A New Approach for Vertex Guarding of Planar Graphs

Branko Kaučič[1] and Borut Žalik[2]

[1]Faculty of Education, University of Maribor, Slovenia
[2]Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia

Vertex guarding is one of many optimisation problems in graph theory with wide area of applications. It is proven to be NP-hard, therefore fast approximative solutions are significant.

In the paper, at first, known algorithms are considered, and then a new algorithm working on planar graphs is introduced. The new algorithm is based on the dynamic approach and produces better and faster solutions. Its efficiency among other algorithms is demonstrated experimentally. In addition, ideas to additionally improve the algorithm are presented at the end.

Keywords: vertex guarding, graph, optimisation, approximative optimal algorithm, dynamic programming.

1. Introduction

Making decisions about optimal or near optimal solutions arise daily. In many cases, the fact that an optimisation problem is being solved is sometimes not even obvious. The reason for that is, that such problems are mostly simple and small. On the other hand, more complex optimisation problems with huge input data arise that need to be solved efficiently. Luckily, in some cases their solution procedures are adequate for related problems, too.

In the paper, an important optimisation problem of guarding graphs is being tackled. Graph guarding is one of many NP-hard problems in graph theory with a wide variety of applications. It is especially useful in applications where optimum positions of arbitrary facilities have to be determined. The examples are determining the location of radars, TV, telephone or radio transmitters, fire and watch towers, etc. With some modifications, the problems as determining the position of energetic sources (e.g. windmills) can also be considered.

The paper is constructed as follows. In Section 2, the necessary graph theory is given. In Section 3, all existing methods are presented, and in Section 4, our approach using dynamic programming is introduced. Comparison of all algorithms on planar graphs is given in Section 5. The paper is concluded with ideas about future work in Section 6, and summarised in Section 7.

2. Preliminaries

An *(undirected) graph* $G = (V(G), E(G))$ consists of a finite, nonempty set $V(G)$ of *vertices*, and a set $E(G)$ of unordered pairs of vertices called *edges*. Two distinct vertices u and v are *adjacent* iff $(u, v) \in E$, and we say that u is a neighbour of v and vice versa. For vertex $v \in V(G)$, the *neighbourhood* of v, denoted $N(v)$, is a set of vertices $u \neq v$ that are adjacent to v. Similarly, for planar graphs, the *triangular neighbourhood* of v, denoted $N_{\triangle}(v)$, represents a set of triangles that have v as one of their vertices. For general graphs, the triangular neighbourhood is defined differently. For example, for visibility graphs, $N_{\triangle}(v)$ is defined as a set of triangles that can be seen from vertex v. The *neighbourhood of set* S, $N(S)$ is a union of all neighbourhoods $N(v)$ of vertices $v \in S$. Similarly, the *triangular neighbourhood of set*

S, $N_\triangle(S)$ is a union of all triangular neighbourhoods $N_\triangle(v)$. Note that graph G must contain triangles. In the continuation, we will use only such graphs, where all vertices are a part of at least one triangle. We will denote all triangles of graph G with $T(G)$.

The triangular neighbourhood can be seen as a guarded area, and we say that triangles in $N_\triangle(v)$ are guarded from v. Figure 1 with grey shaded area shows an example of this. Graph G is *(vertex) guarded* when all triangles are guarded from a set S of vertices v.

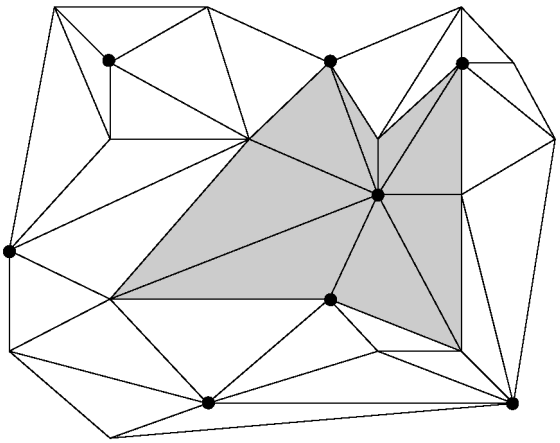

Fig. 1. An example of triangular neighbourhood and vertex guarding.

