2017

Engineering an Efficient Branch-and-Reduce Algorithm for the Minimum Vertex Cover Problem

Haonan Zhong
Colgate University, hzhong2@colgate.edu

Bachelor Thesis

Engineering an Efficient Branch-and-Reduce Algorithm for the Minimum Vertex Cover Problem

Haonan Zhong

Date: May 15, 2017

Advisor: Prof. Darren Strash

Technical Report: COSC-TR-2017-03

Department of Computer Science
Colgate University
Hamilton, New York

Abstract

The Minimum Vertex Cover problem asks us to find a minimum set of vertices in a graph such that each edge of the graph is incident to at least one vertex of the set. It is a classical NP-hard problem and in the past researchers have suggested both exact algorithms and heuristic approaches to tackle the problem. In this thesis, we improve Akiba and Iwata's branch-and-reduce algorithm, which is one of the fastest exact algorithms in the field, by developing three techniques: dependency checking, caching solutions and feeding an initial high quality solution to accelerate the algorithm's performance. We are able to achieve speedups of up to 3.5 on graphs where the algorithm of Akiba and Iwata is slow. On one such graph, the `Stanford` web graph, our techniques are especially effective, reducing the runtime from 16 hours to only 4.6 hours.

Acknowledgments

I would like to thank my thesis advisor, Prof. Darren Strash, for pushing me forward and guiding me as I worked on my thesis. He provided encouragement and advice throughout my time as his advisee and I am extremely lucky to have a advisor who cares so much about my work. I would like to thank the professors, lab instructors and all other staff at the Department of Computer Science. They led me to the amazing world of computer science and guided me as I explored different topics and fields. Special thanks to my computer science advisor Prof. Phil Mulry for instructing me in the past four years. Finally, I would like to thank my family and friends who always support me and help me relieve stress.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Hamilton, NY May 15, 2017

. . . . . .

(Haonan Zhong)

Contents

1 Introduction

The Minimum Vertex Cover problem asks us to find a minimum set of vertices in a graph such that each edge of the graph is incident to at least one vertex of the set. It is a classical NP-hard optimization problem in computer science and so far no one knows if it can be solved in polynomial time. Along with the Minimum Vertex Cover the related Maximum Independent Set and Maximum Clique problems are also in the class.

This problem and its related problems have many applications. For example, it is used in computational biochemistry to resolve conflicts between DNA sequences [32]. Two DNA fragments can have a conflict if they are similar but one potentially has an error due to DNA sequencing. We can build a graph in which vertices represent different DNA fragments and there is an edge between two vertices if the two fragments are in conflict. A minimum vertex cover in the graph can help us detect DNA fragments we suspect that are likely candidates to have errors.

Another application of the Minimum Vertex Cover problem is advanced real-time rendering. Sander et al. [36] formed a dual graph of a triangular mesh, in which they used vertices to represent each face in the mesh and edges to show that two faces are adjacent to each other. Computing a minimum vertex cover of the dual graph, Sander et al. are able to efficiently find a fast traversal of all mesh faces which speeds up the performance of rendering of these meshes.

Cheng et al. [18] invented a disk-based index for fast lookup of single-source shortest path queries and distance queries between any two vertices in a graph. They use a small vertex cover of the graph and compute the distances between vertices in the cover to create their index. An minimum vertex cover can also help them reduce their index size.

1.1 Motivation

Many past studies have been conducted to tackle the Minimum Vertex Cover problem and the complementary Maximum Independent Set problem [2, 3, 9, 11, 12, 22]. Akiba and Iwata [2] implement an advanced *branch-and-reduce* algorithm which intermixes branching with

reduction rules that repeatedly reduces the size of the graph. They compare the algorithm to two state-of-the-art solvers: a branch-and-cut method with an integer programming formulation of the problem by CPLEX, a commercial integer programming solver, and a branch-and-bound method called MCS [41]. They showed that their branch-and-reduce algorithm is superior to those techniques used in industry to solve the Minimum Vertex Cover problem, especially on social networks and web crawl graphs such as `petster-dog` and `BerkStan` [2], which are large and sparse. However, their algorithm still has trouble solving some of the most complex graphs such as the `Stanford` web graph. Strash [38], however, showed that Akiba and Iwata's branch-and-reduce algorithm is not necessary for computing a minimum vertex cover for many instances from Akiba and Iwata's experiments. Processing the graph by first applying a sequence of simple reduction rules is sufficient to solve most graphs in practice.

Based on Strash's empirical results, we hypothesize that by reserving advanced reductions for "difficult" portions of a graph and using a better pruning technique, Akiba and Iwata's algorithm can be sped up on difficult graphs. Also, although the branch-and-reduce algorithm is shown to be more efficient than other state-of-the-art algorithms on most of the graphs from Akiba and Iwata's experiments, it fails to solve some graphs in a reasonable time. For instance, with their original algorithm, it takes 67 270 seconds (18.6 hours) to find a minimum vertex cover for the `Stanford` web graph and it takes 24 365 seconds (6.7 hours) to complete the `sanr400_0.7` [2] graph. By comparison, it only takes the commercial integer programming solver CPLEX 1 836 seconds to complete the `Stanford` graph. It shows that the algorithm is not superior to other solvers in all instances. In this paper, we want understand why their algorithm is slow on instances like the `Stanford` graph, and in addition, improve the time it takes to solve them.

1.2 Our Contribution

In this thesis, we intend to speed up the branch-and-reduce algorithm from Akiba and Iwata by applying reductions in a more targeted and less time-consuming way, and by developing a more advanced pruning technique. With these techniques, we are trying to see whether we can more quickly handle difficult graphs like the `Stanford` web graph, road networks and the `LiveJournal` instance. We develop three techniques. First, we apply reduction rules in a different order so that effective reductions can be applied more times than less effective ones. For instance, for graphs that can be easily solved with lightweight reductions from the exact exponential algorithm by Fomin et al. [23], we apply those rules as much as possible to solve the graph. If those rules encounter the "difficult" portion of the graph, we switch to advanced techniques like the unconfined reduction rule by Xiao and Nagamochi [44]. In

this case graphs such as `Skitter` [2] become 1.3 times faster. For the `Stanford` graph, we even achieve a speedup of 3.5. Finally, we apply a better pruning technique for branching to avoid redundant searching. Instead of repeatedly evaluating all disconnected components, which are groups of vertices in a graph that are connected together but separated from each other, we cache previous solutions to subgraphs and avoid repeated computations. We also show that seeding the branch-and-reduce algorithm with a high quality initial solution helps the algorithm prune more effectively. With all the techniques, we are able to reduce the running time of the `skitter` graph from 2 988.275 seconds to 2 286.676 seconds and the `hollywood-2011` graph from 76.557 to 26.672 seconds.

1.3 Organization

In Chapter 2, we discuss in depth about past related work on the Minimum Vertex Cover problem. In Chapter 3 we explain basic concepts of graphs and describe the branch-and-reduce algorithm by Akiba and Iwata. In Chapters 4 and 5 we explain our two techniques: dependency checking and pruning, and show their effectiveness. In Chapter 6 we give extensive experimental results to compare the effectiveness of our techniques and in the last chapter we conclude our findings and suggest potential future work.

2 Related Work

Many researchers have studied the Minimum Vertex Cover problem and its related problems the Maximum Independent Set problem and the Maximum Clique problem. These problems are computationally equivalent. Suppose we have a graph G. A maximum clique in G's complement graph $\bar{G}$ is also a maximum independent set for G. A maximum independent cover I in G can then be used to find a minimum vertex cover C as $C = V \setminus I$. An example of the three problems in one graph is shown in Figure 2.1.

