A Hybrid Graph Representation for Recursive Backtracking Algorithms

F. N. Abu-Khzam[1]*, M. A. Langston[2], A. E. Mouawad[1] and C. P. Nolan[2]**

[1] Department of Computer Science and Mathematics
Lebanese American University
Beirut, Lebanon
{faisal.abukhzam, amer.mouawad}@lau.edu.lb
http://www.csm.lau.edu.lb/fabukhzam

[2] Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, TN 37996-3450 USA
{langston, nolan}@eecs.utk.edu
http://www.cs.utk.edu/~langston

Abstract. Many exact algorithms for $\mathcal{NP}$-hard graph problems adopt the old Davis-Putman branch-and-reduce paradigm. The performance of these algorithms often suffers from the increasing number of graph modifications, such as deletions, that reduce the problem instance and have to be "taken back" frequently during the search process. The use of efficient data structures is necessary for fast graph modification modules as well as fast take-back procedures. In this paper, we investigate practical implementation-based aspects of exact algorithms by providing a hybrid graph representation that addresses the take-back challenge and combines the advantage of $\mathcal{O}(1)$ adjacency-queries in adjacency-matrices with the advantage of efficient neighborhood traversal in adjacency-lists.

Keywords: data structures, exact algorithms, recursive backtracking, vertex cover, dominating set

1 Introduction

The interest in exact algorithms for hard graph problems is now more than fifty years old. Countless exact and parameterized algorithms with improved worst-case run-times have been developed in the last several years only [5, 6, 11]. However, most research in this area concentrates on worst-case analysis, dealing with data structure design and implementation from an abstract level only.

* This research has been supported in part by the research council of the Lebanese American University.

** This research has been supported in part by the U.S. Department of Energy under the EPSCoR Laboratory Partnership Program. It has used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science under Contract No. DE-AC02-05CH11231.

In this paper, we investigate the practical aspects of some exact algorithms for $\mathcal{NP}$-hard graph-theoretic problems. The majority of such exact algorithms adopt the classical recursive backtracking methodology, with the adjacency-list or adjacency-matrix as default graph representation. We present a hybrid graph representation that trades space for time to combine the advantage of $\mathcal{O}(1)$ adjacency-queries in adjacency-matrices and the advantage of efficient neighborhood traversal in adjacency-lists.

Our interest in practical and efficient exact implementations of graph algorithms for hard computational problems has been motivated by several recent developments such as:

- More random access memory and cache memory availability making memory-intensive computations more affordable.
- The increased availability of high performance platforms.
- The fact that many $\mathcal{NP}$-complete problems are hard to approximate within reasonable relative errors A notorious example is the Maximum Clique problem [3, 9].
- The advances in designing exact and parameterized algorithms, and the fact that search-tree based algorithms are the dominant method.
- Highly demanding application domains. In some application domains, data takes months and years to collect and exact solutions are badly needed. Hence, users are often tolerant to accurate answers that take reasonable time (hours or days).
- Multi-fold approximations. Double inaccuracy arises when "approximate" solutions are provided for "simplified" models of real problems (protein folding methods are typical examples).

This paper is concerned mainly with implementation strategies that are suitable for recursive backtracking algorithms on graphs. Many aspects of our methods can also be used with hyper-graphs and with other types of problems.

2 Background

In the area of exact algorithms and parameterized complexity, the worst-case run-times for many different graph algorithms are constantly being improved. Such improvements usually involve an increase in the number of polynomial-time reductions during search. Due to the exponential number of search-tree nodes, polynomial time (housekeeping) reductions could have a butterfly effect on the efficiency of such algorithms. Obviously, this could render worst-case algorithms less practical then some simpler exact algorithms that tend to require less work at every search-tree node[3].

