Practical performance of incremental topological sorting and cycle detection algorithms

Ragnar Lárus Sigurðsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Master's thesis 2016

Practical performance of incremental topological sorting and cycle detection algorithms

Ragnar Lárus Sigurðsson

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

Practical performance of incremental topological
sorting and cycle detection algorithms
Ragnar Lárus Sigurðsson

Supervisor: Atze van der Ploeg, Department of Computer Science and Engineering
Examiner: Patrik Jansson, Department of Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover:

Typeset in LaTeX
Gothenburg, Sweden 2016

Practical performance of incremental topological sorting
and cycle detection algorithms
Ragnar Lárus Sigurðsson
Department of Computer Science
Chalmers University of Technology

Abstract

Algorithms become more advanced and asymptotic time bounds get lower but there is very little data on the actual performance of new algorithms. The aim of this thesis is to do empirical testing of the most recent incremental topological sorting and cycle detection algorithms in order to compare them and to provide an accessible guide to where each algorithm performs best.
The algorithms are implemented as the articles describe them and compared on even grounds by measuring their performance by adding edges to graphs.
For sparse graphs the HKMST-Sparse [7] algorithm performed best and HKMST-Dense [7] for very dense graphs. The Pearce & Kelly [8] algorithm is a strong contender as it is extremely simple and has acceptable performance across all graph densities and performs best in the range 35-80% density.

Keywords: Topological sorting, Cycle detection, Algorithms, Data structures.

Contents

1 Introduction **1**
1.1 Background 2

2 Setup **5**
2.1 Graph Generation 5
2.2 Testing for Correctness 5

3 Algorithms **7**
3.1 Pearce & Kelly 10
3.2 HKMST Sparse Algorithm $O(m^{3/2} \log n)$ 11
3.3 HKMST Sparse Algorithm $O(m^{3/2})$ 12
3.4 BFGT Sparse $O(min(m^{1/2}, n^{2/3})m)$ 12
3.5 HKMST Dense Algorithm $O(n^2 \log n)$ 13
3.6 BFGT Dense $O(n^2 \log n)$ 15

4 Supporting Algorithms and Data Structures **17**
4.1 Order Maintenance Problem 17
4.1.1 Implementation 17
4.1.2 Scapegoat Tree 18
4.1.3 Choosing α 18
4.1.4 Binary Labeling 19
4.2 Finding The Median 21
4.2.1 Median of Medians 21
4.2.2 Median of Medians With Random 22
4.2.3 QuickSelect 22

5 Results **23**
5.1 HKMST selection method 24
5.2 Super sparse graphs 25
5.3 Sparse graphs 27
5.4 All-round 28
5.5 Dense graphs 29

6 Conclusion **31**
6.1 Source code 31

7 Discussion **33**

Bibliography **35**

1

Introduction

Topological sorting and cycle detection in graphs are well known problems and the algorithms to solve them give solutions to an abundance of theoretical and real world challenges. This thesis focuses on testing algorithms that can perform this incrementally, that is they work on a per edge addition basis and not on a complete graph. All the algorithms for cycle detection have the following generic interface:

- void AddVertex(v); Adds a vertex to the graph
- bool AddEdge(v, w); Adds an edge between two vertices

This interface allows vertex and edge additions to a graph. When a new edge is added to the graph the underlying algorithm will check if the edge addition creates a cycle. If a cycle is detected *AddEdge* will report a failure, if no cycle is detected the edge is added to the graph. We extend this interface to allow for topological ordering and comparison of two vertices:

- void AddVertex(v); Adds a vertex to the graph
- bool AddEdge(v, w); Adds an edge between two vertices
- bool Compare(v, w); Compares the topological order of two vertices
- List Topology(); Returns a topological ordering/linear extension of the graph

A topological order of a directed acyclic graph is a linear order of its vertices such that for every directed edge (v, w) from vertex v to vertex w, v comes before w in the total order. More specifically a DAG uniquely determines a partially ordered set; we view the vertices in the graph as our set of objects and define the inequality relation as follows: $v \leq w$ if there exists a path from v to w or in other words w is reachable from v. This relation is reflexive, antisymmetric and transitive making it a partial ordering. A linear extension of the partial order is a total order that is compatible with the partial order, more specifically, given any two partial orders $\leq$, $\leq^*$ on a set X, $\leq^*$ is a linear extension of $\leq$ when $\leq^*$ is a total order and $\forall(v, w) \in X$, if $v \leq w$ then $v \leq^* w$. A topological order is then the same as a linear extension, and an example of this can be seen in Figure 1.1.