The term "covered" is deliberately omitted because it is reserved for similar properties in graph theory where graph G is *covered* when $N(S) = V(G)$. Similarly, instead of vertices, edges can be used and we talk about *edge guarding*. In the paper, we considered vertex guards, only.

The optimisation problem is to find a set S of vertices $V(G)$, such that all triangles are guarded, i.e. $N_\triangle(S) = T(G)$. An example of vertex graph guarding can be seen in Figure 1, where all vertices from S are presented by black dots. We call S a *guarding set* and the minimum S an *optimal guarding set*. Therefore, the optimisation problem is to find the smallest S for a given graph G.

Closely related to vertex guarding is a branch of graph theory, the theory of domination numbers. A subset D of vertices is called a dominating set of G if for every $v \in V - D$ there is some $u \in D$ such that $(u, v) \in E(G)$. The minimum cardinality of the dominating set D of G is called the *domination number* of G and is denoted by $\gamma(G)$. Sometimes a dominating set is referred to as a vertex-vertex cover [5]. If triangles of the graph are considered as points (Figure 2), vertex guarding is a special case of searching the domination number $\gamma(G)$.

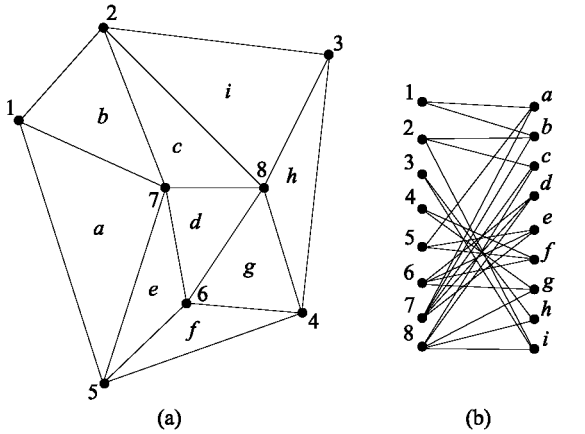

Fig. 2. Connection of vertex guarding with the domination numbers: (a) original graph, (b) triangles substituted with vertices.

The later has been an active area of research for many years [4] and it has been shown the problem is NP-hard [2]. It remains NP-hard even when G is restricted to certain simple classes of graphs. See a work by Livingston [5] for a survey on this subject.

3. Existing Methods

While most of vertex guarding and searching the domination numbers was done theoretically, only three approximative optimal algorithms (greedy add, greedy add with swap, and stingy drop method) based on greedy approach have been mentioned in [3,6]. They have been used for terrain visibility (visibility graphs).

In the continuation, five algorithms are briefly described. The first two are brute-force algorithms, and the last three algorithms are the mentioned algorithms based on the greedy approach. For every algorithm, a short explanation with its pseudocode is given.

a) Brute-Force algorithm *(BF)*:

Brute-force algorithm represents the most naive approach and works by using backtracking. It generates all possible combinations of vertices and stores the solution with the smallest number of vertices so far. The algorithm is presented in Pseudocode 1. S represents temporary solution, v temporary vertex, and GT temporary guarded triangles.

```
Brute-Force(v,GT,S)
{
  if (!Last(v) and !GoodCover(GT))
  {
    AddTriangles(v,GT);
    S ← S ∪ v;
    Brute-Force(Next(v),GT,S);
    RemoveTriangles(v,GT);
    S ← S \ v;
    Brute-Force(Next(v),GT,S);
  }
  if (GoodCover(GT))
    CheckAndUpdateSolution(S);
}
```

Pseudocode 1. Brute-force algorithm.

The algorithm has the ability to find all optimum solutions. Until it has generated all possible combinations, it does not know if it has encountered the optimum solution. Because all combinations are computed, the time complexity is obviously exponential.

b) Improved Brute-Force algorithm *(IBF)*:

In contrast to the previous algorithm, only one optimum solution is searched. Therefore, in improved version the brute-force algorithm is modified to generate combinations with limited number of vertices only-one less than the smallest number of vertices in the best solution so far ($BestNumSoFar$). Its algorithm is given in Pseudocode 2.

