algorithm`
6 **for** $\forall i \in V, j \in V, k \in V$ **do** `// calculation of` I_{ij}
7 $\quad$ **if** $d_{ij}=d_{ik}+d_{kj}$ **then**
8 $\quad\quad I_{ij}=I_{ij} \cup \{k\}$
9 $S=\{k \in V \mid \deg(k) \leq 1\}$ `// unreachable nodes from all paths must be in S`
10 **for** $\forall i \in S, j \in S$ **do** `// build I[S] for S`
11 $\quad I[S]=I[S] \cup I_{ij}$
12 **for** $\forall i \in V, j \in V$ **do** `// Update` I_{ij}`-s by removing`
13 $\quad I_{ij}=I_{ij} \setminus I[S]$ `// all covered vertices`

Algorithm 2: Greedy algorithm - Auxiliary functions

14 **Function** `LargestIncrease`(V,S,I)
15   **for** $\forall i \in V \setminus S$ **do** `// compute` $I_i[S]$`, the set increasing`
16     $I_i[S] = \emptyset$ `//` $I[S]$ `if` i `is included`
17   **for** $\forall j \in S$ **do**
18       $I_i[S] = I_i[S] \cup I_{ij}$
19   $\ell = \underset{i \in V \setminus S}{\operatorname{argmax}} |I_i[S]|$ `// find the node for which` $I[S]$ `would grow most`
20   **return** $\ell, I_\ell[S]$

21 **Function** `LargestIncreasePair`(V,S,I)
22   **for** $\forall i \in V \setminus S, j \in V \setminus S$ **do** `// compute` $I_{ij}[S]$`, the set increasing`
23     $I_{ij}[S] = I_{ij} \cup I_i[S] \cup I_j[S]$ `//` $I[S]$ `if` i,j `are included`
24   $(k,h) = \underset{i,j \in V \setminus S}{\operatorname{argmax}} |I_{ij}[S]|$ `// find pair of nodes for which` $I[S]$ `would grow most`
25   **return** $k, h, I_{kh}[S]$

Auxiliary functions. The description of our greedy method continues in Algorithm 2, where the functions `LargestIncrease` and `LargestIncreasePair` are defined. These compute the vertex (and vertex pair, respectively) which increases most the covered set $I[S]$ if it is included in S. Two notations are used: the sets $I_i[S]$ contain vertices which are covered if node i was included in set S, and sets $I_{ij}[S]$ contain vertices which are covered if both i and j nodes were included in set S. The sets $I_i[S]$ are initialized as empty in line 16. In lines $15-18$ the sets $I_i[S]$ are constructed by the nodes currently in I_{ij}, where i is not an element of S and j is in S. Then, in line 19 we define ℓ to be the node from $V \setminus S$ for which $I_i[S]$ is the largest. Similarly, in the function `LargestIncreasePair(V,S,I)`, at lines $22-23$ the sets $I_{ij}[S]$ are calculated with nodes which would make $I[S]$ increasing if the pair (i,j) were included. Finally, in line 24 such a pair of nodes is selected for which the set $I_{ij}[S]$ is the largest.

Main algorithm. The main loop of our greedy approach is shown in Algorithm 3. The condition in line 29 checks if there is a node which is not covered yet. In lines $30-35$ a heuristic rule is applied, which simply checks if the size of set $I_\ell[S]$ is at least half the size of set $I_{kh}[S]$. This is a greedy choice,however, other conditions could also be applied here to decide whether one node or a pair of nodes should be added to S. After this, in lines $36-37$ the sets I_{ij} get updated by removing all the covered vertices from them. In line 38 we need to choose again the node for which the increase of $I[S]$ is the largest. Finally, the execution of lines $39-42$ depend on the parameter *AddOne*. If it is set as $AddOne = 0$, then the algorithm chooses the best pair of nodes to be added to the set S by calling the `LargestIncreasePair` function. Otherwise, i.e., in case of $AddOne = 1$, this function call is simply skipped, the settings done in line 40 are needed for keeping the consistency. In the numerical experiments reported in Section 3 this simplified version will be referred as `AddOne`.

Note that due to the fact that the input graph G is undirected, similarly to the 0-1 LP model description in Section 1, one can assume the condition $i < j$ for all appropriate cases. This is done in the actual implementation of the greedy algorithm to make it faster, but we omit these details for the easier understanding of the pseudocodes.