2.1 Exact Algorithms

Many exact branch-and-bound algorithms can exactly solve the Maximum Clique problem on small instances [35, 40, 41]. These techniques recursively build a maximum clique and conduct upper bound checks to avoid unnecessary branching. One of the most well-known branch-and-bound algorithms is MCR by Tomita et al. [40]. They used a coloring method [42] with a initial vertex ordering for their algorithm to efficiently find a upper bound of the solution size, and use the bound to reduce search space. Tomita et al. [41] further improved the coloring technique, and created a new branch-and-bound algorithm called MCS. The new algorithm is proven to be more effective for dense graphs than MCR.

Batsyn et al. further [4] showed that initializing MCS [41] with an initial near-optimal solution generated using the local search algorithm by Andrade et al. [3] helped the algorithm improved significantly on many benchmark instances. Their approach has a speedup of 4 over MCS in solving DIMACS graphs. For gen instances there is even a speedup of 11 000 compared to MCS. Overall their algorithm is extremely effective in dense and large graphs.

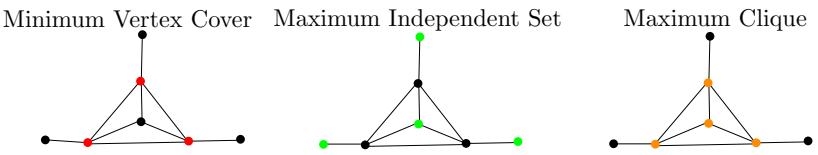

Figure 2.1: Three NP-hard problems

The idea of reducing graph size to lower graph complexity is another technique that has long been studied [1, 11, 24]. A rule for reducing graph size while maintaining the ability to compute an optimal solution is called a *reduction*. These methods are crucial in the Minimum Vertex Cover and the Maximum Independent Set problems. Abu-Khzam et al. [1] showed that a crown structure can be used to quickly identify a minimum vertex cover in a graph. They also suggest that the crown structure based reduction is best applied prior to other reduction rules due to its efficiency but also because of its limitation in reducing dense graphs. Butenko et al. [12] further showed that simple reductions are also effective in reducing graph size in practice. They showed that a critical independent set, which is solvable in polynomial time, can be used to reduce graph size for the Maximum Independent Set problem. These reduction techniques are in general effective for the Minimum Vertex Cover problem but they fail to solve graphs with large *kernel* size, which is the size of a graph when no reduction rules can further remove vertices [31].

Branch-and-reduce algorithms [2, 4, 9, 23, 35, 39, 44] use a combination of branching, reducing, and pruning techniques. Akiba and Iwata's algorithm [2] is shown to be one of the most effective algorithms for the Minimum Vertex Cover problem. They showed that their implementation of branch-and-reduce is superior to other state-of-the-art solvers on many instances, even including some large graphs. However, Strash [38] showed that Akiba and Iwata's algorithm is only beneficial on a few networks with large kernels. For graphs with small kernels with respect to simple reductions, their algorithm can be outperformed by simple reductions combined with MCS. Overall exact algorithms, though computing optimal solutions, are still slow compared to heuristic approaches for most of the largest graphs [10], so heuristic approaches are still needed for the Minimum Vertex Cover and its related problems.

2.2 Heuristic Approaches

Many researchers have investigated heuristic (or inexact) approaches for the Maximum Clique problem [3, 5, 13, 14, 19, 21, 26, 27, 28] and they apply similar ideas of reducing the search space of a graph to a feasible size by removing or adding vertices. For example, Andrade et al. [3] proposed a fast implementation of a local search procedure for the Maximum Independent Set problem, which finds a near-optimal solution for sparse and large graphs such as road networks within a much shorter time than many state-of-the-art algorithms. However, it is outperformed by other methods such as the one by Lamm et al. [31] for web graphs and social networks with hundreds of millions to billions of nodes.

Dahlum et al. [19] showed that three issues exist in the algorithm by Andrade et al. which can be improved upon. First, during vertex selection for perturbation, the algorithm treats

all vertices the same including those with degree one which are always in some solution. It causes extra searching time which can be avoided by including the degree one vertices directly into the solution set. Second, high-degree vertices can be cut out of complex graphs to reduce time since they are less likely to be in the solution than other vertices. Third, a better implementation of "vertex pair selection" can also be achieved by limiting valid pairs to a small number.

San Segundo et al. [35] gave an exact bit-parallel algorithm which makes full use of the ability of CPUs to solve the Maximum Clique problem and further improved the algorithm by incorporating a recoloring strategy by Tomita et al. [41] and optimized bit scanning with the use of compiler intrinsic to compute basic bit string operations [16].

Lamm et al. [31] developed an evolutionary algorithm for the Maximum Independent Set problem. They repeatedly kernelize a graph with a combination of both exact reductions and selections of likely vertices to find a high quality solution. Their implementations not only speed up computations on large graphs but also obtain better results compared to the algorithm by Andrade et al. [3].

NuMVC, which is a heuristic algorithm by Cai et al. [14] is shown to outperform many other state-of-the-art local search algorithms on the Minimum Vertex Cover problem. It adopts two strategies to improve local search performance. First, it uses a vertex exchange strategy which reduces the complexity of selecting vertex and doing search. Second, it introduces an edge weight forgetting mechanism to further speed up the algorithm. Edge weighting is shown to be useful in local search because it separates uncovered vertices from covered vertices. The forgetting mechanism proposes that the weights assigned too far in the past are less useful in performing searches so they can be forgotten to improve the overall algorithm speed.

The FastVC algorithm introduced by Cai [13] also solves the Minimum Vertex Cover problem. For large graphs, the idea of FastVC is to swap vertices within the vertex cover and outside the vertex cover of a graph and calculate gain and loss along the way. At the end it returns a vertex cover with the maximum gain, which is also a high quality solution for the Minimum Vertex Cover problem. He showed that FastVC is much faster than NuMVC on large graphs and it also finds better solutions. For example, FastVC is able to find a high quality solution for the graphs that NuMVC can not even finish in a reasonable time.

Comparatively, exact algorithms can not achieve the same efficacy as heuristic approaches when dealing with complex graphs with large kernel sizes but heuristic methods only find locally optimal solutions. A method that is both efficient and fast is needed and thus we propose to further improve Akiba and Iwata's branch-and-reduce algorithm which has already shown great results on many large graphs.

3 Preliminaries

3.1 Graphs

A *graph* $G = (V, E)$ is a mathematical structure that represents pairwise interactions of objects. Each object in the graph is represented as a vertex $v \in V$ and each of the two related objects $(v, u) \in E$ are connected with an edge $e \in E$. We use V to denote all the vertices in the graph and E for all the edges. We use $N(v)$ to denote vertex v's open neighborhood, which are the vertices in the graph that are directly connected to v. The closed neighborhood $N[v] = N(v) \cup \{v\}$.

3.1.1 The Minimum Vertex Cover Problem

The Minimum Vertex Cover problem is a classical NP-hard problem in graph theory. NP-hard is the class of problems that are at least as difficult as any problem in NP. So far no one knows if these problems can be solved in polynomial time [25]. A vertex cover C is a set of vertices of a graph G such that each edge of G is incident to at least one vertex in C. In brief, it is a subset of vertices $C \subseteq V$ that *covers* all the edges. A minimum vertex cover is a vertex cover with minimum cardinality.