Moreover, search-tree based algorithms suffer from the increasing number of actions associated with branching decisions that have to be taken, then (frequently) taken back, at every search-tree node. A challenging task, therefore,

[3] We use the expression "worst-case algorithm" when referring to the algorithm (for a given problem) with the current smallest asymptotic upper bound on its run-time.

is to reduce the additional cost of *undo* operations. Generally, every operation is pushed onto a stack and later popped out and performed in reverse. We denote this action by "explicit-undo." Our hybrid graph representation addresses those challenges by reducing the cost of deletion operations via "implicit-undo." To illustrate the efficiency of our representation, several implementations using different techniques were developed and compared for two well-known graph problems: DOMINATING SET and VERTEX COVER.

2.1 Classical Graph Representation

For the sake of completeness, we ought to mention some elementary facts. Graphs are usually represented using one of two data structures: adjacency matrices (AM) or adjacency lists (AL). In addition, we often use a degrees' array to keep track of active vertices and the current cardinalities of their neighborhoods. When using AM, neighborhood traversal takes $\Omega(n)$ where n is the number of vertices in the graph. This is reduced to $\mathcal{O}(d)$, where d is the maximum vertex degree, if we use AL instead. On the other hand, checking if two vertices are adjacent requires $\mathcal{O}(d)$ in AL and $\mathcal{O}(1)$ in AM.

2.2 The Dominating Set Problem

In the Dominating Set problem, henceforth DS, we are given an n-vertex graph $G = (V, E)$, and we are asked to find a set $D \subset V$ of smallest possible cardinality such that every vertex of G is either in D or adjacent to some vertex in D. DS has received great attention, being a classical $\mathcal{NP}$-hard graph optimization problem with many logistical applications.

Until 2004, the best algorithm for DS was still the trivial $\mathcal{O}^*(2^n)$ enumeration[4]. In that same year, three algorithm were independently published breaking the $\mathcal{O}^*(2^n)$ barrier [7, 8, 10]. The best worst-case algorithm was presented by Grandoni with a running time in $\mathcal{O}^*(1.8019^n)$ [8]. Using measure-and-conquer, a bound of $\mathcal{O}^*(1.5137^n)$ was obtained on the running time of Grandoni's algorithm [4]. This was later improved to $\mathcal{O}^*(1.5063)$ in [13] and the current best worst-case algorithm can be found in [12] where a general algorithm for counting minimum dominating sets in $\mathcal{O}^*(1.5048)$ is also presented.

For our experimental work, we implemented two versions of the algorithm of [4] where DS is solved by reduction to MINIMUM SET COVER:

- AL_DS_OPT: optimization version using the adjacency-lists representation;
- HYBRID_DS_OPT: optimization version using the hybrid graph representation.

[4] Throughout this paper we use the modified big-Oh notation that suppresses all polynomially bounded factors. For functions f and g we say $f(n) \in \mathcal{O}^*(g(n))$ if $f(n) \in \mathcal{O}(g(n)poly(n))$, where $poly(n)$ is a polynomial.

2.3 The Vertex Cover Problem

In the (parameterized) Vertex Cover problem, or VC for short, we are given a graph $G = (V, E)$, together with a parameter k, and we are asked to find a set C of cardinality k such that $C \subseteq V$ and the subgraph induced by $V \setminus C$ is edgeless. The current fastest worst-case VC algorithm runs in $\mathcal{O}(kn + 1.2852^k)$ time [1]. An optimization algorithm for VC can be obtained by obvious modifications to the parameterized algorithm of [1], or using the Maximum Independent Set algorithm from [6].

For comparison purposes, four versions were implemented for VC:

- AL_VC_OPT: an optimization version using the adjacency-lists representation, based on simple modifications of the parameterized VC algorithm;
- HYBRID_VC_OPT: an optimization version using the hybrid graph representation;
- HYBRID_VC_PARM: a parameterized version using the hybrid graph representation but not taking advantage of the folding technique described in [1, 2];
- HYBRID_VCF_PARM: a parameterized version using the hybrid graph representation and modified for fast edge-contraction operations.