Algorithm 3: Greedy algorithm - Main

```
GreedyInit                                         // initialize I_ij sets and S
[ℓ, I_ℓ[S]] = LargestIncrease(V,S,I)                 // ℓ would make I[S] grow most
[k,h,I_kh[S]] = LargestIncreasePair(V,S,I)     // k and h would make I[S] grow most
while |I_ℓ[S]| + |I_kh[S]| > 0 do                   // the set is not geodetic yet
    if |I_ℓ[S]| > |I_kh[S]|/2 then       // balance adding one or two vertices to S
        S = S ∪ l
        I[S] = I[S] ∪ I_ℓ[S]                                        // update I[S]
    else
        S = S ∪ {k,h}
        I[S] = I[S] ∪ I_kh[S]                                       // update I[S]
    for ∀i ∈ V, j ∈ V do                                // Update I_ij-s by removing
        I_ij = I_ij \ I[S]                                  // all covered vertices
    [ℓ, I_ℓ[S]] = LargestIncrease(V,S,I)                         // recompute I_ℓ[S]
    if AddOne then                                // AddOne is a control parameter
        k = h = 0; I_kh[S] = ∅
    else
        [k,h,I_kh[S]] = LargestIncreasePair(V,S,I)              // recompute I_kh[S]
```

2.1.1 Computational complexity

The greedy heuristic uses Floyd's algorithm to calculate the distances in the input graph, which needs time $\mathcal{O}(n^3)$. There are nested loops used to build I_{ij}, the complexity of these loops is again time $\mathcal{O}(n^3)$. The calculation of $I[S]$, as well as the update of I_{ij}, has time $\mathcal{O}(n^2)$.

In the main loop of the algorithm there are nested loops, starting with an outer loop to check if there is still any non-empty $I_\ell[S]$ in which it can enter up to n times in the worst case. Then, the inner loop to update all I_{ij} takes time $\mathcal{O}(n^2)$. Both of the auxiliary functions to find the vertex or vertices that make $I[S]$ growing the most, have basically two loops, resulting in time $\mathcal{O}(n^2)$. Taking the outer and inner loop together makes the complexity of this part in total time $\mathcal{O}(n^3)$. Therefore the computational complexity for the heuristic algorithm is $\mathcal{O}(n^3)$.

2.2 Locally greedy algorithm

In our locally greedy algorithm, the purpose is the same as earlier, i.e., to find upper bound on $g(G)$ efficiently. Furthermore, by using local information only we aimed at making the algorithm faster compared to the method introduced in Section 2.1. Therefore, instead of calculating all shortest paths using Floyd's algorithm, we calculate the distances from a specific node v to all nodes not in the geodetic set S by Dijkstra's algorithm.

The details of locally greedy algorithm are given in Algorithm 4. The algorithm takes node v as an input, this node can be a degree-one node for the same reason discussed in section 2.1, or a simplicial node, where the simplicial node is the node such that its neighbors form a clique (a complete subgraph), namely, every two neighbors are adjacent. In [1] the authors proved that simplicial nodes are always part of the geodetic set.

Node v is the starting node for set S. Then `LargestLocalIncrease` function in line 2 returns the node u for which the set $I[S]$ would grow most. This function is detailed in lines 13-19. First, it calculates the distance from node v to all other nodes in the graph together with the shortest paths and fills the sets $I_{v,:}$. Dijkstra's algorithm is not detailed in the pseudocode as it is well-known. In lines 15-17 the function computes the sets $I_j[S]$ for all $j \in V \setminus S$. In line 18 the function finds the node u that would increase $I[S]$ most by adding it to S.

In lines 3-5 the algorithm adds u to the geodetic set S, updates $I[S]$, and then removes $I[S]$ from R, which is the set of remaining nodes to be covered. The main loop of the algorithm is given in lines 7-12, in each iteration the algorithm checks if set R is empty, namely, if there are still any uncovered nodes. As long as there are still uncovered nodes the algorithm will repeat the execution in lines 8 by calling `LargestLocalIncrease` function to node w.