3.2 The Branch-and-Reduce Paradigm

Branch-and-reduce is a technique that computes an optimal solution by reducing graph size and selecting vertices to temporarily add to a growing solution. The branch-and-reduce algorithm by Akiba and Iwata [2] recursively computes many candidate solutions from different branching choices and in the end computes a minimum vertex cover. Initially we have a graph G, a current vertex cover X to store the current growing cover and a best vertex cover Y (the smallest vertex cover) formed so far to be the final result. We apply a set of reduction rules to reduce the size of G and during the reduction we also add vertices to X. The reduction set is a set of techniques that are used to remove or fold, forming

a smaller graph. They also help to identify vertices that are probably in some minimum vertex cover. Akiba and Iwata include simple reductions such as degree one, degree two, vertex folding and domination reductions [23], but also use advanced reductions such as the unconfined and twin reductions [44]. With repeated application of reduction rules, G is minimized to a *kernel*. Then we select a vertex v using a set of branching rules and remove v from G and add it to X. Since G's structure has been changed, we re-apply all reductions to reduce its size until it again reaches a kernel. After repeating the whole reducing and branching process until either G is empty or it reaches the lower bound, the algorithm compares current cover X with the best cover Y and update Y to be X if X has a smaller size. After updating Y, the algorithm recursively returns to the previous kernel from before it branched. This time it selects a different vertex to branch on and continues recursively until it finds another solution. Akiba and Iwata further experimented with three lower-bounding techniques: clique cover, LP relaxation and cycle cover [2] to check if the solution can be improved further (see Chapter 5). The whole algorithm's pseudocode is shown below in Algorithm 1.

3.3 Branching Rules

The branching rules by Akiba and Iwata include vertex selection, mirror branching, satellite branching and packing branching. Vertex selection [23] selects a vertex with the maximum degree to branch on and if there are multiple vertices with the same maximum degree then we choose one vertex v that minimizes the total degree of $N(v)$. Mirror branching is another technique by Fomin et al. [23]. It first finds a vertex v that has a *mirror*, which is a set of $u \in N^2(v)$ (v's neighbors' neighbors) such that $N(v) \setminus N(u)$ forms a clique or the empty set. Then we remove v from the graph and add all its mirrors to the vertex cover. Satellite branching is a different branching rule by Kneis et al. [29]. The idea is similar to mirror branching but we need to find a vertex v's *satellites*, which is a set of vertices $u \in N^2(v)$ such that for a $w \in N(v)$, $N(w) \setminus N[v] = \{u\}$. We then discard v's satellites S and include v and S in the vertex cover. The last branching rule, packing branching, is correlated with the *packing constraints*. A packing constraint indicates that if we want to find a minimum vertex cover that does not include vertex v, then for its neighbor $w \in N(v)$, we can add a set of constraints $\sum_{u \in N^+(w)} x_u \leq |N^+(w)| - 1$ ($N^+ = N(w) \setminus N[S]$) in which $x_u = 1$ if a vertex u is in the cover and $x_u = 0$ otherwise. These constraints are used to speed up the search for a branching candidate.

Algorithm 1 A branch-and-reduce algorithm for the Minimum Vertex Cover problem

INPUT Graph $G = (V, E)$, current solution X, best solution Y
$Solver(G)$
  while G is not a kernel **do**
    $G \leftarrow reduce(G)$ **AND** update X
  end while
  if G is not satisfied packing constraints **then**
    return X
  else if G is empty **then**
    return X
  else if X + LowerBound(G) > Y **then**
    return X
  else if G is disconnected **then**
    for each component $C \in G$ **do**
      $X \leftarrow X + Solve(C)$
    end for
    return X
  end if
  if $|X| \leq |Y|$ **then**
    $Y \leftarrow X$
  end if
  $G_1, G_2 \leftarrow branch(G)$
  $Y \leftarrow Solve(G_1)$
  $Y_2 \leftarrow Solve(G_2)$
  if $|Y_2| \leq |Y|$ **then**
    $Y \leftarrow Y_2$
  end if
  return Y

4 Dependency Checking

In this chapter, we discuss *dependency checking*, a technique to reduce time spent on reductions, and describe our implementation of this technique.

4.1 Dependency Checking Overview

Dependency checking is a technique first introduced by Strash [38]. He showed that simple reduction can be quickly applied when considering only parts of the graph that change. Inspired by his idea, we want to apply simple reductions rules on graphs as much as possible before using advanced reduction techniques in order to fully utilize simple reductions [37]. Simple reductions are the ones that are effective and quick to reduce graphs. These include the degree one reduction, the degree two reduction [23] and vertex folding [17]. Advanced reductions, on the other hand, are ones that are effective at reducing the graph size but are time-consuming, which includes the unconfined reduction and the twin reduction [44].

There are two reasons to use dependency checking. First, many graphs can be easily reduced to small kernels with just simple reductions [38], so it is unnecessary to use advanced reductions which output similar kernel sizes with much more time. Second, some advanced techniques such as the unconfined reduction are expensive so if we can reduce the graph size before using them, they can take less time to apply. Overall our idea is to reorder the way Akiba and Iwata apply their suite of reductions to achieve a speedup.

In order to implement our dependency checking strategy. We use our own version of a deque (double-ended queue) stored "circularly" in an array, which can only be pulled from one end at a time. Our deque has two *anchor* pointers pointing the two ends and is used to store vertices from a graph. It supports four operations: PushRight, PushLeft, PopRight and PopLeft. We will refer the two anchor pointers as left and right. The section of the deque before the left pointer is the left end and the right section after the right pointer is the right end (see Figure 4.1). With our deque we store vertices that are candidates for simple reductions in the left end, and store the rest vertices for advanced techniques in the right end. We first populate the deque with all vertices in a graph, so we can pop them and apply reductions in order later. If a vertex is eligible for our simple reductions, we push it to

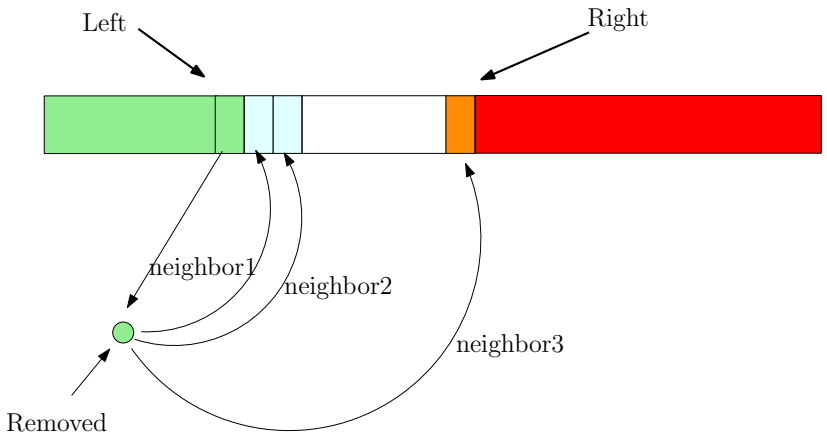

Figure 4.1: Pulling a vertex from the left end, removing it from the graph and adding its neighbors back to the deque

the left end, else we push it to the right end. We want to pop as many vertices from the left as possible because we want to fully utilize simple reductions to reduce graph before applying advanced reductions. Along the process we add the neighbors of the removed vertices back to the left end if they are eligible for our simple reductions, and we push the rest neighbors to the right end. If the left end is empty, we pop one vertex from the right and apply advanced reductions. If the popped vertex can be removed from the graph with our advanced reductions, we add its neighbors to our deque based on the same rules as before, otherwise we do nothing.