3 The Hybrid Graph Representation

We describe our hybrid graph representation using the example of Figure 1. The adjacency list of a vertex v is stored in an array denoted by AL$[v]$. Accordingly, AL$[v][i]$ holds the index of the i^{th} vertex in the list of neighbors of v. The adjacency matrix, denoted henceforth by IM, is used as an index table for the adjacency list, as follows: the entry IM$[u][v]$ is equal to the index of u in AL$[v]$. IM$[u][v]$ is -1 when the two vertices are not connected.

Considering the graph G of Figure 1, the initial contents of AL and IM are as follows:

$$\text{IM} \quad \begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & -1 & 0 & 0 & 0 & -1 & -1 & -1 & -1 \\ 1 & 0 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 2 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 3 & 2 & -1 & -1 & -1 & 0 & 0 & -1 & -1 \\ 4 & -1 & -1 & -1 & 1 & -1 & 1 & 0 & -1 \\ 5 & -1 & -1 & -1 & 2 & 1 & -1 & -1 & 0 \\ 6 & -1 & -1 & -1 & -1 & 2 & -1 & -1 & -1 \\ 7 & -1 & -1 & -1 & -1 & -1 & 2 & -1 & -1 \end{array} \qquad \text{DEGREE} \quad \begin{array}{c|c} 0 & 3 \\ 1 & 1 \\ 2 & 1 \\ 3 & 3 \\ 4 & 3 \\ 5 & 3 \\ 6 & 1 \\ 7 & 1 \end{array} \qquad \text{AL} \quad \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 0 & & \\ 2 & 0 & & \\ 3 & 0 & 4 & 5 \\ 4 & 3 & 5 & 6 \\ 5 & 3 & 4 & 7 \\ 6 & 4 & & \\ 7 & 5 & & \end{array}$$

Note that AL is implemented using a two dimensional array for fast (direct) access via the indexing provided by IM. We allocate enough memory to fit the neighbors of each vertex only. In addition to IM and AL, we introduce three linear

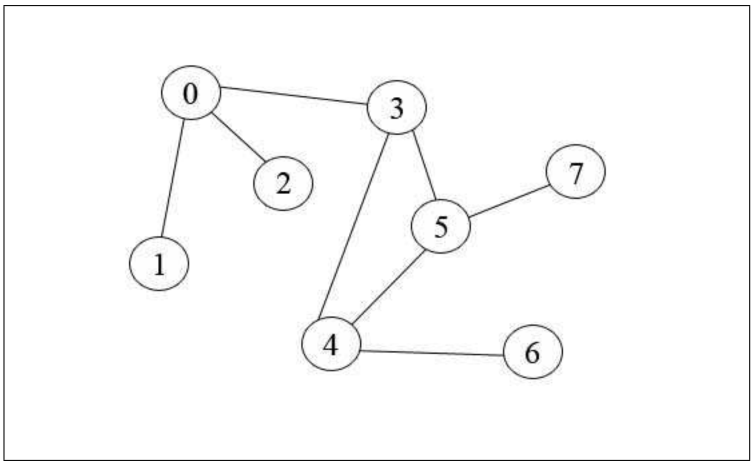

Fig. 1. Graph G.

arrays: the degree vector (DEGREE), the vertex list (LIST), and the vertex index list (IDXLIST). The degree vector is the current neighborhood cardinality, the vertex list is the list of currently active (not deleted) vertices that will be used instead of the degree vector for more efficient complete graph traversals, and the vertex index list is the index of each vertex in the vertex list. In other words, LIST$[i]$ is the i^{th} vertex in the list of active vertices and IDXLIST$[u]$ is the index of vertex u in LIST.

All data structures except for the DEGREE vector are global and their memory is allocated at startup only. The DEGREE vector is local to every search-tree node (i.e: every search-tree node receives a new copy of the vector).

$$\begin{array}{cc}
\textsc{list} & \textsc{idxlist} \\
\begin{array}{cccccccc} 0&1&2&3&4&5&6&7 \end{array} & \begin{array}{cccccccc} 0&1&2&3&4&5&6&7 \end{array} \\
\left[\begin{array}{cccccccc} 0&1&2&3&4&5&6&7 \end{array}\right] & \left[\begin{array}{cccccccc} 0&1&2&3&4&5&6&7 \end{array}\right]
\end{array}$$

In the next section, we show how these structures are dynamically modified during the search, while performing various operations. We note that some operations, like edge contraction for example, require additional bookkeeping that we briefly describe later. However, most common operations can be performed using the five data structures described above, which when combined together form the (generic) hybrid graph representation.