Algorithm 4: Locally Greedy algorithm

Input: v a degree-one or simplicial node

1 $R = V, S = v, I[S] = \emptyset, I_i[S] = \emptyset \quad \forall i \in V, I_{ij} = \emptyset \quad \forall i, j \in V$
2 $[u, I_u[S]] =$ `LargestLocalIncrease`(v, V, S, I) // $I[S]$ would grow most for node u
3 $S = S \cup \{u\}$ // update S
4 $I[S] = I[S] \cup I_u[S] \cup \{v, u\}$ // update $I[S]$
5 $R = R \setminus I[S]$ // update R by removing covered nodes
6 $w = u$
7 **while** $|R| > 0$ **do** // the set is not geodetic yet
8     $[u, I_u[S]] =$ `LargestLocalIncrease`(w, V, S, I) // compute $I_u[S]$
9     $S = S \cup \{u\}$ // update S
10     $I[S] = I[S] \cup I_u[S] \cup \{u\}$ // update $I[S]$
11     $R = R \setminus I[S]$ // update R by removing covered nodes
12     $w = u$
13 **Function** `LargestLocalIncrease`(v, V, S, I)
14     $I_{v,:} =$ `Dijkstra`(v) // shortest paths by Dijkstra algorithm
15   **for** $\forall j \in V \setminus S$ **do**
16   **for** $\forall i \in S$ **do**
17             $I_j[S] = I_j[S] \cup I_{ij}$
18     $u = \underset{j \in V \setminus S}{\operatorname{argmax}} |I_j[S] \setminus I[S]|$ // find the node for which $I[S]$ would grow most
19   **return** $u, I_u[S]$

2.2.1 Computational complexity

The locally greedy algorithm uses Dijkstra's algorithm to calculate the distances in the input graph, which needs time $\mathcal{O}(n^2)$. There are nested loops used to build $I_j[S]$, the complexity for these loops is again $\mathcal{O}(n^2)$. In total, the `LargestLocalIncrease` function needs time $\mathcal{O}(n^2)$. The main while loop of the algorithm is used to fill the geodetic set S by calling the function `LargestLocalIncrease` until set R becomes empty, that is, at most n times. Thus, the total complexity is $\mathcal{O}(n^3)$, like for the other greedy methods proposed in Section 2.1.

3 Numerical experiments

In order to investigate the execution time and quality of the upper bound algorithms discussed in Section 2, the ILP model and the algorithms were implemented in AMPL [7]. As a solver we used Gurobi with parameters setup `mipfocus=1,timelim=3600`. Tests were run on a computer with a 3.10 GHz i5-2400 CPU and 8GB memory.

The obtained results are reported in Tables 1, 2, 3, and 4. The meaning of the columns are:

graph: size of the graphs as number of vertices (n) and number of edges (m);
exact: exact geodetic number (or best solution found by Gurobi in case of running out of time) and the time in seconds to find this solution;
greedy: upper bound found by the greedy algorithm (i.e., Algorithm 3) by using $AddOne = 0$ and its execution time in seconds;
greedy (`AddOne`): upper bound found by the greedy algorithm (i.e., Algorithm 3) using $AddOne = 1$, i.e., the addition of only 1 vertex at a time, and its execution time in seconds;
locally greedy: upper bound found by the locally greedy algorithm (i.e., Algorithm 4) and its execution time in seconds.

3.1 Graph instances

As input graphs for the algorithms, we have generated several random graphs which are structurally different to obtain deeper insights how the solution of the geodetic number problem depends on the graph G. Besides, a small set of real-world graphs were also included in the benchmark.

3.1.1 Random graphs

The random graphs were generated by using the following standard models.

Erdős-Rényi (ER) model [6], where a graph is chosen uniformly at random from the set of all graphs which have n nodes and m edges.

Watts-Strogatz (WS) model [13], which produces graphs with small world properties, namely (i) the average value of all shortest paths in the graph is low, and (ii) high clustering coefficient. Property (ii), which measures the average probability that two neighbors of a node are themselves neighbor of each other, makes them different from ER graphs.

Barabási-Albert (BA) model [2], which creates graphs using preferential attachment growing mechanism, where the more connected a node is, the more likely it is to receive new links. This leads to scale-free property, i.e., power law distribution of the form $p_k \sim k^{-\alpha}$, where p_k is the fraction of nodes with degree k and α is a parameter typically in the range $2 < \alpha < 3$. Note that BA graphs have low clustering coefficient (similarly to ER graphs) and short path lengths (similarly to WS graphs).

Regarding the number of nodes and edges the following approach were used:

- the number of nodes were $n = 10, 20, 30, 40, 50, 60, 70, 80, 90, 100$, and
- for the number of edges we followed the scheme as in [8]:
  - for each case one can have maximum $n \cdot (n-1)/2$ edges,
  - and we took 20%, 40%, 60%, and 80% of this maximum number of edges.
- Apart from these graphs, we created some bigger ones with $n = 115, 135, 150$ nodes using the same procedure as above with the only difference that 25%, 50%, and 75% of the maximum number of edges were taken.