We use degree one, degree two and vertex folding [23] as the suite of simple reductions and use the unconfined reduction [44] as our advanced reduction. There are two main reasons for these choices. First, based on our preliminary experiments with Akiba and Iwata's branch-and-reduce algorithm, degree one, degree two, vertex folding and unconfined are the most effective reductions among all other reductions: they successfully reduce graphs the most. Secondly, after breaking down algorithm running time, we found that the unconfined reduction on average takes half the total running time. For instance, the overall running time for the `NotreDame` web graph [2] is around 17 seconds but more than 8 seconds are spent on running unconfined reductions. By these observations, we think degree one, degree two, vertex folding and unconfined are the most critical reductions in the branch-and-reduce algorithm. In addition, since degree one, degree two, and vertex folding reductions on average are much less time-consuming than the unconfined reduction, we use them as our simple reduction suite and leave unconfined alone as the advanced technique. We now explain these reduction rules and walk through an example of our dependency checking in the following sections.

4.2 Reduction rules

We now explain the main reduction rules used in our dependency checking strategy.

Degree One: For any vertex $v \in V$, if $|\{N(v)\}| \leq 1$, then we know there must exist a minimum vertex that does not have v. We thus include v's neighbor $n \in N(v)$ in our vertex cover and remove both n and v from G.

Degree Two: If a vertex has 2 neighbors connected by an edge, then both those neighbors are in some minimum vertex cover.

Vertex Folding: For any vertex v which only has two neighbors n_1 and n_2 and there is no edge between them, we remove $N[v]$ from the graph G and introduce a new vertex v' to connect with vertices that are distance two away from v, $N(n_1)$ and $N(n_2)$. In doing that, we change the graph from G to G' with two fewer vertices, and for any minimum vertex cover C' in G', if C' contains v', then a minimum vertex cover C for the original graph G is equal to $C' \cup \{v\}$, else C is equal to $(C' \setminus \{v'\}) \cup N(v)$.

Unconfined: There are three steps in the unconfined reduction. For any $v \in V$, first initialize $S = \{v\}$. Second, go through all $u \in N(v)$. If $|N(u) \cap S| = 1$, meaning u is only connected to one vertex in S and $|N(u) \setminus N[S]|$ is minimized, we move to next step, otherwise the vertex is not unconfined and we are finished. Third, if $N(u) \setminus N[S] = \emptyset$, then v is an unconfined vertex and we add it to our vertex cover. If $N(u) \setminus N[S]$ only has one vertex w, we go back to the second step and add w to S. If $N(u) \setminus N[S]$ has more than one vertex, we know v is not an unconfined vertex so we move on to test next vertex in G.

Other Rules: When our deque is empty, we run the remaining reductions from Akiba and Iwata's implementation. The reduction set includes the LP-based reduction [34], the twin reduction [43], the alternative reduction [43] and a set of packing reductions [2].

4.3 Updating Vertices

If we successfully remove a vertex v from graph, based on different reduction rules, we either add v to our vertex cover and remove its neighbors $N(v)$, or we remove v and add its neighbors to the cover instead. Either way the degrees of $v's$ two degree neighbors $N^2(v)$ are reduced by one. We then loop through all $u \in N^2(v)$ and if $|N(u)| \leq 2$, we add it to the left end of our deque, else we add it to the right. We further expand this technique to all other reductions in order to add more candidates for our three reductions. If any reduction in the algorithm removes a vertex from the graph, we apply the same rule to add its two degree neighbors to the deque. This way we can more efficiently utilize our deque to achieve a better reduction performance.

4.4 Our Dependency Checking Algorithm

Given a graph $G = (V, E)$, we first check all $v \in V$ and add v to the left end of deque if $|N(v)| \leq 2$ or otherwise add v to the tail. We then pull one vertex v from the left end, apply degree one reduction on it if $N(v) \leq 1$ or degree two vertex folding reduction if $|N(v)| = 2$. Either way $N^2(v)$ have changed since we remove v from the G. For $n \in N(v)$, if $|N(n)| \leq 2$, we add n to the left end, otherwise add it to the right end. We pull vertices from the left end until the left end is empty, then we pull a vertex v' from the right end. If v' can be removed under the unconfined reduction, then we add v's neighbors to the deque based on the same criterion discussed above.

Algorithm 2 Dependency Checking

```
Graph G = (V, E), deque
while ! deque.isEmpty() do
    while v ∈ deque such that |N(v)| ≤ 2 do
        if |N(v)| ≤ 1 and deg-one(G) then
            add N(v) to deque
        else if |N(v)| = 2 and deg-two(G) then
            add N(v) to deque
        end if
    end while
    if unconfined(G) then
        add N(v) to deque
    end if
end while
return
```

4.5 Potential Issues

There are potential issues associated with our dependency checking technique. First, our deque will keep applying degree one, degree two vertex folding and unconfined reductions as long as one of them is still reducing the graph, while other reduction rules are deprived the chance of being applied. So if the three reductions we choose are not effective on a particular graph, continuously applying them will only slow our algorithm down. Second, giving these reductions higher priorities than any other reductions might change a graph's structure which makes other reductions less likely to apply. For example, if a graph's original

structure can be reduced significantly by another rule such as the twin reduction [43] in the reduction set, but we change the graph structure drastically in our dependency checking, then twin reduction's effectiveness might be decreased.

5 Pruning

In this chapter, we discuss what pruning is, our implementation of pruning techniques and why they help to improve the branch-and-reduce algorithm.

5.1 Pruning Overview

When we conduct a recursive or backtracking algorithm to find an optimal solution, we do not need to branch on all vertices. The idea is that through bound checking, we can find that some recursive calls will only generate a solution no better than current best solution, so it is unnecessary to waste time on them. We add additional idea to pruning besides bound checking. Through caching process we can also save time on computing identical sub graphs.

Many researchers have implemented pruning techniques for the Minimum Vertex Cover problem and its related problems [20, 40, 41]. One of the first pruning ideas was introduced by Carraghan and Pardalos [15] for the Maximum Clique problem. In their algorithm, they introduced the notion of depth and used it to prune unneeded branches. They found that the technique helped them to reduce running time drastically for graphs with high density. Tomita et al. [40] in their MCR algorithm used approximate graph coloring to both prune and branch. They used a greedy coloring algorithm and assigned a different color (with numbers) to vertices, then they sort vertices by color to decide branching order. Tomita et al. [41] further added recoloring of vertices in their pruning and managed to improve the speed of their algorithm on even dense graphs.

Batsyn et al. [4] implemented the idea to feed a high quality solution (found using local search) to MCS to solve the Maximum Clique problem and they found that the algorithm is able to compute "difficult" graphs from DIMACS instances as much as 11 000 times faster than MCS. In Akiba and Iwata's original algorithm, they use lower bound checks to avoid some branches on computing subgraphs but they did not prune the main branch. In this paper we incorporate the idea from Batsyn et al.'s algorithm into a branch-and-reduce algorithm for the Minimum Vertex Cover problem and also develop our own pruning technique, which we now describe.