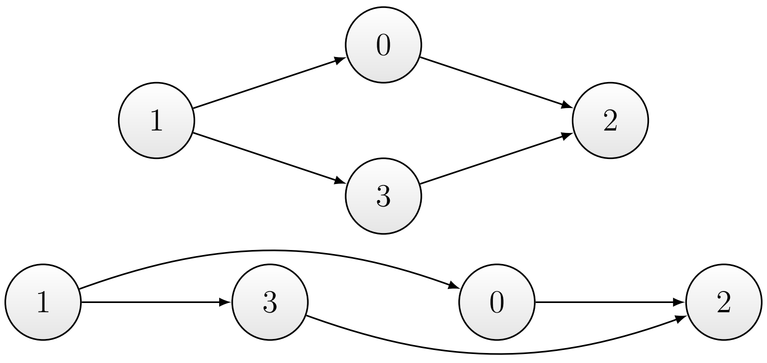


Figure 1.1: A topological order can be seen as aligning the vertices on a straight line where all edges are from left to right. The possible topological orders for this graph are [1,3,0,2] (as shown above) and [1,0,3,2].

Incremental cycle detection is a problem that is still being researched, our intention is to compare the most modern of these algorithms in order to better understand for which sizes and densities of graphs they are most suited.

1.1 Background

Incremental cycle detection is an advanced problem in Computer Science and many well known researchers have developed algorithms to solve this problem, such as Pearce and Kelly [8], and Robert Tarjan [3].
Topological sorting and cycle detection have many practical applications where ordering is important and cycles are not allowed. A few well known uses are: pointer analysis [9], circuit evaluation [1], deadlock detection systems [2] and functional reactive programming [10]. Incremental cycle detection is of great use in all of these examples. For some of these cases it can be difficult to envision the link to topological sorting and cycle detection. If we take deadlock detection as an example, we imagine that resources are represented in a resource allocation graph, if the pointers pointing between the resources create a cycle, there is a possible deadlock. The same logic transfers into locking mechanisms in database control systems where transactions use table locks to maintain data integrity. Multiple transactions can thus cause deadlocks when locking tables. As an example, transaction $T1$ wants to read from table A and write to table B, transaction $T2$ wants to read from table B and write to table A. If $T1$ has a read lock on table A and $T2$ has a read lock on table B, they are stuck in a deadlock. In database concurrency control mechanism a wait-for graph is used to deal with these situations.

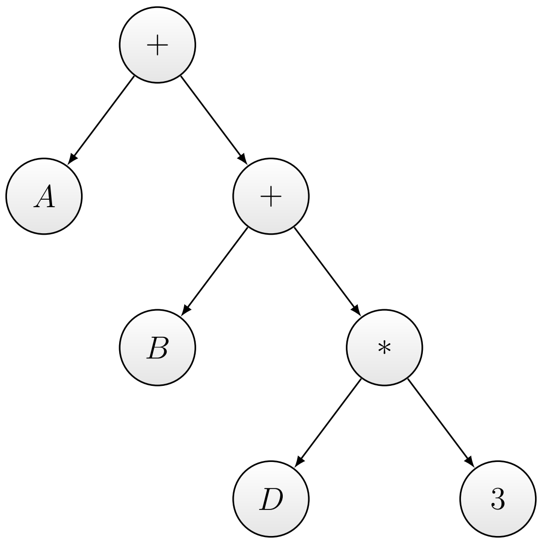


Figure 1.2: Expression: $A + (B + D * 3)$. Topological order: $[+, A, +, B, *, D, 3]$

Incremental topological sorting and cycle detection is also crucial in functional reactive programming. If we examine the expression in Figure 1.2, we see that the reverse topological order of the expression tree gives us the evaluation order. In FRP these expression trees can change during runtime, as the changes are most likely affecting a small part of the graph we can limit the number of nodes that have to be re-evaluated. This means there is a lot to gain from performing this task incrementally.