To generate all these input graphs the `R/igraph` package was used with the appropriate functions.

3.1.2 Real-world graphs

The graphs have been used are benchmark graphs, the first set of the graphs taken from UCINET software datasets[1], the other graphs are well-known graphs from Network Repository[2].

3.2 Discussion on results with random graphs

3.2.1 General observations

Before the discussion of the results for the different types of random graphs, a general overview can be summarized as follows.

- The exact solutions were found in less than half of the cases, and this is caused by running out of time. In these cases, the reported solution is a lower bound of the geodetic number.
- Both of the two versions of the greedy algorithm (Algorithm 3) were able to finish their execution below 2 seconds for each and every tested random graph. Their execution times were roughly equal. The version which takes pair of vertices into

[1] https://sites.google.com/site/ucinetsoftware/datasets/
[2] http://networkrepository.com/

account (`AddOne=0`) gives better or equal upper bound compared with the version taking one vertex (`AddOne=1`).
- For the locally greedy algorithm (Algorithm 4) the execution time was less than 0.2 seconds for each tested graph. This algorithm is the fastest one on average.
- Not surprisingly, as the number of nodes increases the geodetic number and its computation time increases as well (for the same density). Taking the averages for the graphs with the same density we can conclude that the geodetic number is decreasing as the density is increasing, while the computational effort is maximal when the density is 0.4, and minimal when it is 0.8.

3.2.2 Erdős-Rényi random graphs

The results for the Erdős-Rényi graphs are reported in Table 1. The ILP solver was able to find the optimal solution within the time limit for 22 instances (45% of the cases).

The greedy algorithm missed the optimal solution in 28 cases. So it reported the optimal value as upper bound in 43% of the cases. For 20 graphs it missed the optimum with one more vertex, and for 7 graphs with two more vertices in the minimal geodetic set. On the other hand, it found a better solution in one case than the reported upper bound by Gurobi. Comparing the two versions of the greedy algorithm: in 38 cases they reported the same values, `AddOne=1` resulted in better upper bounds only for 2 input graphs, while the default version gave better upper bounds in 9 cases.

The locally greedy algorithm reported the optimal solution in 22 cases (45% of the cases). In 16 cases the upper bound missed the optimal solution with one more vertex, in 6 cases with two more vertices and in 5 cases with three or more vertices.

Comparing the greedy algorithm with the locally greedy algorithm: the two algorithms reported the same upper bound in 32 cases. In 7 cases locally greedy algorithm gave better upper bound, while in 10 cases the greedy algorithm was closer to the optimal solution.

The last line of Table 1 reports the mean values for the obtained bounds of the geodetic number as well as the average execution times. Although the differences are really small, the greedy algorithm is the best in getting good upper bounds, whereas the locally greedy method was the fastest one for the Erdős-Rényi random graphs.

3.2.3 Watts-Strogatz

The results for the Watts-Strogatz graphs are shown in Table 2. The exact method ran out of time in 23 instances, so it was only able to solve the problem in 53% of the cases.

The greedy algorithm obtained the same value as the exact method in 29% of the cases (for 14 graphs). When it did not find the optimal (or best reported) value, it gave +1 for 25 graphs and +2 for 10 graphs.

In 38 cases the two versions of the greedy algorithm reported the same values, `AddOne=1` resulted in better upper bounds in one case only, while the default version gave better upper bounds 10 times.

Table 1 Numerical results for the Erdős-Rényi random graphs, time is given in seconds. The best results are highlighted.

