An Implicit Backtracking Implementation of Exact Vertex Cover Algorithms

Faisal Abu-Khzam and Karam Al Kontar

Lebanese American University

June 2019

1 Introduction

This paper describes the implementation technique used to develop the Exact Vertex Cover solver submitted to the PACE challenge 2019. The source code is publicly available and can be found here.

The following sections explain the main steps, starting from reading the input graph, applying preprocessing followed by branching, to finally printing the optimal solution.

In the algorithm, we aim to achieve a simple, easy-to-implement algorithm, that performs well on general graphs that reflect real-life instances and usually do not exhibit worst case scenarios. We believe that more sophisticated methods, which may perform better on a small number of hard instances, tend to be slower on the general instances arising from real-world applications.

2 Preliminaries

Throughout the algorithm, we employ a number of methods for memory management using various forms of graph representation as well as other auxiliary data structures, used mainly to implement an implicit backtracking scheme.

2.1 Memory Management

In order to speed up the process of allocating and freeing memory during the branching phase, we use a a 2D dynamic array of arrays of size $V(G)$. Since most allocations during execution will require an array of size $V(G)$, we create something similar to an arraylist of arrays, which doubles in size when needed. This guarantees that every array will be allocated exactly once, and given to the method in need. When the array is freed, it is just marked as available, so any other method in need of an array can reuse it without a new memory allocation.

2.2 Edge List

We use a struct to store the edges of the input into an array. These edges have the vertex of smaller index before that of larger index, and are sorted by the first vertex, then by the second. This helps us find (and remove) duplicate edges. It is also useful in verifying the final solution.

2.3 Graph Representation

For the graph representation, we a hybrid method to efficiently delete and restore edges. The representation consists of an adjacency list of size linear in the number of edges. We also use an index matrix; $IM[u][v]$ is the index of u in the adjacency list of the vertex that is at index v in the adjacency list of u. Whenever we want to delete the vertex u, we loop over all of its neighbors in the adjacency list AL, and for each neighbor, we find the index of u in that neighbor's adjacency list, swap it with the last element in the neighbor's adjacency list, and decrement the neighbor's degree. Hence, deleting an edges requires constant time. Restoring the edge also requires constant time, as all that is required is to restore the degree of the neighbor. This approach was first presented in [3].

The list of vertices is kept in an array L, where IL is the index of the vertex in this list. This also aids in deletion of vertices, as the vertex is moved to the last index and the size of the list is decremented. Restoring the size, restores the vertex.

For handling connected components, we use the array L as well, with two other arrays st and en. $st[i]$ and $en[i]$ denote the indices in L at which the i^{th} connected component begins and ends respectively. Hence, L is partitioned into contiguous subarrays representing the connected components.

2.4 Color Lists

During branching, we may sometimes choose to merge two vertices into one vertex to perform the folding technique (introduced in [5]) on degree-two vertices. Initially, every vertex has some color, and every color has only one vertex. By merging two vertices (or two colors to be precise), we set the color of both colors equal to the color of one of them.

For simplicity, let's use "bag" to mean a set of vertices with the same color. The graph can now be viewed as a set of bags instead of a set of vertices. The *color* array points to the color of a vertex (which bag it belongs to). The *color_deg* array is the degree of the bag, which is the sum of the degree of the vertices. We will show later that vertices of the same bag are all pairwise independent. To be able to loop on vertices of a bag, to delete or calculate degree or find neighbors, we represent a bag as a linked list, with a head stored in *color_head*. For every vertex, we have to find the vertex before it and that after it in the bag, which are stored in the *color_prev* and *color_next* arrays.

This representation allows us to loop on the vertices of a bag. Also, whenever a vertex is added to a bag, we can add it to the head in constant time. Moreover, deleting a vertex from a bag also takes constant time, as we only change the previous and next pointers in the vertex's predecessor and successor.

3 Algorithm

Recall that we no longer branch of the graph vertices, but rather on the bags or colors discussed previously. All vertices with self-loops are put in the solution and deleted. All duplicate edges are discarded. All isolated vertices are also deleted.

Before we start branching, we use the factor 2 approximation algorithm for vertex cover to construct a starting best solution. Note that our code may potentially run faster given a better first-guess (starting solution).

Before we proceed to branching, if the current solution is not less than the best solution, then we stop the current branch. Otherwise, we check if the graph is edgeless and update the best solution if it is. Finally, if the graph is not edgeless and the current solution is not less than the best solution minus 1, we also stop the current branch.

3.1 Preprocessing

In the preprocessing phase, we remove bags of large degree (vertices that, if not in the solution, their neighbors would cause the current solution to exceed the best solution so far).

Then, we remove bags of degree one, and include their neighbors (also removing them).

After that, we find and return bags of degree two if any.

Finally, we detect bags of degree three, whose neighborhood forms a clique. For such bags, it is best to include all the three neighbors in the solution, and delete them with the degree 3 vertex.