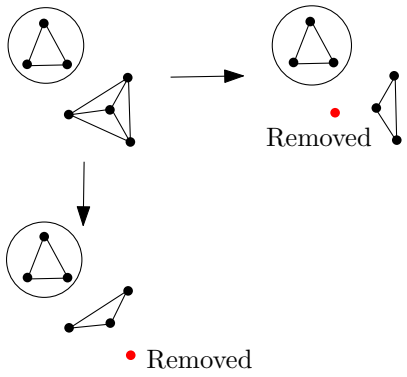

Figure 5.1: The triangle component never changes, so we can compute its solution once and store it

5.2 Cache Subgraphs

In Akiba and Iwata's original algorithm [2], they detect when the graph becomes disconnected and solve components individually, then combine their solutions together into the final result. For instance, in a graph $G = (V, E)$ there is a set of components C. For each $c \in C$ if branching or reducing breaks c into two new components c_1 and c_2, the original algorithm solves c_1 and c_2 separately to find their minimum vertex covers and make the union of the two results a minimum cover for c. This strategy is inefficient when there are many disconnected components in G, where there will be at most one c whose structure is changed at each vertex branching. However, we nevertheless have to calculate the minimum vertex covers for all $c \in C$. There is a lot of repeated work in these computations, especially for sparse graphs which may have many components, so our idea is to cache solutions of previously encountered components and, if these parts never change, we use the cached solutions without computing again (see Figure 5). If a component is modified and we have new additional subgraphs, then we compute their solutions and add them to our cached solution set.

In order to implement this idea, we use a mapping technique to store previous components' solutions. Each component c is given an id which is equivalent to the smallest vertex number in the component. Previous subgraph solutions are also indexed based on their component id and they are stored in an array at positions equal to their ids. When a new set of components C' is generated after branching, we check $c' \in C'$ with old $c \in C$, for each c' that is the same as c, we just reuse the existing solution. Thus we only solve the components whose structures are changed. This technique can improve the original algorithm on graphs in which solving subgraphs is time consuming or there are many unique disconnected components.

There are some potential issues with our caching. First, we add overhead of comparing components. If a graph's components can be easily solved then this comparison is unnecessary and will only slow the algorithm down. Second, in order to compare cached solutions with new components more efficiently, we store cached solutions based on its component id, which can be as large as the graph size (component id is defined by a vertex id). We allocate more space than the total number of components in a graph to avoid array index out of bounds errors, so inefficient use of space is another shortfall of our caching technique.

5.3 Using an Initial High Quality Solution

Lower bound checking is used in the original algorithm to avoid unnecessary branching. For a component c in a graph G, we can calculate whether it is worth solving it by conducting a lower bound check. If we are able to successfully compute a lower bound size in c, then we can add it with our current solution set and compare with the size of our best solution. If it has a larger size, then we know it is unnecessary to solve c. Akiba and Iwata used three different lower bound checking techniques, we briefly explain below.

5.3.1 Lower bound checking techniques

Clique Cover: The first one uses a clique cover. A clique C is a subset of a graph $G = (V, E)$ such that for every two $v_1, v_2 \in C$, there exists an edge $e = (v_1, v_2) \in E$. A clique cover is just a set of disjoint cliques $C_1, ..., C_k$ which cover all the vertices in G, we can use $\sum_{i=1}^{k} (|C_i| - 1) = |V| - k$ to calculate a lower bound.

LP Relaxation: The second technique is based on linear programming (LP) which is associated with the LP-based reduction in the algorithm. After applying the reduction, the remaining graph admits a half integral optimal solution of value $\frac{|V|}{2}$ [2].

Cycle Cover: The third technique is from a cycle cover. A cycle is a set of vertices which are connected to form a closed simple cycle and each vertex v in the graph has exactly degree two. A cycle cover is a disjoint set of cycles $C_1, ..., C_k$ which cover all the vertices in a graph G. We can use $\sum_{i=1}^{k} \frac{|C_i|}{2}$ as a lower bound.

5.3.2 Why a initial solution?

Although these lower bound techniques are helpful in solving subgraph solutions, they

are not useful until we first have a solution for the current subproblem being evaluated in recursive search. A high quality initial solution is effective in upper-bound checking for the branch-and-bound algorithm MCS, as shown in experiments by Batsyn et al. [4]. If we can also help our algorithm to conduct a lower bound check before it even starts solving a subgraph the first time it is encountered, then we would expect improved performance. However,since our goal is to output a minimum vertex cover of a graph, how do we feed a solution to it prior to running the algorithm? Our idea is to use a heuristic solution such as the one by Dahlum et al. [19] which is near-optimal and can be obtained in less than a few seconds.

6 Experimental Results

In this chapter, we discuss our experimental setup, data sets, and results

6.1 Experimental Setup

6.1.1 Machine and Language

The machine we use is equipped with 2 6-core Intel Xeon E7540 processors running at 2.0 GHz. It has 64 GB local memory, 18 MB L3-Cache and 256 KB L2-Cache. The operating system we use is 64-bit Ubuntu 16.04.2 LTS, with Linux kernel version 4.4.0-31.

The language we use is Java with JDK version 1.8.0, with no special compiler flags.

In our experiment, we use the original Java Implementation of Akiba and Iwata [1]

6.1.2 Graphs

Our datasets include real world sparse networks such as social networks, web graphs, road networks and computer networks obtained from the Stanford Large Network Dataset [33], the Koblenz Network Collection [30] and Laboratory for Web Algorithmics [6, 7, 8].

6.1.3 Algorithms Compared

We compare the original branch-and-reduce algorithm with our four implementations. First, with just dependency checking technique. Second, with just the caching technique. Third, with just feeding initial solution and last, with both dependency checking and feeding initial solution.

[1] `https://github.com/wata-orz/vertex_cover`

6.2 Results

6.2.1 Summary Statistics

There are 47 graphs in our dataset, which we summarize in Table 6.1. The largest graph based on number of vertices is the `LiveJournal` network which has 4 847 571 vertices and 42 851 237 edges. The smallest graph based on vertices is `ca-GrQc` graph and it has 5 242 vertices with 14 484 edges. The largest graph based on edges is `hollywood-2011` with 114 492 816 edges and 1 985 306 vertices, and the smallest one is also the `ca-GrQc` graph. The average number of vertices of all the graphs is 858 391 vertices, so most of them are quite large. They also on average have 9 500 530 edges. Most of the graphs are sparse and the most dense graph is `wiki-Vote` with 7 115 vertices, 100 762 edges and a density of 0.004. The most sparse graph is the `wiki-Talk` graph with a density of 1.6×10^{-6}.

6.2.2 Results of the Original Algorithm

The original algorithm of Akiba and Iwata is efficient on most of the graphs (see Table 6.2). 39 graphs out of all 47 finish within 10 seconds. The datasets `p2p-Gnutella08`, `p2p-Gnutella05`, `p2p-Gnutella06` and `Wiki-Vote` are solved almost instantly with running time of 0.01, 0.012, 0.012 and 0.014 seconds. They are in general small graphs with less than 10 000 vertices. The slowest one is the `Stanford` web graph with a running time of 58 983 seconds (16 hours). It has 281 903 vertices, 1 992 636 edges and a minimum vertex cover size of 118 513. The `as-skitter` graphs is also slow. It takes the original algorithm 2 988 seconds (50 minutes) to compute a minimum vertex cover of 525 835 vertices. Other slow graphs include `libimseti`, `web-BerkStan`, `hollywood-2011` graphs with running times of 2 126 seconds (35 minutes), 203 seconds and 76 seconds respectively. These slow graphs are all relatively large with more than 500 000 vertices, except for `libimseti` and `Stanford`, which have around 200 000 vertices.