graph		exact		greedy		greedy (AddOne)		locally greedy	
n	m	value	time	value	time	value	time	value	time
10	9	4	0.004	4	0.004	4	0.003	4	0.001
10	18	4	0.004	4	0.004	4	0.004	4	0.004
10	30	4	0.007	4	0.002	4	0.002	4	0.002
10	36	3	0.011	4	0.004	4	0.004	3	0.004
20	37	5	0.063	5	0.009	5	0.008	6	0.004
20	76	4	0.073	4	0.011	4	0.011	4	0.006
20	114	4	0.767	5	0.01	5	0.011	5	0.007
20	152	3	0.197	3	0.011	3	0.012	3	0.006
30	87	6	0.906	7	0.018	8	0.017	7	0.011
30	174	6	3.258	8	0.019	8	0.019	6	0.013
30	261	4	2.198	5	0.016	5	0.015	5	0.011
30	348	4	3.259	4	0.02	4	0.018	4	0.009
40	156	7	76.422	7	0.03	7	0.027	9	0.011
40	312	6	89.274	8	0.031	7	0.031	7	0.016
40	468	4	17.424	6	0.039	6	0.037	6	0.011
40	624	3	4.086	4	0.036	4	0.037	4	0.013
50	245	7	208.332	9	0.056	9	0.051	10	0.023
50	490	6	1169.8	7	0.059	8	0.052	8	0.018
50	735	5	832.257	6	0.068	6	0.059	5	0.014
50	1000	3	18.231	4	0.061	4	0.057	4	0.016
60	354	≤ 9	> 3600	10	0.092	11	0.083	11	0.028
60	708	≤ 7	> 3600	8	0.106	8	0.096	8	0.025
60	1062	≤ 5	> 3600	6	0.111	6	0.097	5	0.024
60	1416	4	429.412	4	0.104	4	0.101	4	0.016
70	483	≤ 8	> 3600	10	0.131	10	0.121	12	0.033
70	966	≤ 7	> 3600	9	0.164	9	0.139	9	0.038
70	1449	≤ 5	> 3600	6	0.156	7	0.152	6	0.031
70	1932	4	663.129	4	0.147	4	0.142	4	0.025
80	632	≤ 9	> 3600	9	0.193	10	0.181	13	0.048
80	1264	≤ 8	> 3600	9	0.235	10	0.208	9	0.047
80	1896	≤ 6	> 3600	6	0.242	6	0.215	6	0.039
80	2528	≤ 4	> 3600	4	0.225	4	0.214	4	0.029
90	801	≤ 10	> 3600	10	0.277	13	0.261	15	0.638
90	1602	≤ 8	> 3600	9	0.331	10	0.295	9	0.052
90	2403	≤ 5	> 3600	6	0.347	6	0.316	5	0.041
90	3204	≤ 4	> 3600	5	0.319	5	0.302	5	0.041
100	990	≤ 12	> 3600	11	0.375	14	0.348	15	0.076
100	1980	≤ 9	> 3600	9	0.449	9	0.387	9	0.061
100	2970	≤ 6	> 3600	6	0.479	6	0.423	6	0.047
100	3960	≤ 4	> 3600	4	0.433	4	0.425	4	0.038
115	1638	≤ 13	> 3600	15	0.627	15	0.572	14	0.101
115	3277	≤ 7	> 3600	8	0.746	8	0.648	8	0.074
115	4916	≤ 5	> 3600	5	0.704	5	0.672	5	0.068
135	2261	≤ 14	> 3600	15	0.943	14	0.823	16	0.152
135	4522	≤ 8	> 3600	8	1.107	8	0.953	8	0.107
135	6783	≤ 4	> 3600	5	1.101	5	1.042	5	0.088
150	2793	≤ 15	> 3600	16	1.393	16	1.154	16	0.182
150	5587	≤ 8	> 3600	8	1.528	8	1.356	8	0.134
150	8381	≤ 5	> 3600	5	1.563	5	1.465	5	0.107
average		6.224	2055.492	6.898	0.309	7.122	0.279	7.184	0.053

Table 2 Numerical results for the Watts-Strogatz random graphs, time is given in seconds. The best results are highlighted.