2

Setup

In this chapter we go over the modules needed to test the performance of the algorithms. To be able to compare the running time of the algorithms we need a variety of randomly generated graphs. In addition to that, we also need to make sure the algorithms are all correctly implemented, in the sense that they give valid outputs.

2.1 Graph Generation

The graph generation module creates random directed acyclic graphs by randomly adding edges to a graph. If a newly added edge creates a cycle it is removed. When generating graphs we take as input the number of nodes n and a value p. The variable p is the percentage of edges, where $p = 1$ is a fully connected graph. In our case we do not allow self looping edges or cycles, this means that with $n = 1000$ and $p = 0.5$ we can expect the generated graph to have approximately $\frac{n*(n-1)}{2} * p = \frac{1000*(1000-1)}{2} * 0.5 = 249.750$ edges.

2.2 Testing for Correctness

To ensure the algorithms give valid outputs, we check them for correctness. That is, we check if they detect cycles when cycles are added and we make sure the topological order they output is a valid order.
As topological orders are not necessarily unique for a given graph, we cannot simply compare an algorithms output to a correct order. As an example, a graph of 3 vertices where only 2 are connected, can have 3 valid topological orders as shown in Figure 2.1.

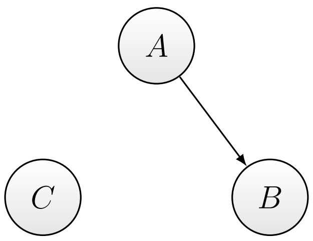


Figure 2.1: In a topological order of this graph, C can appear in any position of the order, since it is unconnected.

In order to test the correctness of the topological order of an algorithm, we generate a reachability matrix using the Floyd-Warshall algorithm. The reachability matrix tells us if there is a path from v to w. If there is a path between two nodes, then we know that they should appear in the same order as in the algorithm's topological order.

3
Algorithms

In this chapter, we describe the different techniques used in the algorithms for incremental topological sorting and cycle detection. In the next chapter, we quantify the difference in performance between these techniques and reason about when one should be preferred over the other.
To introduce our terminology and to illustrate the inner workings of such algorithms, let us consider these two naive example algorithms: one which only detects cycles incrementally, and one slightly less naive algorithm that does incremental topological sort in addition to cycle detection. Our incremental cycle detection algorithms must support the following operations:

- void AddVertex(v): Creates a new vertex in the graph
- bool AddEdge(v, w): Adds a new edge between v and w in the DAG. If the new edge would create a cycle, the algorithms reports it and the edge is not added to the graph

Our first naive example algorithm which only detects cycles, works as follows: each vertex stores a list of references to vertices it has edges from, along with boolean variables $Visited$ and *Cycle*, both initially false. When a new edge (v, w) is added, the algorithm tries to detect a cycle by finding a path from w to v. This is done by recursively traversing edges backwards from v, and labeling the vertices visited using the $Visited$ variable. If a vertex has already been visited, its edges are not traversed again. We continue the search until either w is found using the *Cycle* variable, in which case there is a cycle, or until all edges reachable from v have been traversed, in which case the graph remains acyclic.

```
AddEdge(v, w)
  nodesVisited = []
  w.Cycle = true;
  hasCycle = Visit(v, w, ref nodesVisited);

  w.Cycle = false;
  foreach(node in nodesVisited)
    node.Visited = false;

  return hasCycle;

Visit(parent, child, ref nodesVisited)
  parent.Visited = true;
  nodesVisited.Add(parent);
  if(parent.Cycle)
    return true; // Stop and report cycle
  foreach(inEdge in parent.incoming)
    if(!inEdge.Visited)
      if(Visit(inEdge, parent, ref nodesVisited)) // parent becomes child
        return true; // cycle
  return false;
```