4 Efficient Reduction Operations

4.1 Edge Deletion

The simplest and most frequent operation performed during the search is probably edge deletion. Maintaining the hybrid data structure in this case is straightforward. For deleting an edge (u, v), the degrees of u and v are decremented by one and the adjacency lists of the two vertices are adjusted respectively by placing u at the last position of AL$[v]$ and v at the last position of AL$[u]$. Simply, each of these two operations consists of a a single swap with the last element of the respective list, together with an adjustment of the positions in IM.

```
The delete_edge function
Input: Edge $(u,v)$.
Begin
    int $i, j, x$;
    $i = $ IM$[v][u]$;
    $j = --$DEGREE$[u]$;
    $x = $ AL$[u][j]$;
    AL$[u][i] = x$;
    AL$[u][j] = v$;
    IM$[x][u] = i$;
    IM$[v][u] = j$;
    Repeat the previous steps for $i = $ IM$[u][v]$ and $j = --$DEGREE$[v]$;
End
```

Going back to our illustrative graph G, after deleting edge (0, 3), the modified AL, IM, and DEGREE will look as follows (changes in bold):

$$\text{IM} \quad \begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & -1 & 0 & 0 & \mathbf{2} & -1 & -1 & -1 & -1 \\ 1 & 0 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 2 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 3 & 2 & -1 & -1 & -1 & 0 & 0 & -1 & -1 \\ 4 & -1 & -1 & -1 & 1 & -1 & 1 & 0 & -1 \\ 5 & -1 & -1 & -1 & \mathbf{0} & 1 & -1 & -1 & 0 \\ 6 & -1 & -1 & -1 & -1 & 2 & -1 & -1 & -1 \\ 7 & -1 & -1 & -1 & -1 & -1 & 2 & -1 & -1 \end{array} \qquad \text{DEGREE} \quad \begin{array}{c|c} 0 & \mathbf{2} \\ 1 & 1 \\ 2 & 1 \\ 3 & \mathbf{2} \\ 4 & 3 \\ 5 & 3 \\ 6 & 1 \\ 7 & 1 \end{array} \qquad \text{AL} \quad \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 0 & & \\ 2 & 0 & & \\ 3 & \mathbf{5} & 4 & \mathbf{0} \\ 4 & 3 & 5 & 6 \\ 5 & 3 & 4 & 7 \\ 6 & 4 & & \\ 7 & 5 & & \end{array}$$

Notice that edge deletion runs in $\mathcal{O}(1)$ since all the information required for switching positions in AL can be found in IM. Now assume we want to undo this operation. This can be accomplished by simply setting DEGREE$[0] =$ DEGREE$[3] = 3$. The only difference between this state and the initial state is that the positions of the neighbors have changed in the adjacency lists. Knowing that every search-tree node will maintain its own copy of the DEGREE vector, no actions whatsoever need to be taken to undo this operation, thus we have an "implicit-undo" for edge deletion operations.

Remark 1. Checking if two vertices u and v are adjacent still takes constant time but requires one additional checking: u and v are adjacent when $-1 <$ IM$[u][v] <$ DEGREE$[v]$.

4.2 Vertex Deletion

Deleting a vertex v runs in $\Omega(d)$ time where d is the (current) degree of v. We run the edge deletion operation for every active neighbor of v. In addition, we remove v from the list of active vertices by swapping it with the last active vertex in LIST and decrementing the number of active vertices by one. The IDXLIST plays the same role as IM for deleting a vertex from LIST. As in the case of edge-deletion, the undo of vertex deletion requires no actions since every edge deletion can be considered as an independent operation.