6.2.3 Results with Dependency Checking

With dependency checking, the fastest solved graphs are `ca-GrQc`, `p2p-Gnutella08` with a running time around 0.04 seconds (see Table 6.2). The originally slow graphs such as the `Stanford` web graph and the `as-skitter` graph are all solved faster with dependency checking. We solve the `Stanford` graph 3.5 times faster than the original algorithm, and it only takes 4.6 hours to compute a minimum vertex cover instead of the former 16 hours. We also achieve a speedup of 1.6 on the `as-skitter` graph, reducing the running time from

graph	vertices	edges	density
as-skitter	1696415	11095298	0.00001
ca-AstroPh	18772	198050	0.00112
ca-CondMat	23133	93439	0.00035
ca-GrQc	5242	14484	0.00105
ca-HepPh	12008	118489	0.00164
ca-HepTh	9877	25973	0.00053
dblp-2010	300647	807700	0.00002
dblp-2011	933258	3353618	0.00001
email-Enron	36692	183831	0.00027
email-EuAll	265214	364481	0.00001
flickr-growth	2302925	22838276	0.00001
flickr-links	1715255	15555041	0.00001
hollywood-2009	1107243	56375711	0.00009
hollywood-2011	1985306	114492816	0.00006
in-2004	1382867	13591473	0.00001
p2p-Gnutella04	10876	39994	0.00068
p2p-Gnutella05	8846	31839	0.00081
p2p-Gnutella06	8717	31525	0.00083
p2p-Gnutella08	6301	20777	0.00105
p2p-Gnutella09	8114	26013	0.00079
p2p-Gnutella24	26518	65369	0.00019
p2p-Gnutella25	22687	54705	0.00021
p2p-Gnutella30	36682	88328	0.00013
p2p-Gnutella31	62586	147892	0.00008
soc-Epinions1	75879	405740	0.00014
soc-LiveJournal1	4847571	42851237	0.00000
soc-Slashdot0811	77360	469180	0.00016
soc-Slashdot0902	82168	504230	0.00015
web-BerkStan	685230	6649470	0.00003
web-Google	875713	4322051	0.00001
web-NotreDame	325729	1090108	0.00002
wiki-Talk	2394385	4659565	0.00000
wiki-Vote	7115	100762	0.00398
youtube-links	1138499	2990443	0.00000
youtube-u-growth	3223643	9376594	0.00000
flickr-growth	2302925	22838276	0.00001
flickr-links	1715255	15555041	0.00001
libimseti	220970	17233144	0.00071
petster-friendships-cat-uniq	149700	5448197	0.00049
petster-friendships-dog-uniq	426820	8543549	0.00009
youtube-links	1138499	2990443	0.00000
youtube-u-growth	3223643	9376594	0.00000
zhishi-baidu-internallink	2141300	17014946	0.00001
zhishi-baidu-relatedpages	415641	2374044	0.00003
zhishi-hudong-internallink	1984484	14428382	0.00001
petster-carnivore	623766	15695166	0.00008
web-Stanford	281903	1992636	0.00005

Table 6.1: Summary Table (density is rounded to 5 decimal points)

2 988 seconds to 2 308 seconds (38 minutes). There are also significant speedups for both hollywood graphs. The `hollywood-2009` graph is solved is 2.8 times faster, with its running time reduced from 43 seconds to 15 seconds. The `hollywood-2011` instance is also solved 2.5 times faster with a new running time of 30 seconds compared to previous 76 seconds. However, the running time of `libimseti` stays relatively unchanged and is even 50 seconds (2%) slower. We hypothesize that it can be due to small variations of running time and overhead of pushing and popping vertices. Surprisingly the `web-BerkStan` graph is 184 seconds (90%) slower with dependency checking. We hypothesize that the `web-BerkStan` is better solved with other reductions (see Chapter 4). In addition, most of the previous quickly solved graphs are slightly slower compared to the original algorithm, but it is expected since we add overhead of pushing and pulling from a deque. Overall most of the former slow graphs have been sped up with our technique.

6.2.4 Results with Caching

Our pruning technique does not reduce the overall running time by a large scale but there is also no instance with significantly worse performance (unlike the `web-BerkStan` graph under dependency checking). The fastest solved graph is the `p2p-Gnutella08` graph with a running time of 0.009 seconds, which is 0.001 seconds faster than the original algorithm. The `p2p-Gnutella` graphs in general take the least amount of time but they are not improved under pruning because they are also solved instantly under the original algorithm. One of the largest speedups by real value (seconds) is the `web-BerkStan` graph which reduced from 203 seconds to 195 seconds. The `hollywood-2009` graph is also slightly faster with a running time reduction of 4 seconds. Most of the graphs are solved with the same running time as the original algorithm and it seems pruning is not particularly effective with the graphs in our experiments.

6.2.5 Results with a High-Quality Initial Solution

Giving a high-quality initial solution always works well. 40 out of the 45 graphs are solved faster when feeding a initial exact solution and the other five's running times are at most 1 second slower than the original algorithm. The effect is especially prominent on graphs that are quickly solved. It takes the original algorithm 0.012 seconds to solve the `p2p-Gnutella06` graph and it only needs 0.005 seconds to compute a minimum vertex cover with this technique. There are also performance improvements on previously slow graphs. For instance, it takes the original algorithm 17 seconds to compute a minimum vertex cover for the `NotreDame` graph but with a initial solution, the running time is reduced

to 14 seconds. Also, for the `as-skitter` graph, the running time is also cut down by 124 seconds from 2 988 to 2 864 seconds. However, the performance overall is not as pronounced as with the dependency checking technique.

6.2.6 Results with Dependency Checking and a High Quality Initial Solution

When we combine dependency checking with a high-quality initial solution, the algorithm further improves on previously slow graphs. For instance, we achieve a speedup of 3.4 for the `hollywood-2009` graph with a running time cut from 43 seconds to 12 seconds. This is 2 seconds faster than with just dependency checking. We also solve both the `as-skitter` and `Stanford` graph faster. The `Stanford` graph is solved 3.5 times faster, from 58 983 seconds (16 hours) to 16 768 seconds, reduced 53 seconds compared to with just dependency checking. This indicates that both techniques can be effective on graphs but since feeding an initial solution is shown to only affect algorithm performance slightly, the major running time reduction is still due to dependency checking.

6.2.7 Issues Combining Techniques

Combining dependency checking and subproblem caching currently gives incorrect results for some of the graphs such as the `Stanford` network. We hypothesize the reason to be that some structures of the graphs are changed in our dependency checking process but they are replaced by the cached solutions. In the future, we need a better monitoring technique of graph structures and also conduct a better comparison to find candidates for the cached solutions.

6.2.8 Results conclusion

Our dependency checking technique together with a high-quality initial solution shows the highest efficacy in improving the algorithm by Akiba and Iwata. Originally slow graphs such as the `Stanford` and `as-skitter` graphs are solved much faster. This is especially true for the `Stanford` graph, which requires around 16 hours to find a minimum vertex cover with the original algorithm, only needs 4.6 hours under our implementation. Although many previously fast-to-compute graphs have been slightly slowed down due to overhead of manipulating our deque, they are still being solved quickly with 33 of the 47 graphs finishing within 10 seconds. We have also addressed the reason of the slowness which can be the graphs are not particularly desirable for degree one, degree two folding and unconfined reductions. Caching and feeding an initial solution does not seem to improve