In this naive depth first algorithm, choosing which edge to traverse next and traversing that edge is called a ***search step***. Due to the simplicity of this algorithm, in

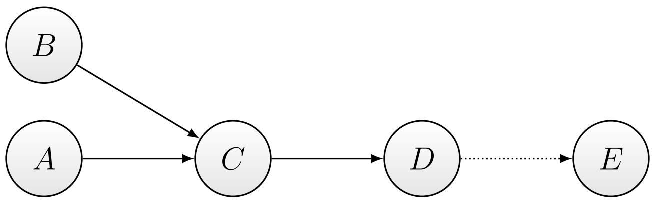


Figure 3.1: When the edge (D, E) is added, both A and B have to be visited to maintain a higher level than C.

the worst case the entire graph has to be traversed, depth first, in order to conclude that no cycle exists. More advanced algorithms limit the search space by using additional information, such as the current topological ordering. Choosing the next edge to traverse then becomes more complicated and some implementations utilize techniques such as priority queues and median finding.

Our second example algorithm also maintains a topological ordering of the graph in addition to cycle detection. We use a simple numbering scheme to achieve this by extending the previous algorithm, adding a *Level* variable, initially 0, to each vertex. This will ultimately give us a topological ordering of the graph by ordering the vertices by *level* in ascending order.

```
AddEdge(v, w)
  nodesVisited = []
  w.Cycle = true;
  hasCycle = Visit(v,w, ref nodesVisited);

  w.Cycle = false;
  foreach(node in nodesVisited)
    node.Visited = false;

  return hasCycle;

Visit(parent, child, ref nodesVisited)
  parent.Visited = true;
  nodesVisited.Add(parent);
  if(parent.Cycle)
    return true; // Stop and report cycle
  if(parent.Level <= child.Level)
  {
    parent.Level = child.Level + 1;
    foreach(inEdge in parent.incoming)
      if(!inEdge.Visited)
        if(Visit(inEdge, parent, ref nodesVisited)) // parent becomes child
          return true; // cycle
  }
  return false;
```

Extending the algorithm further, recall that the search is traversing backwards. Each time a vertex is visited, we compare its *level* to the *level* of the vertex traversed before it (we call that a child vertex). As the code sample shows, a vertices successor always has a lower level than its predecessor. This method of maintaining the topological order causes two problems: we can theoretically run out of numbers as the *level* can only increase. This could be solved by a more advanced numbering scheme. Secondly and more importantly, it is hard to limit our search space since we depend on traversing all vertices higher in the topology, which is in fact the worst case. An example of this behavior is shown in Figure 3.1

Our naive example algorithm has a decent performance on graphs that are very sparse. The denser the graphs (longer paths) become, the more vertices have to be traversed. This results in poor performance. The advanced algorithms we want to measure are generally designed for either sparse or dense graphs, since different methods of traversal are better suited for different graph densities. An example of a different traversal method is traversing the topological order of the graph instead of the edges. For this reason, each article describes a dense and a sparse algorithm. In the Table 3.1 we list the asymptotic time complexities of each one, where our naive example algorithm is called Simple Incremental.

	Time per step	Time per edge	Time per m edges
Static Tarjan			$O(nm)$
Static Kahn			$O(nm)$
Simple Incremental	$O(1)$	$O(m)$	$O(m^2)$
HKMST Two-Way	$O(\log n)$	$O(\sqrt{m}\log n)$	$O(m^{3/2}\log n)$
HKMST Soft-Threshold	$O(1)$	$O(\sqrt{m})$	$O(m^{3/2})$
HKMST Dense	-	-	$O(n^2\log n)$
BFGT Sparse	-	-	$O(min(m^{1/2}, n^{2/3})m)$
BFGT Dense	-	-	$O(n^2\log n)$
Pearce & Kelly	-	-	-

Table 3.1: Time complexity of the algorithms where n and m stand for vertices and edges respectively. Cells are left empty if the articles did not provide complexity numbers. In the case of Pearce & Kelly, they use a different system to measure the efficiency of their algorithm which is not comparable.