To illustrate the purpose of the LIST vector, consider the two operations of copying the DEGREE vector and searching for the vertex of highest degree. Doing so would consume $\Omega(n)$ time if the DEGREE vector is used alone. This is reduced to $\Omega(n_c)$, where n_c is the number of currently active vertices, when combining the DEGREE and LIST vectors. Iterating from $i = 0$ to n_c, DEGREE[LIST[i]] returns the degree of the vertex at position i in LIST.

```
delete_vertex function
Input: Vertex $v$, and total number of active vertices $n_c$.
Begin
    int $last, i, u, d$;
    $d$ = DEGREE[$v$];
    $last$ = LIST[$n - 1$];
    $i$ = IDXLIST[$v$];
    LIST[$i$] = $last$;
    LIST[$n - 1$] = $v$;
    IDXLIST[$last$] = $i$;
    IDXLIST[$v$] = $n - 1$;
    for($i = d - 1; i \geq 0; i - -$)
        $u$ = AL[$v$][$i$];
        delete_edge($u, v$);
End
```

4.3 Edge Contraction

The next operation we consider is edge contraction. Contracting edge (u, v) replaces vertices u and v by a new vertex whose neighborhood is $N(u) \cup N(v) \setminus \{u, v\}$.

To implement this operation, we use a coloring technique that requires additional bookkeeping. Simply, vertices with the same color are treated as one single vertex obtained by contracting edges between them. Initially, every vertex v_i is assigned color c_i, and every color class c_i has initial cardinality one and degree $d(c_i) = d(v_i)$. In addition to previously discussed data structures, we use the following:

– the VCOLOR vector: holds the current color of every vertex;

- the COLOR_CARD (CC) vector indicates the current cardinality of every color set;
- the COLOR_DEGREE (CD) vector holds the current degree of every color set;
- the COLOR_SET_LIST (CSL) holds the list of vertices belonging to every color set.

The LIST and IDXLIST do not hold vertex information anymore, but they maintain the list of active (not deleted) colors instead, since all operations now involve color sets. When no edge contraction operations are performed, color sets would be identified with their corresponding vertices.

$$
\begin{array}{cc} & \text{CD} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}3\\1\\1\\3\\3\\3\\1\\1\end{bmatrix}\end{array}
\quad
\begin{array}{cc} & \text{CC} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}1\\1\\1\\1\\1\\1\\1\\1\end{bmatrix}\end{array}
\quad
\begin{array}{cc} & \text{CSL} \\ & \begin{array}{ccc}0&1&2\end{array} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}0&&\\1&&\\2&&\\3&&\\4&&\\5&&\\6&&\\7&&\end{bmatrix}\end{array}
\qquad
\begin{array}{cc} & \text{DEGREE} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}3\\1\\1\\3\\3\\3\\1\\1\end{bmatrix}\end{array}
\quad
\begin{array}{cc} & \text{AL} \\ & \begin{array}{ccc}0&1&2\end{array} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}1&2&3\\0&&\\0&&\\0&4&5\\3&5&6\\3&4&7\\4&&\\5&&\end{bmatrix}\end{array}
$$

$$
\begin{array}{c} \text{VCOLOR} \\ \begin{array}{cccccccc}0&1&2&3&4&5&6&7\end{array} \\ \begin{bmatrix}0&1&2&3&4&5&6&7\end{bmatrix}\end{array}
\quad
\begin{array}{c} \text{LIST} \\ \begin{array}{cccccccc}0&1&2&3&4&5&6&7\end{array} \\ \begin{bmatrix}0&1&2&3&4&5&6&7\end{bmatrix}\end{array}
\quad
\begin{array}{c} \text{IDXLIST} \\ \begin{array}{cccccccc}0&1&2&3&4&5&6&7\end{array} \\ \begin{bmatrix}0&1&2&3&4&5&6&7\end{bmatrix}\end{array}
$$