Graph		Original	Init	DepChecking	DepChecking/Init	Pruning	Pruning/Init
	result	time	time	time	time	time	time
as-skitter	525835	2988.28	2864.37	2308.52	**2286.68**	2918.92	2861.17
ca-AstroPh	12012	0.09	0.09	0.11	0.06	0.08	**0.04**
ca-CondMat	13521	0.05	**0.02**	0.09	0.03	0.05	**0.02**
ca-GrQc	2783	0.03	**0.01**	0.03	**0.01**	0.03	**0.01**
ca-HepPh	7012	0.06	0.06	0.08	**0.04**	0.06	0.06
ca-HepTh	4981	0.04	0.02	0.05	**0.01**	0.05	**0.01**
dblp-2010	166234	0.48	0.44	0.53	**0.24**	0.48	0.36
dblp-2011	498969	1.63	1.20	1.66	1.14	1.55	**0.97**
email-Enron	14437	0.12	**0.04**	0.12	0.05	0.12	**0.04**
email-EuAll	18316	0.12	**0.09**	0.26	0.10	0.11	**0.09**
flickr-growth	640921	2.00	1.73	9.05	6.80	1.82	**1.65**
flickr-links	474637	1.38	1.29	6.99	3.90	1.35	**1.21**
hollywood-2009	890039	43.17	46.01	15.00	**12.60**	39.15	44.91
hollywood-2011	1657357	76.56	77.42	30.68	**26.67**	78.72	72.66
in-2004	486146	6.58	3.76	62.82	48.17	6.09	**3.46**
p2p-Gnutella04	4348	**0.01**	**0.01**	0.06	0.02	**0.01**	**0.01**
p2p-Gnutella05	3428	**0.01**	**0.01**	0.05	0.02	**0.01**	**0.01**
p2p-Gnutella06	3405	**0.01**	**0.01**	0.05	0.02	**0.01**	**0.01**
p2p-Gnutella08	2054	**0.01**	**0.01**	0.04	0.01	**0.01**	**0.01**
p2p-Gnutella09	2574	**0.02**	**0.01**	0.05	0.02	0.02	0.02
p2p-Gnutella24	7209	0.02	**0.01**	0.09	0.02	0.02	**0.01**
p2p-Gnutella25	6017	0.02	**0.01**	0.08	0.04	0.02	**0.01**
p2p-Gnutella30	9268	0.03	**0.01**	0.10	0.05	0.03	0.02
p2p-Gnutella31	15693	0.04	**0.02**	0.16	0.05	0.04	**0.02**
soc-Epinions1	22280	0.08	**0.03**	0.18	0.09	0.08	**0.03**
soc-LiveJournal1	2215668	11.01	11.73	37.41	22.74	10.85	**9.01**
soc-Slashdot0811	24046	0.08	**0.03**	0.24	0.09	0.08	**0.03**
soc-Slashdot0902	25770	0.10	0.04	0.25	0.09	0.10	**0.03**
web-BerkStan	276748	203.63	201.63	387.98	384.01	**195.62**	200.61
web-Google	346575	1.19	0.76	2.42	1.52	1.24	**0.69**
web-NotreDame	73878	17.15	14.83	15.54	**12.47**	17.73	14.89
wiki-Talk	56163	0.66	0.57	1.54	1.55	0.75	**0.53**
wiki-Vote	2249	**0.01**	**0.01**	0.08	0.04	0.02	**0.01**
youtube-links	278125	0.58	0.48	1.16	0.72	0.69	**0.36**
youtube-u-growth	840060	1.58	**1.53**	5.14	2.55	1.56	1.58
flickr-growth	640921	1.76	**1.42**	10.06	6.43	1.94	1.72
flickr-links	474637	1.36	**0.97**	6.58	4.46	1.59	1.27
libimseti	93676	2126.17	**2099.70**	2176.82	2196.89	2166.36	2177.65
petster-friendships-cat-uniq	41103	4.16	**2.84**	20.94	16.42	4.61	**2.82**
petster-friendships-dog-uniq	150119	6.12	**5.05**	17.88	16.84	6.68	6.14
youtube-links	278125	0.54	0.49	1.20	0.60	0.63	**0.40**
youtube-u-growth	840060	1.55	1.55	4.61	2.08	1.53	**1.50**
zhishi-baidu-internallink	636967	**1.66**	1.74	6.06	4.77	1.73	2.31
zhishi-baidu-relatedpages	142667	1.84	**1.22**	3.19	1.66	1.97	1.35
zhishi-hudong-internallink	502742	1.63	1.42	11.72	6.24	1.71	**1.27**
petster-carnivore	371189	4.41	3.91	19.77	14.84	4.20	**3.88**
web-Stanford	118513	58983.33	58946.04	**16768.45**	16821.76	59560.83	60282.17

Table 6.2: Time (in seconds) to solve each instance

the algorithm performance much, but they do not harm the performance either. Giving a high-quality initial solution even speeds up solving 40 of 45 graphs although the effect is very small. Caching on the other hand, shows a potential to further reduce running time on large graphs such as `web-BerkStan`, but our current implementation does not give a significant improvement. We summarize the results of all algorithms in Table 6.2 and an algorithm speedup chart is shown in Figure 6.1. It includes the slowest graphs in our experiments which are `web-NotreDame`, `as-skitter`, `libimseti`, `web-BerkStan`, `Stanford`, `hollywood-2009`, `hollywood-2011` and `soc-LiveJournal1`. It is also obvious from the chart that dependency checking and dependency checking with an initial solution have the best running time improvement among other techniques.