The time complexities listed in the table are split into three columns: time per step, time per edge and time per m edges. Time per step is the time complexity of traversing one step, which can be an edge or a vertex. This is where the advanced algorithms limit their search by intelligently choosing which edge or vertex to traverse and which edges or vertices can be skipped for the current step. Time per edge is how long it takes to add one edge to the graph. For our naive example algorithm this is $O(m)$ since it can potentially have to traverse every edge in the graph. Finally time per m edges, is how long it takes to add m edges or an entire graph. This complexity can thus be compared to the time complexities of static graph algorithms such as Tarjan or Kahn.
All the algorithms share one common method of pruning the search. If the vertices of the edge being added are already in order, no search has to be done and the edge can be added. An example of this can be seen in Figure 3.2.

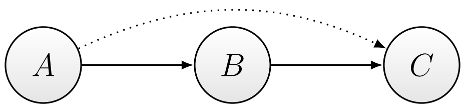


Figure 3.2: No search has to be performed to add the edge (A, C) as the vertices are already in order.

In the following sections we cover each algorithm in more detail, specifically how it limits the search space and if it uses any supporting algorithms.

3.1 Pearce & Kelly

In 2006 Pearce & Kelly [8] proposed an algorithm that had worse time bounds than its counterparts at the time but performed better in practice. Their claim was backed up by experiments they performed against other algorithms. They claim that the simplicity of the algorithm along with its use of basic data structures yields an over all better practical performance. It is interesting to see if simplicity still has the upper hand.
The algorithm uses a two-way depth first traversal and stores a total topological ordering in an array of size $|V|$. This means that each vertex is assigned to a unique number in the total order from 1 to $|V|$. We examine Figure 3.3 which shows the edge (J, A) being added. We refer to the vertices between A and J in the topological order as the affected region. The algorithm searches forward from A and backwards from J traversing only edges leading to vertices within the affected region. There is no need to traverse out of the affected region as the order of those vertices remains the same and they can not lead to a cycle.

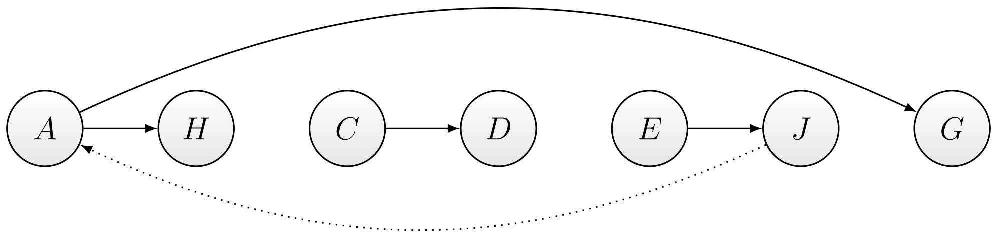


Figure 3.3: PK prune the search by only traversing edges leading to vertices between A and J, G will not be traversed.

During the search the algorithm keep tracks of the vertices traversed by the forward and backward searches separately in the sets F and B respectively. To restore topological order, the vertices in B have to be placed in front of the vertices in F while maintaining the original internal order of the sets. This is done by repositioning the vertices in F and B. Only vertices that were traversed will have their ordering altered. As seen in Figure 3.4, vertices C and D remain in the same position in the total order.

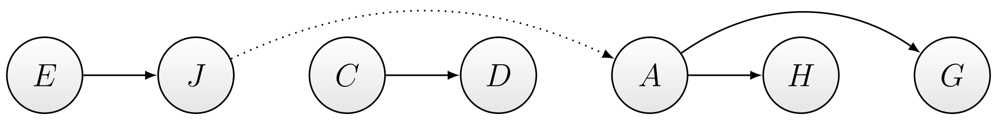


Figure 3.4: Topological order has been restored, vertices C and D were not affected.