graph		exact		greedy		greedy (`AddOne`)		locally greedy	
n	m	value	time	value	time	value	time	value	time
10	9	5	0.008	5	0.003	5	0.003	5	0.002
10	18	3	0.013	3	0.004	3	0.004	3	0.003
10	30	3	0.012	3	0.004	3	0.004	3	0.003
10	36	2	0.007	2	0.001	2	0.001	2	0.001
20	37	5	0.102	6	0.011	6	0.008	6	0.007
20	76	5	0.512	6	0.011	6	0.011	6	0.008
20	114	4	0.477	4	0.009	4	0.008	4	0.006
20	152	4	0.728	4	0.011	5	0.011	5	0.007
30	87	7	2.294	7	0.019	7	0.018	9	0.013
30	174	6	6.423	6	0.02	6	0.02	7	0.011
30	261	5	3.852	5	0.022	5	0.019	5	0.012
30	348	4	1.052	4	0.019	4	0.018	4	0.006
40	156	7	20.866	9	0.036	9	0.032	10	0.018
40	312	6	106.294	6	0.035	9	0.032	8	0.019
40	468	4	35.792	4	0.038	4	0.035	5	0.015
40	624	3	2.963	4	0.035	5	0.034	3	0.011
50	245	7	82.019	8	0.054	8	0.047	11	0.023
50	490	≤ 7	> 3600	9	0.067	9	0.058	9	0.024
50	735	5	255.426	6	0.072	6	0.063	6	0.019
50	1000	4	4.742	5	0.059	5	0.058	5	0.017
60	354	9	771.2	10	0.094	11	0.085	12	0.031
60	708	≤ 7	> 3600	9	0.102	9	0.094	9	0.031
60	1062	5	561.631	7	0.107	8	0.1	6	0.026
60	1416	4	10.923	5	0.097	5	0.095	5	0.021
70	483	≤ 8	> 3600	10	0.138	9	0.126	11	0.035
70	966	≤ 7	> 3600	9	0.161	9	0.137	9	0.037
70	1449	≤ 5	> 3600	6	0.157	6	0.141	6	0.026
70	1932	4	28.026	5	0.148	5	0.137	5	0.024
80	632	≤ 9	> 3600	10	0.208	10	0.183	13	0.047
80	1264	≤ 8	> 3600	9	0.238	10	0.207	10	0.048
80	1896	≤ 5	> 3600	7	0.239	7	0.216	7	0.042
80	2528	4	390.748	5	0.213	5	0.208	5	0.028
90	801	≤ 10	> 3600	11	0.276	13	0.265	13	0.058
90	1602	≤ 8	> 3600	9	0.336	9	0.286	10	0.061
90	2403	≤ 6	> 3600	7	0.344	7	0.311	7	0.049
90	3204	4	106.156	6	0.327	6	0.302	5	0.041
100	990	≤ 11	> 3600	11	0.371	13	0.343	15	0.082
100	1980	≤ 8	> 3600	9	0.472	9	0.387	9	0.068
100	2970	≤ 6	> 3600	7	0.452	7	0.417	7	0.059
100	3960	4	350.92	6	0.415	6	0.398	6	0.043
115	1638	≤ 13	> 3600	14	0.614	14	0.574	15	0.106
115	3277	≤ 8	> 3600	9	0.715	9	0.648	9	0.091
115	4916	≤ 5	> 3600	6	0.693	6	0.645	6	0.071
135	2261	≤ 14	> 3600	15	0.927	16	0.832	15	0.135
135	4522	≤ 7	> 3600	9	1.112	9	0.952	9	0.116
135	6783	≤ 5	> 3600	6	1.065	6	0.997	6	0.086
150	2793	≤ 13	> 3600	14	1.287	16	1.164	15	0.183
150	5587	≤ 8	> 3600	8	1.497	8	1.338	9	0.141
150	8381	≤ 5	> 3600	6	1.502	6	1.411	6	0.106
average		6.265	1745.78	7.163	0.303	7.45	0.28	7.673	0.043

Table 3 Numerical results for the Barabási-Albert random graphs, time is given in seconds. The best results are highlighted.