When edge contraction is possible, the initial state for the graph G consists of all structures previously shown. AL, IM, CSL, LIST and IDXLIST would be globally stored (in RAM), while DEGREE, CD, CC and VCOLOR would be copied at every search-tree node. To contract an edge (v_0, v_3), we actually assign both vertices the same color. Assuming we assign the two vertices color c_0, the modifications required are shown below in bold (changes to IM not shown here but are required):

$$
\begin{array}{cc} & \text{CD} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}\mathbf{4}\\1\\1\\\mathbf{0}\\3\\3\\1\\1\end{bmatrix}\end{array}
\quad
\begin{array}{cc} & \text{CC} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}\mathbf{2}\\1\\1\\\mathbf{0}\\1\\1\\1\\1\end{bmatrix}\end{array}
\quad
\begin{array}{cc} & \text{CSL} \\ & \begin{array}{ccc}0&1&2\end{array} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}0&\mathbf{3}&\\1&&\\2&&\\3&&\\4&&\\5&&\\6&&\\7&&\end{bmatrix}\end{array}
\qquad
\begin{array}{cc} & \text{DEGREE} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}\mathbf{2}\\1\\1\\\mathbf{2}\\3\\3\\1\\1\end{bmatrix}\end{array}
\quad
\begin{array}{cc} & \text{AL} \\ & \begin{array}{ccc}0&1&2\end{array} \\ \begin{array}{c}0\\1\\2\\3\\4\\5\\6\\7\end{array} & \begin{bmatrix}1&2&3\\0&&\\0&&\\\mathbf{5}&4&\mathbf{0}\\3&5&6\\3&4&7\\4&&\\5&&\end{bmatrix}\end{array}
$$

$$\begin{array}{ccc} \text{VCOLOR} & \text{LIST} & \text{IDXLIST} \\ \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 1 & 2 & \mathbf{0} & 4 & 5 & 6 & 7 \end{array} & \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 1 & 2 & \mathbf{7} & 4 & 5 & 6 & \mathbf{3} \end{array} & \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 1 & 2 & \mathbf{7} & 4 & 5 & 6 & \mathbf{3} \end{array} \end{array}$$

Since we are dealing with simple graphs, any edge between two vertices belonging to the same color set is deleted and no more than one edge is allowed between a color set and another. Clearly, such an operation can be implicitly taken back, but is also more time and space consuming than simple vertex deletion.

This technique makes it possible to implement the vertex folding operation, introduced in [1], for the parameterized VC algorithm.

4.4 The Vertex Cover Folding Operation

Let (G, k) be an instance of the Vertex Cover problem and let $u \in V(G)$ be a degree-two vertex with neighbors v and w. If v and w are adjacent, then there is a minimum vertex cover that contains v and w (and not u). So it is safe to delete u, add v and w to the potential solution and decrement k by two. In the case where v and w are non-adjacent, an equivalent Vertex Cover instance is obtained by contracting edges uv and uw and decrementing k by one. This latter operation is known as degree-two vertex folding.

As we shall see, applying the coloring technique to implement vertex folding considerably improves the runtime on certain recalcitrant instances, but slows down the computation on graphs where folding rarely occurs. In such cases, the overhead of maintaining color-sets is a drawback. Note that folding alone made it possible to obtain a worst-case run time of $O^*(1.285^k)$ in [1]. Yet, our results show that excluding folding from the same algorithm is faster on a large number of instances, especially real ones.

5 Experimental Results

Four different versions were implemented for the VERTEX COVER algorithm. AL_VC_OPT and HYBRID_VC_OPT are two generic search-tree optimization versions using the adjacency-list and hybrid graph representations respectively. In Table 1, the running times for both versions are reported for a number of DIMACS graphs. HYBRID_VC_PARM is a parameterized hybrid version that does not take advantage of vertex folding, while HYBRID_VCF_PARM is a parameterized version implemented using the coloring technique described in the previous section for folding.

In general, the folding technique is at most two times slower than the simple generic branching algorithm. It gets faster as the difference between the highest and lowest vertex-degrees gets smaller. In particular, applying vertex folding, via our coloring technique, is much faster on regular graphs. To illustrate, tests were run on a 4-regular graph, by varying the input parameter, and results are reported in Table 2.