3.2 HKMST Sparse Algorithm $O(m^{3/2} \log n)$

HKMST Sparse [7] improves the previous best bound on sparse graphs ($m/n = O(1)$) by a logarithmic factor [1]. This increase in performance is not achieved by limiting the search space further then its preceding algorithms, but by changing the underlying data structures for more efficient ones. The algorithm requires a special data structure for its reordering procedure. This advanced data structure solves a problem referred to as the order maintenance problem, is covered in detail in section 4.1. The data structure allows us to maintain the topological order of our vertices by giving us access to the following methods that run in constant amortized time.

- bool Query(x, y): Given references to x and y, returns true if x appears before y in the total order.
- Item Insert(x, y): Given a reference to x, inserts y right after it in the total order and returns a reference to the inserted item.
- void Delete(x): Given a reference to x, removes it from the order.

We start by explaining an earlier algorithm the article uses to improve on. This algorithm uses the same underlying data structures, but utilizes priority queues to select edges to traverse. We then continue to show how taking advantage of certain properties of the data structure achieves a lower time complexity.
The algorithm uses a two-way vertex guided search. When the edge (v, w) is added, the area between v and w in the topological order is the affected region. We can say that the backward search starts on the right side of the affected region and the forward search starts on the left side. The algorithm traverses edges from compatible vertices. We say that a pair of vertices is compatible when a forward vertex is left of a backward vertex. Candidate vertices for a search step are the incoming/outgoing edges from previously traversed vertices for the backward and forward search respectively. When no compatible pair can be found the searches have crossed and no more edges need to be traversed. Cycles are detected by the forward search traversing a vertex visited by the backward search, or vice versa. We examine Figure 3.5 and note that w and v are compatible vertices, the forward search will traverse into A and the backward search into E. In the next search step we traverse A, C and E, H. Finally we find that C and H are not compatible and stop the search.

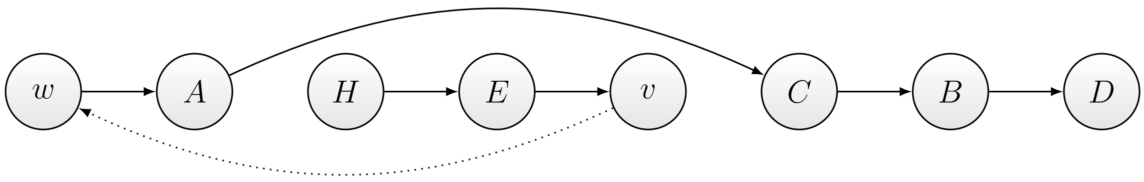


Figure 3.5: The forward search traverses w, A and the backward search v, E, in the next step A and E are compatible vertices and are traversed. The search ends since H and C are not compatible.

To restore the topological order we find a vertex t which is the leftmost vertex in

$\{v \cup F_A\}$, where F_A are forward vertices with untraversed edges. If we use Figure 3.5 as an example then $t = min(v, C) = v$. As the search traverses edges out of compatible vertices, F_A can contain vertices outside of the affected region as is the case with C. We then proceed to find $F_<$ of forward vertices left of t and $B_>$ of backward vertices right of t. Since $t = v$, $B_>$ is empty and we can proceed to insert all vertices in $F_<$ just after t. Alternatively if $t \neq v$ we insert all vertices in $F_<$ just before t and all vertices in $B_>$ before the vertices in $F_<$. We can see the restored topological order in Figure 3.6.

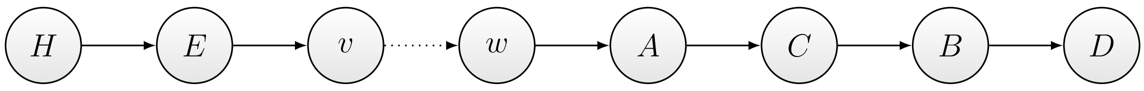


Figure 3.6: w, A, C were inserted just after v in the topological order.