The algorithm comes rapidly near the optimum solution, but still needs to check all possible combinations with one vertex less than in the optimum solution. Therefore, the time complexity remains exponential.

```
IBrute-Force(v,GT,S,Count)
{
  if (!Last(v) and !GoodCover(G) and
     Count < BestNumSoFar)
  {
    AddTriangles(v,G);
    S ← S ∪ v;
    IBrute-Force(Next(v),G,S,Count+1);
    RemoveTriangles(v,G);
    S ← S \ v;
    IBrute-Force(Next(v),G,S,Count);
  }
  if (GoodCover(GT))
    CheckAndUpdateSolution(S);
}
```

Pseudocode 2. Improved brute-force algorithm.

c) Greedy Add algorithm *(GA)*:

The algorithm represents the first smart approach based on greedy programming. Corresponding data structure is a triangle-driven adjacency list as shown in Figure 2 (b). At every step, the vertex that covers the largest number of triangles (the largest area) is added in the solution. Then, the data structure is updated where needed to eliminate the influence of the just selected vertex. The algorithm continues until all triangles are guarded. Its pseudocode is given in Pseudocode 3.

```
GreedyAdd()
{
  S ← ∅;
  NumGuarded = 0;
  while (NumGuarded < NumTriangles) {
    v, NumNew=FindMaxVertex();
    NumGuarded += NumNew;
    S ← S ∪ v;
    UpdateDataStructure();
  }
}
```

Pseudocode 3. Greedy add algorithm.

d) Greedy Add with Swap algorithm *(GAS)*:

In the previous algorithm, it can happen that when a new vertex is added to the solution, some vertices are not needed in the solution any more. In such a case, each redundant vertex is identified and removed from the solution. The

modification of previous algorithm is shown in Pseudocode 4.

```
GreedyAddWithSwap()
{
  S ← ∅;
  NumGuarded = 0;
  while (NumGuarded < NumTriangles) {
    v, NumNew=FindMaxVertex();
    NumGuarded += NumNew;
    S ← S ∪ v;
    FindAndRemoveRedundantVertex();
    UpdateDataStructure();
  }
}
```

Pseudocode 4. Greedy add with swap algorithm.

e) Stingy Drop algorithm *(SD)*:

The algorithm works in the way opposite to the previous two algorithms. At first, all vertices are added to the solution. Next, at every step the redundant vertex is searched and removed from the solution. The redundant vertex is a vertex that covers the smallest number of triangles and can be removed from the solution without any triangle becoming unguarded. As in previous algorithms, the same adjacency list is used. The algorithm is presented in Pseudocode 5.

```
StingyDrop()
{
  S ← V(G);
  while ((v =FindRedundant(S))!= null) {
    S ← S \ v;
    UpdateGuardedTriangles();
  }
}
```

Pseudocode 5. Stingy drop algorithm.

4. Our Approach

Last three algorithms in the previous Section use the same idea (common sense): take the "best" vertex and put it in the solution. The best vertex is a vertex that covers the most triangles, or that covers the least triangles and is redundant in the solution. Clearly, they do not produce optimum solutions, but approximative optimal.

Our approach operates in a completely different way. It searches for the vertex that almost certainly will not be in the solution and from its neighbouring vertices select the vertex that almost certainly is in the optimum solution. It works on the basis of dynamic programming and at each step three operations are performed:

1. The vertex that guards the smallest number of vertices is identified. Such vertex is usually found as a part of boundary of the graph. It is called *min vertex*.
2. An arbitrary triangle from $N_{\triangle}(\textit{min vertex})$ is selected. Next, a vertex that guards that triangle (neighbouring vertex in our case of planar graphs) and that guards the largest number of triangles is identified. By intuition: if we know that min vertex is not a part of the solution, then a part of the solution is clearly a vertex that guards one of its triangle(s). Such vertex is called *max vertex*.
3. All triangles that are guarded by the max vertex are removed from the graph. At the same time, it is checked if some of neighbouring vertices of max vertex can be removed, too.

An example of that can be seen in Figure 3.