3.2 Folding

Whenever preprocessing returns a bag of degree two, it is best not to branch, but rather to fold. Denote by w the degree-two bag, and by u and v its neighbors.

If the neighbors of the bag are adjacent (forming a triangle), it is not worse to include both neighbors, u and v, in the solution, and delete all three bags.

Otherwise, we include w in the solution and delete it, and merge u and v into one bag u. The branching method is invoked. When it returns, if the new solution contains u, we replace w by v. As mentioned previously, the vertices of a bag are all pairwise independent due to the fact that bags are merged only if they are not adjacent.

3.3 Low Branches

When the graph consists of at most 20 vertices, preprocessing, connected components, and other activities meant to speedup branching may in fact slow it down, as the graph is quickly decomposed by branching. Since the number of leaves in a tree is close to the total number of internal nodes, resolving these leaves quickly gives a great advantage.

3.4 Connected Components

After 30 consecutive branching operations, we invoke a connected components method. This method partitions the graph into connected components as discussed in the preliminaries, using graph traversal. For each connected component, a sub-problem is created, and the factor 2 approximation algorithm is applied to find a starting best solution. This means that all branching will be focused on one connected component at a time.

3.5 Bipartite Graphs

In case the graph becomes bipartite, which is checked while finding the connected components, the known poly-time algorithm (Konig's theorem), which is based on matching, is applied instead of recursive backtracking.

3.6 Maximum Degree

Branching is always performed on a bag of maximum degree. While finding this maximum-degree bag, we also perform some checks on whether to proceed with branching. First, we let k be the maximum number of bags that can be added while keeping the current solution less than the best so far. If the k bags of highest degree have a total degree less than the number of edges, then we stop the current branching. In addition, if the number of non-isolated bags exceeds $k * (1 + \frac{maxdeg}{mindeg=3})$, then we also stop the current branch. Otherwise, we return the max degree bag.

3.7 Recursive Backtracking

The state is stored, and the branching method is invoked on the instance after including the maximum-degree bag, call it v, in the solution and deleting it. After that, the state is restored.

If adding the neighbors of v will result in a solution greater than the best so far, we return. Otherwise, we include the neighbors of v in the solution, and delete them along with v.

4 Conclusion

The approach presented in this paper provides simple, yet powerful, techniques for implementing recursive backtracking algorithms. Many of the presented ideas, like connected components and color bags, provide useful implementation solutions to many reduction and branching rules, not only in vertex cover, but also in other recursive backtracking algorithms.

All in all, although the approach may not give the fastest-known algorithm, it is novel for its generality and simplicity, and it tends to deliver the fastest solutions on general graphs and many real-instances because they don't usually exhibit worst-case scenarios that require sophisticated reduction/pruning methods such as ILP-based kernelization and the crown decomposition method (introduced independently in [1] and [7] and developed further in [4], [2] and [6]).

In fact, we decided not to perform these powerful (yet sophisticated) methods when we noticed they make the code slower on general instances that are more likely to be faced in real-life applications.

5 Source Code

The code for the described solver is publicly available and can be accessed through the following link:

`https://github.com/karamkontar99/Vertex-Cover-Solver`

References

[1] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In L. Arge, G. F. Italiano, and R. Sedgewick, editors, *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics, New Orleans, LA, USA, January 10, 2004*, pages 62–69. SIAM, 2004.

[2] F. N. Abu-Khzam, M. R. Fellows, M. A. Langston, and W. H. Suters. Crown structures for vertex cover kernelization. *Theory Comput. Syst.*, 41(3):411–430, 2007.

[3] F. N. Abu-Khzam, M. A. Langston, A. E. Mouawad, and C. P. Nolan. A hybrid graph representation for recursive backtracking algorithms. In D. Lee, D. Z. Chen, and S. Ying, editors, *Frontiers in Algorithmics, 4th International Workshop, FAW 2010, Wuhan, China, August 11-13, 2010. Proceedings*, volume 6213 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 2010.

[4] F. N. Abu-Khzam, M. A. Langston, and W. H. Suters. Fast, effective vertex cover kernelization: a tale of two algorithms. In *2005 ACS / IEEE International Conference on Computer Systems and Applications (AICCSA 2005), January 3-6, 2005, Cairo, Egypt*, page 16. IEEE Computer Society, 2005.

[5] J. Chen, I. A. Kanj, and W. Jia. Vertex cover: Further observations and further improvements. *J. Algorithms*, 41(2):280–301, 2001.

[6] M. Chlebík and J. Chlebíková. Crown reductions for the minimum weighted vertex cover problem. *Discrete Applied Mathematics*, 156(3):292–312, 2008.

[7] B. Chor, M. Fellows, and D. Juedes. Linear kernels in linear time, or how to save k colors in o(n^2) steps. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 257–269. Springer, 2004.