3.3 HKMST Sparse Algorithm $O(m^{3/2})$

The algorithm described in the previous section uses priority queues or min/max queues to find compatible vertices. This improved algorithm is the same as the previous one except it uses a different method to find compatible vertices. They call this method soft-threshold search and with it they eliminate the need for priority queues reducing the time complexity by $\log n$.
The search maintains 3 sets for each search direction, one with vertices that have already been traversed and vertices that have been considered and eliminated, we call these vertices dead. Additionally we keep track of passive vertices that are candidates for future traversal steps and finally active vertices which are considered for the current search step. Soft-threshold search makes additional vertices dead in comparison to the vertex guided search reducing the number of vertices it has to traverse. A vertex s is selected to be the soft-threshold, initially v. When either of the active sets becomes empty, we pick a new s from the relevant passive set. This is done either uniformly at random or with a median selection method which is covered in Section 4.2. We then proceed to move vertices in the relevant passive set to the active set if they are $\leq$ or $\geq$ compared to s for forward and backward sets respectively.

3.4 BFGT Sparse $O(min(m^{1/2}, n^{2/3})m)$

BFGT is our third sparse algorithm. The benefits of this algorithm over HKMST Sparse are mostly in its use of simple data structure and that it does not use selection methods to pick edges to traverse. BFGT is also a two-way search algorithm, however the searches have different purposes. The algorithm uses two ways to bound its search. Each vertex has a level, $k(v)$, initially 0. The backward search can only search within one level and is stopped if it traverses $\Delta = min(m^{1/2}, n^{2/3})$ edges, an example of this is seen in Figure 3.7. The forward search only traverses edges whose level increases, more specifically if we have an edge (x, y) it will only be traversed if $k(x) > k(y)$, subsequently we increase the level of y to the level of x. This behavior

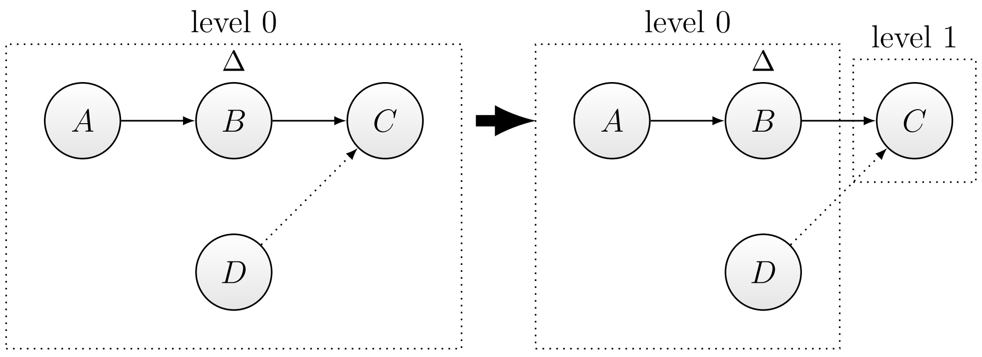


Figure 3.7: Backward search traverses (D, C) and stops since $\Delta = 1$. This causes C to be raised to a level above D. The forward search then has no edges to traverse, since $k(C) > k(D)$.

is shown in Figure 3.8. This makes the upper bound on all forward searches the number of vertex level increases.