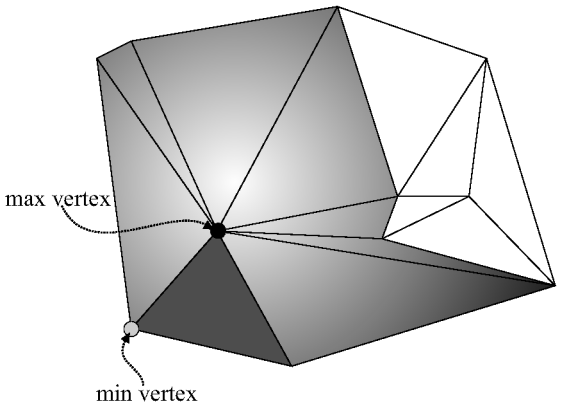

Fig. 3. An example of how our approach works.

As min vertex, the vertex at light grey dot is selected and its triangle is shaded with dark grey colour. As the max vertex, the vertex plotted by a black dot is selected and all neighbouring triangles (shaded with light grey) are removed from the graph. Together with the min vertex, three other vertices are removed, too. For the next step remains the graph with 6 vertices and 5 triangles.

The algorithm is presented in Pseudocode 6.

```
OurApproac|h()
{
  S ← ∅;
  while (!Empty(G)) {
    Min = FindMinVertex();
    T = FindTriangle(Min);
    Max = FindMaxVertex(Min,T);
    S ← S ∪ Max;
    RemoveTriangles(Max);
    RemoveVertices(Max);
  }
}
```

Pseudocode 6. Algorithm of our approach.

5. Comparison of Algorithms

All algorithms were implemented in C++ and optimised with several programming techniques to decrease running times. In our approach, for an operation where the guarded triangle of min vertex is selected, just the first neighbouring triangle is selected. We were interested in the quality of the solutions and running times. The quality is observed as the number of vertices in the computed solutions and a performance bound ε by the equation [1]:

$$\frac{|G_A| - |G^*|}{|G^*|} \leq \varepsilon \,, \qquad (1)$$

where $|G_A|$ represents number of vertices in the approximative solution, and $|G^*|$ number of vertices in the optimum solution. For testing, a portable computer with Intel Celeron 500MHz and 64MB memory has been used. Planar graphs have been generated with n random points and triangulated with fast Delaunay triangulation [7]. Five different graphs for each n have been used.

Table 1 shows the number of vertices in solutions for several different graphs. Because of high time complexity of brute-force approaches, the optimum results are given only for small number of vertices. Whenever our algorithm in average produced better approximative solutions, the results are emphasized, and where our algorithm in average produced the same results as the other best approximative algorithm, the results are italicized. Table 2 shows average and maximum error (performance bound ε) of the solutions, and Table 3 gives running times of the algorithms.

n	*BF*	*GA*	*GAS*	*SD*	Our
10	3,0	3,4	3,4	3,6	**3,2**
15	5,0	5,2	5,2	5,2	*5,2*
20	7,0	7,8	7,8	8,0	*7,8*
25	8,8	9,4	9,4	10,0	9,6
30	11,0	12,2	12,2	11,6	**11,2**
35	12,6	13,4	13,4	13,4	**12,8**
40	14,5	15,8	15,8	16,4	**15,4**
1000	?	422,8	422,8	431,2	**394,4**
2000	?	839,0	839,0	862,6	**788,6**
3000	?	1260,8	1260,8	1289,2	**1178,4**
4000	?	1689,2	1689,2	1721,2	**1577,2**
5000	?	2110,0	2110,0	2159,8	**1988,6**

Table 1. Comparison of solutions.

n	*GA*	*GS*	*SD*	Our
avg	0,088	0,088	0,106	**0,058**
max (ε)	0,133	0,133	0,200	**0,114**

Table 2. Comparison of error in approximability of solutions by equation (1).

n	*GA*	*GS*	*SD*	Our
1000	0,78	0,94	0,03	**0,02**
2000	3,99	4,26	0,14	**0,08**
3000	9,31	9,80	0,40	**0,24**
4000	17,22	18,01	1,31	**0,59**
5000	27,66	28,59	2,16	**1,10**

Table 3. Comparison of CPU running times (seconds).