As for the Dominating Set problem, AL_DS_OPT denotes the optimization version using the adjacency-lists representation and HYBRID_DS_OPT the optimization version using the hybrid graph representation. Running times of the DS implementations on random graphs, with various densities, are given in Table 3. In addition, real DS instances for biological problems were obtained from the Gene Expression Omnibus (GEO) data-sets available at `http://www.ncbi.nlm.nih.gov` and the results are shown in Table 4. The raw data (SOFT) files were transformed into simple unweighted graphs using Pearson's coefficients and appropriate thresholding. The threshold value used for each graph appears in the file extension in Table 4.

All codes were implemented in standard C, and experiments were run on two types of machines (in two labs): Intel Core2 Duo 2327 MHz and Intel Xeon Processor X5550 (Nahalem) 2.66 GHz Quad Core. However, the numbers reported in each row were obtained on the same architecture.

Table 1. AL_VC_OPT vs. HYBRID_VC_OPT (no folding).

Graph	$\|V\|$	$\|E\|$	$\|C\|$	AL_VC_OPT	HYBRID_VC_OPT
brock800_1.clq	800	207505	790	1 min 54 sec	32 sec
p_hat300-1.clq	300	10933	261	1 min 25 sec	41 sec
p_hat500-1.clq	500	31569	450	3 hr 48 min	1 hr 23 min
p_hat500-2.clq	500	62946	464	40 sec	12 sec
p_hat700-1.clq	700	60999	635	$>$ 1 week	93 hr 20 min
p_hat700-2.clq	700	121728	651	15 min 10 sec	3 min 44 sec
p_hat700-3.clq	700	183010	690	20 sec	6 sec
p_hat1000-2.clq	1000	244799	946	31 hr 26 min	5 hr 28 min
p_hat1000-3.clq	1000	371746	989	2 min 47 sec	48 sec
p_hat1500-3.clq	1500	847244	1488	20 min 57 sec	5 min 3 sec

Table 2. HYBRID_VC_PARM vs. HYBRID_VCF_PARM (with folding) on a 4-regular graph having 300 vertices and 600 edges.

Vertex Cover Size ($\|K\|$)	Answer	No Folding	With Folding
192	yes	14 sec	$<$ 1 sec
191	yes	17 sec	6 sec
190	yes	2 hr 14 min	6 min 27 sec
165	no	$>$ 4 days	46 min 56 sec
160	no	38 hr 2 min	2 min 32 sec

Table 3. AL_DS_OPT vs. HYBRID_DS_OPT.

Graph	$\|V\|$	$\|E\|$	$\|D\|$	AL_DS_OPT	HYBRID_DS_OPT
rgraph1	100	400	16	21 min 4 sec	4 min 11 sec
rgraph2	100	600	11	5 min 5 sec	53 sec
rgraph3	100	1500	6	41 sec	5 sec
rgraph4	150	1200	14	16 hr 46 min	2 hr 27 min
rgraph5	150	1500	11	3 hr 31 min	28 min 20 sec
rgraph6	150	3000	6	2 min 8 sec	12 sec
rgraph7	150	3000	7	27 min 16 sec	2 min 1 sec
rgraph8	200	4500	9	5 hr 44 min	30 min 8 sec
rgraph9	200	5000	8	1 hr 20 min	6 min 46 sec
rgraph10	200	6000	6	1 hr 36 min	7 min 13 sec
rgraph11	200	12000	4	6 min 49 sec	17 sec
rgraph12	250	9000	8	14 hr 37 min	56 min 53 sec
rgraph13	250	10000	7	1 hr 30 min	5 min 34 sec
rgraph14	250	12000	5	4 hr 41 min	16 min 19 sec
rgraph15	250	24000	3	19 sec	$<$ 1 sec
rgraph16	300	22461	4	8 min 31 sec	17 sec
rgraph17	300	22258	4	28 min 32 sec	1 min 10 sec
rgraph18	300	11063	8	133 hr 38 min	5 hr 54 min
rgraph19	300	11287	8	$>$ 7 days	8 hr 14 min
rgraph20	1000	374633	3	4 min 37 sec	2 min 29 sec
rgraph21	1000	374552	3	28 min 37 sec	6 min 36 sec

Table 4. AL_DS_OPT vs. HYBRID_DS_OPT.