graph		exact		greedy		greedy (AddOne)		locally greedy	
n	m	value	time	value	time	value	time	value	time
10	9	6	0.004	8	0.004	6	0.004	6	0.002
10	18	4	0.007	4	0.004	4	0.004	4	0.001
10	30	4	0.007	4	0.004	4	0.004	4	0.002
10	36	3	0.009	3	0.002	3	0.001	3	0.001
20	37	7	0.042	7	0.012	7	0.011	8	0.008
20	76	5	0.058	6	0.011	7	0.009	6	0.007
20	114	4	0.176	5	0.012	5	0.012	5	0.007
20	152	3	0.046	3	0.008	3	0.007	3	0.005
30	87	9	0.269	10	0.021	12	0.02	11	0.011
30	174	6	0.382	7	0.02	8	0.017	7	0.012
30	261	5	0.995	5	0.016	5	0.015	5	0.007
30	348	3	0.724	4	0.019	4	0.019	4	0.011
40	156	10	20.117	11	0.032	12	0.033	12	0.016
40	312	7	4.014	10	0.038	10	0.035	10	0.02
40	468	5	9.844	7	0.037	7	0.034	7	0.017
40	624	4	1.955	4	0.036	4	0.035	4	0.013
50	245	11	6.754	12	0.064	12	0.051	13	0.026
50	490	8	197.846	10	0.063	9	0.056	10	0.022
50	735	5	58.891	7	0.067	8	0.061	6	0.019
50	1000	4	12.052	6	0.059	6	0.053	5	0.014
60	354	≤ 11	> 3600	12	0.097	12	0.092	16	0.037
60	708	≤ 8	> 3600	10	0.103	10	0.096	9	0.031
60	1062	6	645.853	6	0.101	7	0.094	7	0.028
60	1416	4	26.435	4	0.092	4	0.089	4	0.018
70	483	≤ 11	> 3600	11	0.142	14	0.129	18	0.048
70	966	≤ 9	> 3600	11	0.164	11	0.139	11	0.043
70	1449	≤ 6	> 3600	6	0.151	6	0.135	6	0.025
70	1932	4	67.771	5	0.146	5	0.137	5	0.027
80	632	≤ 11	> 3600	11	0.207	12	0.187	18	0.068
80	1264	≤ 9	> 3600	9	0.224	9	0.203	11	0.051
80	1896	≤ 6	> 3600	7	0.231	7	0.208	7	0.041
80	2528	4	132.305	5	0.217	5	0.207	5	0.032
90	801	≤ 13	> 3600	14	0.305	17	0.283	20	0.081
90	1602	≤ 9	> 3600	10	0.315	11	0.281	11	0.064
90	2403	≤ 6	> 3600	7	0.334	8	0.311	7	0.047
90	3204	4	713.099	5	0.305	5	0.283	5	0.042
100	990	≤ 15	> 3600	14	0.392	18	0.363	18	0.101
100	1980	≤ 9	> 3600	10	0.421	13	0.378	13	0.081
100	2970	≤ 7	> 3600	8	0.435	8	0.402	8	0.063
100	3960	≤ 4	> 3600	5	0.416	5	0.387	5	0.052
115	1638	≤ 15	> 3600	16	0.626	18	0.566	18	0.121
115	3277	≤ 8	> 3600	8	0.673	8	0.608	10	0.081
115	4916	≤ 5	> 3600	5	0.648	5	0.632	6	0.062
135	2261	≤ 15	> 3600	15	0.938	18	0.828	21	0.176
135	4522	≤ 9	> 3600	10	1.064	11	0.927	10	0.115
135	6783	≤ 5	> 3600	6	1.053	6	0.963	7	0.106
150	2793	≤ 15	> 3600	17	1.282	17	1.192	20	0.221
150	5587	≤ 9	> 3600	9	1.443	11	1.291	10	0.137
150	8381	≤ 6	> 3600	6	1.415	7	1.342	6	0.118
average		7.27	1802.034	8.061	0.3	8.653	0.27	9.081	0.048

The locally greedy algorithm reported the optimal solution in 8 cases. In 22 cases the upper bound missed the optimal solution with one more vertex, in 12 cases with two more vertices and in 7 cases with three or more vertices.

The greedy algorithm and the locally greedy algorithm reported the same upper bound in 29 cases, the locally greedy algorithm gave better upper bound in 3 cases, while the default version gave better upper bound in 17 cases.

Regarding the average performance, which is reported in the last line of Table 2, the default greedy algorithm was the best with respect to the upper bound, while the locally greedy approach was the quickest for the Watts-Strogatz random graphs.

3.2.4 Barabási-Albert

Finally, for the Barabási-Albert graphs the computational results are reported in Table 3. Gurobi was able to find the optimal solution within the time limit for 25 instances (51% of the cases). The greedy algorithm was not able to obtain the same value as the exact method in 31 cases, meaning a 37% success rate. For 21 graphs it reported +1, for 8 graphs +2 and for one graph +3 compared to the value obtained by the exact method. For one graph instance it found better upper bound than Gurobi. By comparing the two versions of the greedy algorithm: in 29 cases they reported the same values, `AddOne=1` resulted in better upper bound in 2 cases, while the default version gave better upper bounds 18 times.

The locally greedy algorithm achieved the optimal solution in 10 cases. In 19 cases the upper bound missed the optimal solution with one more vertex, in 10 cases with two more vertices and in 10 cases with three or more vertices.

The default version of greedy algorithm and locally greedy algorithm reported the same upper bound in 25 cases. In 4 cases locally greedy algorithm gave better upper bound, whereas in 20 cases the default version was better.