It can be seen that our algorithm dominates in the quality of the solutions and in running times. The bigger the graphs are, the better solutions are produced with our algorithm. In some cases, our algorithm produced worse solutions than algorithm *GA*. The reason for this is a simple criterion for selecting the min vertex. Therefore, in some cases the selected min vertex is not a part of the boundary of the graph. The testing has also shown, that *GA* and *GAS* always produced the same solutions, which is probably because planar graphs were used. In other types of graphs, this is not expected.

6. Future Work

During testing, a few new ideas arised. Perhaps a swap technique used in *GAS* would find redundant vertices in the solution. In addition, different ways of selecting the neighbouring triangle of min vertex and consequently the max vertex can be applied. Random triangle or one-step look-ahead approach where max vertex that guards the most triangles from $N_{\triangle}(\mathit{min\ vertex})$, can be used. Solutions will probably be better if selection of min vertex is restricted to vertices on the boundary of the graph, but will probably increase the running time. The running time can be additionaly improved by the fact that if min vertex is not removed, the same min vertex will be selected as the next min vertex.

It is also interesting how our approach can be used for non-planar graphs, especially visibility graphs, and for other types of guarding, especially edge guarding.

7. Conclusion

Optimisation problems on graphs arise daily and have wide area of applications. In the paper, an approximative optimisation algorithms for vertex guards are considered. Known algorithms are presented, and then our new approach is given. It is shown that our approach produces better solutions and needs less CPU time than all known algorithms.

Acknowledgments

The authors wish to thank Boštjan Ledinek, who helped them at programming.

References

[1] R. BAR-YEHUDA, S. EVEN, On approximating a vertex cover for planar graphs. *Proceedings 14th Annual ACM Symposium on Theory of Computing*; 1982 May 5–7, San Francisco, United States; 1982. pp. 303–309.

[2] M.R. GAREY, D.S. JOHNSON, *Computers and Intractability — A Guide to the Theory of NP-Completeness*. San Francisco: Freeman, W.H. & Company; 1979.

[3] M.F. GOODCHILD, J. LEE, Coverage problems and visibility regions on topographic surfaces. *Annals of Operations Research* 1989; 18: pp. 175–186.

[4] S.M. HEDETNIEMI, R.C. LASKAR, Bibliography on domination in graphs and some basic definitions of domination parameters. *Discrete Math.* 1990; 86: pp. 257–277.

[5] M. LIVINGSTON, Q.F. STOUT, Constant Time Computation of Minimum Dominating Sets. *Congressus Numerantium* 1994; 105: pp. 116–128.

[6] J. LEE, Analyses of visibility sites on topographic surfaces. *Int. J. Geographical Information Systems* 1991; 5(4): pp. 413–429.

[7] B. ŽALIK, I. KOLINGEROVA, An incremental construction algorithm for Delaunay triangulation based on two-level uniform subdivision. *Contributions to Geometric Modelling and Multimedia* 2001; 1(5): pp. 1–32.

Received: June, 2002
Accepted: September, 2002

Contact address:
Branko Kaučič
Faculty of Education,
University of Maribor
Koroška c. 160,
SI-2000 Maribor, Slovenia
e-mail: branko.kaucic@uni-mb.si

Borut Žalik
Faculty of Electrical Engineering and Computer Science,
University of Maribor
Smetanova ul. 17, SI-2000 Maribor, Slovenia
e-mail: zalik@uni-mb.si

BRANKO KAUČIČ is a PhD candidate at the Faculty of EE & CS, University of Maribor, Slovenia, since 2001. His main interests are computational geometry, terrain visualisation and terrain visibility. He is currently a teaching assistant at the Department of Mathematics at the Faculty of Education, University of Maribor, Slovenia.

BORUT ŽALIK is an associate professor in the Department of Computer Science, Faculty of EE & CS, University of Maribor, Slovenia. He received his BSc in electrical engineering in 1985, MSc and PhD in computer science, both from the University of Maribor in 1989 and 1993 respectively. He is also a visiting senior research fellow at De Montfort University, U.K. His research interests include geometric modelling, computational geometry, GIS, and multimedia applications.