Graph	$\|V\|$	$\|E\|$	$\|D\|$	AL_DS_OPT	HYBRID_DS_OPT
GDS3211.96	4636	11249	1567	3 min 57 sec	1 min 3 sec
GDS3221.95	5759	43991	1551	11 min 55 sec	3 min 6 sec
GDS3221.94	8517	131498	2042	1 hr 5 min	11 min 8 sec
GDS3221.93	11065	315488	2427	3 hr 58 min	26 min 27 sec
GDS3221.92	13712	649073	2758	$>$ 6 hours	54 min 43 sec

6 Conclusion

We presented a hybrid graph representation that efficiently trades space for time and facilitates many common graph operations required during recursive back-tracking. Experiments on both Vertex Cover and Dominating Set showed the utility of using this dynamic data structure. The running times of the same algorithm were shown to be consistently reduced, sometimes from days to hours.

The main focus in this paper was on operations reducing the original graph size, such as vertex deletion and edge contraction. However, some algorithms require operations that do not decrease the size of an input graph. Such operations are harder to implement. Edge addition is a notable example that remains to be considered.

References

1. Chen, J., Kanj, I.A., Jia, W.: Vertex cover: Further observations and further improvements. Journal of Algorithms, 41:313–324 (2001)
2. Chen, J., Liu, L., Jia, W.: Improvement on vertex cover for low-degree graphs. Networks, 35(4):253–259 (2000)
3. Engebretsen, L., Holmerin, J.: Clique is hard to approximate within $n^{1-o(1)}$. In: Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP), pp. 2–12, London, UK. Springer-Verlag (2000)
4. Fomin, F.V., Grandoni, F., Kratsch, D.: Measure and conquer: domination - a case study. In: Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP), Springer LNCS, pp. 191–203. Springer (2005)
5. Fomin, F.V., Grandoni, F., Kratsch, D.: Solving connected dominating set faster than 2^n. Algorithmica, 52(2):153–166 (2008)
6. Fomin, F.V., Grandoni, F., Kratsch, D.: Measure and conquer: a simple $O(2^{0.288n})$ independent set algorithm. In: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm (SODA), pp. 18–25, New York, USA (2006)
7. Fomin, F.V., Kratsch, D., Woeginger, L., Woeginger, G.J.: Exact (exponential) algorithms for the dominating set problem. In: Proceedings of the 30th Workshop on Graph Theoretic Concepts in Computer Science (WG), pp. 245–256. Springer (2004)
8. Grandoni, F.: A note on the complexity of minimum dominating set. J. Discrete Algorithms, 4(2):209–214 (2006)
9. Hastad, J.: Clique is hard to approximate within $n^{(1-\epsilon)}$. Acta Mathematica, pp. 627–636 (1996)
10. Randerath, B., Schiermeyer I.: Exact algorithms for minimum dominating set. Technical report, Zentrum für Angewandte Informatik Köln, Lehrstuhl Speckenmeyer (2004)
11. van Rooij, J.M., Bodlaender H.L.: Exact algorithms for edge domination. Technical Report UU-CS-2007-051, Department of Information and Computing Sciences, Utrecht University (2007)
12. van Rooij, J.M., Nederlof, J., van Dijk T.C.: Inclusion/exclusion meets measure and conquer: Exact algorithms for counting dominating sets. Technical Report UU-CS-2008-043, Department of Information and Computing Sciences, Utrecht University (2008)
13. van Rooij, J.M., Bodlaender H.L.: Design by measure and conquer, a faster exact algorithm for dominating set. In: Susanne Albers and Pascal Weil, editors, STACS, volume 1 of LIPIcs, pp. 657–668. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2008)