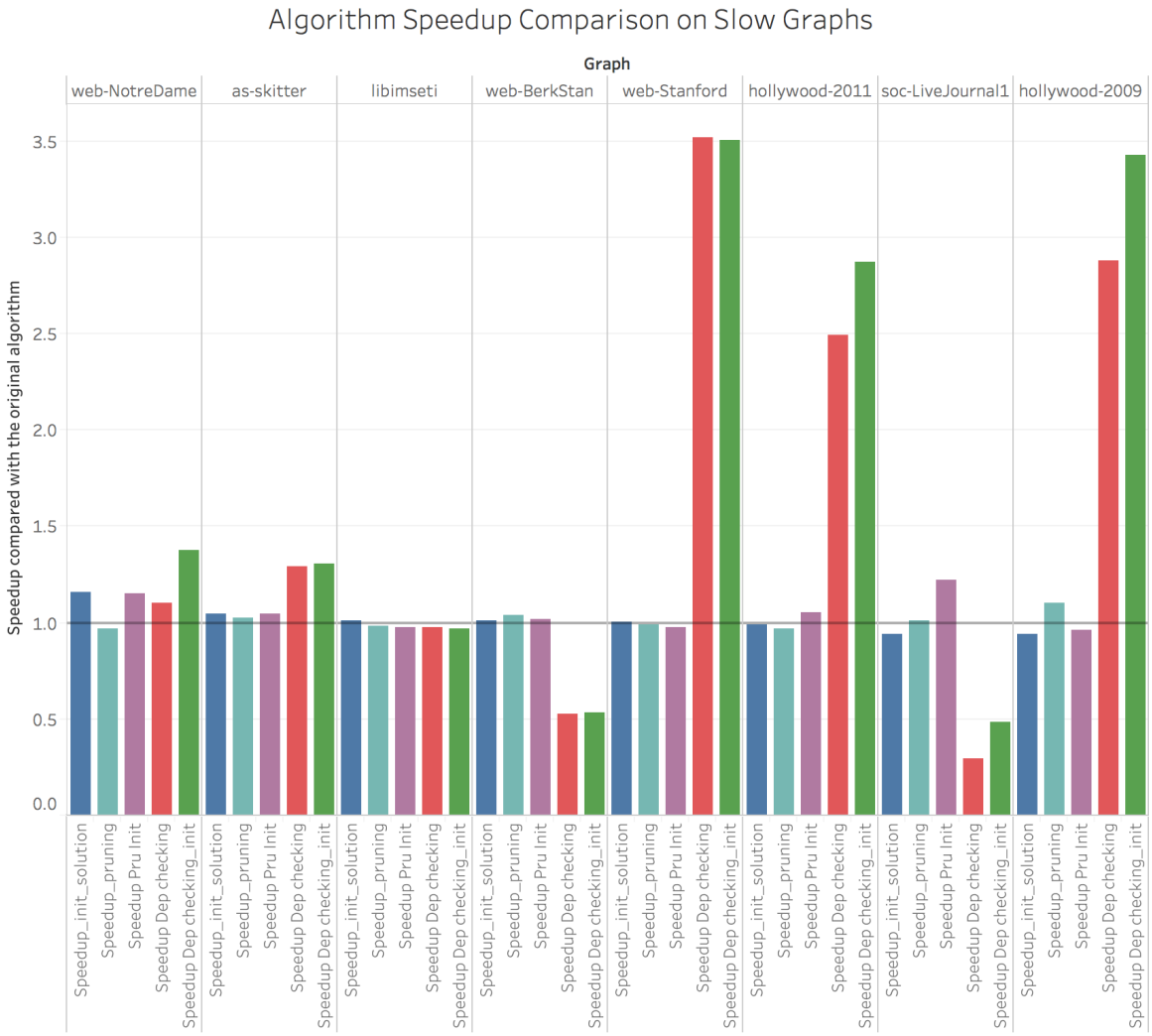

Figure 6.1: A summary of speedups for each technique over the original algorithm of Akiba and Iwata for slow instances: those taking more than 10 seconds with the original algorithm

7 Conclusion

7.1 Contributions

Through our implementations, many of the previous graphs, on which Akiba and Iwata's algorithm is slow, have been solved faster. Dependency checking together with a high-quality initial solution is the most effective on these slow graphs. Alone, giving an high-quality initial solution has also proven to have high efficacy on most of the graphs, regardless of whether they are fast or slow with the original algorithm. Although caching is not as effective as we expected, we are still able to show its potential on a few graphs such as `web-BerkStan` graph and `LiveJournal1`.

7.2 Future Work

An obvious direction for future work is to focus on effectively combining all three of the techniques discussed in this paper. We would like to see whether combining the techniques gives an even greater speedup. Also, dependency checking has clearly showed its power on many "difficult" graphs with significant speed improvements. We might be able to further explore this technique with adding additional reductions into our dependency checking scheme.

Bibliography

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

[2] T. Akiba and Y. Iwata. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theoretical Computer Science*, 609, Part 1:211–225, 2016.

[3] D. V. Andrade, M. G. Resende, and R. F. Werneck. Fast local search for the maximum independent set problem. *Journal of Heuristics*, 18(4):525–547, 2012.

[4] M. Batsyn, B. Goldengorin, E. Maslov, and P. M. Pardalos. Improvements to mcs algorithm for the maximum clique problem. *Journal of Combinatorial Optimization*, 27(2):397–416, 2014.

[5] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem 1. *Algorithmica*, 29(4):610–637, 2001.

[6] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*, pages 587–596. ACM, 2011.

[7] P. Boldi and S. Vigna. Laboratory for web algorithmics datasets.

[8] P. Boldi and S. Vigna. The WebGraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[9] N. Bourgeois, B. Escoffier, V. T. Paschos, and J. M. van Rooij. Fast algorithms for max independent set. *Algorithmica*, 62(1-2):382–415, 2012.

[10] S. Butenko, P. Pardalos, I. Sergienko, V. Shylo, and P. Stetsyuk. Finding maximum independent sets in graphs arising from coding theory. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 542–546. ACM, 2002.

[11] S. Butenko, P. Pardalos, I. Sergienko, V. Shylo, and P. Stetsyuk. Estimating the size of correcting codes using extremal graph problems. In *Optimization*, pages 227–243. Springer, 2009.

[12] S. Butenko and S. Trukhanov. Using critical sets to solve the maximum independent set problem. *Operations Research Letters*, 35(4):519–524, 2007.

[13] S. Cai. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *IJCAI*, pages 747–753, 2015.

[14] S. Cai, K. Su, C. Luo, and A. Sattar. Numvc: An efficient local search algorithm for minimum vertex cover. *Journal of Artificial Intelligence Research*, 46:687–716, 2013.

[15] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9(6):375–382, 1990.

[16] P. S. S. Carrillo, F. M. Espada, D. R.-L. González, and M. H. Gutiérrez. An improved bit parallel exact maximum clique algorithm. 2011.

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

[18] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 457–468. ACM, 2012.

[19] J. Dahlum, S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. Accelerating local search for the maximum independent set problem. In *International Symposium on Experimental Algorithms*, pages 118–133. Springer, 2016.

[20] T. Fahle. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. *Algorithms—ESA 2002*, pages 47–86, 2002.

[21] Y. Fan, C. Li, Z. Ma, L. Brankovic, V. Estivill-Castro, and A. Sattar. Exploiting reduction rules and data structures: Local search for minimum vertex cover in massive graphs. *arXiv preprint arXiv:1509.05870*, 2015.

[22] T. A. Feo, M. G. Resende, and S. H. Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42(5):860–878, 1994.

[23] F. V. Fomin, F. Grandoni, and D. Kratsch. A measure & conquer approach for the analysis of exact algorithms. *Journal of the ACM (JACM)*, 56(5):25, 2009.

[24] J. Gajarskỳ, P. Hliněnỳ, J. Obdržálek, S. Ordyniak, F. Reidl, P. Rossmanith, F. S. Villaamil, and S. Sikdar. Kernelization using structural parameters on sparse graph classes. In *European Symposium on Algorithms*, pages 529–540. Springer, 2013.

[25] M. R. Gary and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

[26] A. Grosso, M. Locatelli, and F. Della Croce. Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem. *Journal of Heuristics*, 10(2):135–152, 2004.

[27] P. Hansen, N. Mladenović, and D. Urošević. Variable neighborhood search for the maximum clique. *Discrete Applied Mathematics*, 145(1):117–125, 2004.

[28] K. Katayama, A. Hamamoto, and H. Narihisa. An effective local search for the maximum clique problem. *Information Processing Letters*, 95(5):503–511, 2005.

[29] J. Kneis, A. Langer, and P. Rossmanith. A fine-grained analysis of a simple independent set algorithm. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[30] J. Kunegis. Konect: the Koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[31] S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. Finding near-optimal independent sets at scale. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 138–150. SIAM, 2016.

[32] G. Lancia, V. Bafna, S. Istrail, R. Lippert, and R. Schwartz. SNPs problems, complexity, and algorithms. In *European Symposium on Algorithms*, pages 182–193. Springer, 2001.

[33] J. Leskovec and A. Krevl. SNAP Datasets:Stanford large network dataset collection. 2015.

[34] G. L. Nemhauser and L. E. Trotter. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975.

[35] P. San Segundo, D. Rodríguez-Losada, and A. Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

[36] P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient traversal of mesh edges using adjacency primitives. In *ACM Transactions on Graphics (TOG)*, volume 27, page 144. ACM, 2008.

[37] D. Strash. personal communication.

[38] D. Strash. On the power of simple reductions for the maximum independent set problem. In *International Computing and Combinatorics Conference*, pages 345–356. Springer, 2016.

[39] R. E. Tarjan and A. E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[40] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global optimization*, 37(1):95–111, 2007.

[41] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. *A simple and faster branch-and-bound algorithm for finding a maximum clique*, pages 191–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[42] E. Tomita and M. Yamada. An algorithm for finding a maximum complete subgraph. In *Conference Records of the National Convention of IECE*, volume 8, page 1978, 1978.

[43] M. Xiao and H. Nagamochi. Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs. *Theoretical Computer Science*, 469:92–104, 2013.

[44] M. Xiao and H. Nagamochi. Exact algorithms for maximum independent set. In *International Symposium on Algorithms and Computation*, pages 328–338. Springer, 2013.