The last line of Table 3 reports the average performances. Similarly to the other two types of random graphs, the default greedy algorithm is the best one for obtaining upper bound for the geodetic number, and the locally greedy algorithm can do this the fastest for the Barabási-Albert random graphs.

3.3 Discussion of the results on larger graph instances

It is obvious from the values reported in Table 4 that ILP can take many hours to get the exact geodetic number for graphs with thousands of nodes and edges, while our proposed algorithms were able to obtain acceptable upper bounds in reasonable time. Note that for this set of graphs we did not use any time limit for Gurobi. Even for the largest graph instance (ia-email-univ) both variants of the greedy algorithm were able to report a slightly worse upper bound than the exact value in less than 700 seconds. Although the locally greedy algorithm was the fastest, for the larger graphs it missed the upper bound by a much larger margin than the greedy method. This trend can also be seen in the last line of Table 4, where the average performances are reported.

Table 4 Numerical results for real-world graphs, time is given in seconds, unless indicated otherwise

graph			exact		greedy		AddOne		loc. greedy	
name	n	m	value	time	value	time	value	time	value	time
karate	34	78	16	0.06	16	0.024	16	0.019	16	0.015
mexican	35	117	7	0.21	7	0.025	8	0.023	9	0.011
sawmill	36	62	14	0.09	14	0.022	14	0.019	15	0.015
chesapeake	39	170	5	0.12	5	0.026	5	0.023	5	0.008
ca-netscience	379	914	253	37 m	256	21.6	260	20.8	264	14.1
bio-celegans	453	2025	172	1 h	183	40.8	188	34.3	225	14.8
rt-twitter-copen	761	1029	459	6 h	459	101.7	459	103.4	490	112.6
soc-wiki-vote	889	2914	275	14 h	276	236.4	277	232.0	409	120.5
ia-email-univ	1133	5451	244	16 h	248	698.6	250	677.9	464	269.0
average			160.6	4 h	162.7	122.1	164.1	118.7	210.8	59.0

4 Conclusions

Given the fact that the graph geodetic number problem is NP-hard, it is desirable to establish efficient algorithmic upper bounds which can provide feasible solutions of acceptable quality and reasonable running time. We have proposed greedy type approaches and experimentally shown that these algorithms can determine such upper bounds. Their running time can be a small fraction of the time required to obtain the exact geodetic number.

Acknowledgements We thank our reviewers for their critical comments and valuable suggestions that highlighted important details and helped us in improving our paper.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002), by grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology, Hungary and by the grant SNN-135643 of the National Research, Development and Innovation Office, Hungary.

References

1. Ahangar, H.A., Fujie-Okamoto, F., Samodivkin, V.: On the forcing connected geodetic number and the connected geodetic number of a graph. Ars Combinatoria **126**, 323-335 (2016)
2. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Reviews of Modern Physics **74**(1), 47–97 (2002)
3. Atici, M.: Computational complexity of geodetic set. International Journal of Computer Mathematics **79**(5), 587–591 (2002)
4. Brešar, B., Klavžar, S., Horvat, A.T.: On the geodetic number and related metric sets in cartesian product graphs. Discrete Mathematics **308**(23), 5555–5561 (2008)
5. Chartrand, G., Harary, F., Zhang, P.: On the geodetic number of a graph. Networks: An International Journal **39**(1), 1–6 (2002)
6. Erdős, P., Rényi, A.: On random graphs. Publicationes Mathematicae **6**, 290–297 (1959)

7. Fourer, R., Kernighan, B.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press (2002)
8. Hansen, P., van Omme, N.: On pitfalls in computing the geodetic number of a graph. Optimization Letters **1**(3), 299–307 (2007)
9. Harary, F., Loukakis, E., Tsouros, C.: The geodetic number of a graph. Mathematical and Computer Modelling **17**(11), 89–95 (1993)
10. Manuel, P., Klavžar, S., Xavier, A., Arokiaraj, A., Thomas, E.: Strong geodetic problem in networks. Discussiones Mathematicae Graph Theory (2018)
11. Soloff, J.A., Márquez, R.A., Friedler, L.M.: Products of geodesic graphs and the geodetic number of products. Discussiones Mathematicae Graph Theory **35**(1), 35–42 (2015)
12. Wang, F.H., Wang, Y.L., Chang, J.M.: The lower and upper forcing geodetic numbers of block–cactus graphs. European Journal of Operational Research **175**(1), 238–245 (2006)
13. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440–442 (1998)