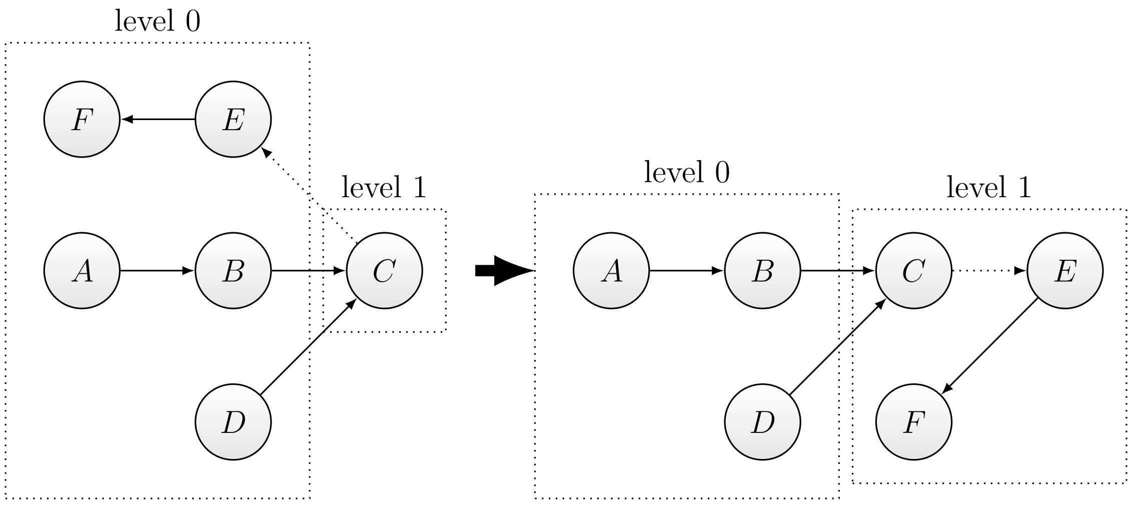


Figure 3.8: There are no backwards edges out of C within level 1, the backward search does nothing. Since $k(E) < k(C)$, E is moved to the level of C. The forward search then traverses (E, F) raising the level of F to its own.

For cycle detection the algorithm keeps track of vertices traversed by each search, keeping their order. The backward search detects cycles by traversing into v, while the forward search looks for vertices already traversed in the backward search. If no cycle is detected, the algorithm then assigns new unique identifiers to the vertices that were traversed.

3.5 HKMST Dense Algorithm $O(n^2 \log n)$

HKMST Dense is the first algorithm we look at that is specifically designed for dense graphs. According to the article, the soft-threshold search used in their sparse algorithms becomes less efficient as the graph grows denser. For this reason they

designed an algorithm which traverses the topological order of the graph instead of its edges. They claim that this approach is faster for sufficiently dense graphs.
The algorithm maintains the graph as an adjacency matrix indicating whether there is an edge between two vertices, along with an array representing the topological order of the graph.
The algorithm uses two-way topological traversal.

When a new edge (v, w) is added we perform the search if the vertices are out of order, or $w < v$. The algorithm alternates between searching backwards and forwards. The forward search starts at the topological position of w, we call this i. The search then increments i until a vertex is found at position i that has an edge from a vertex in F, initially $F = w$. This vertex is then added to F and the algorithm proceeds to search backwards in a similar fashion from j. This is repeated until the searches meet or $i = j$.

```
i = w.TopologicalPosition;
j = v.TopologicalPosition;

F = []; F.Push(i);
B = []; B.Push(j);

while (true)
    i++;
    while(i<j & no vertex in F has an edge to vertex i)
        i++;
    if (i == j)
        break;
    F.Push(i);
    j--;
    while(i < j & no vertex in B has an edge from vertex j)
        j--;
    if (i == j)
        break;
    B.Push(j);
```

After the traversal we move on to the second phase of the algorithm, cycle detection. Checking for a cycle is done by looking for an edge (u, z) such that u is in F and z is in B, if no such edge exists, we do not have a cycle. The remaining task is to reorder the affected area. We know that vertices before w and after v in the order are not affected by adding this edge and all vertices that need to be reordered are now contained in either F or B.

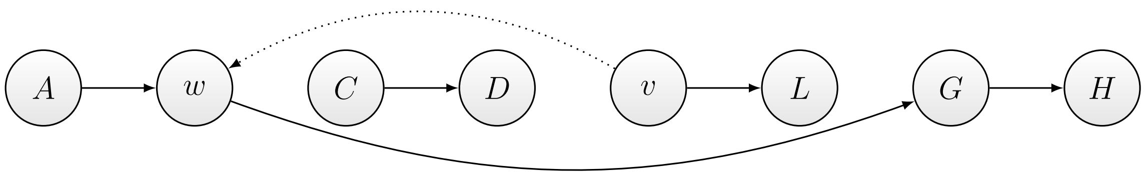


Figure 3.9: Reordering only affects edges $[w, C, D, v]$. C and D will not be included in F or B as they are not connected, however they will be shifted during the reordering phase.

The re-ordering is done by putting all vertices in F after the vertices in B. Some vertices might have to be moved to make room for the insertion as shown in Figure 3.9. Vertices C and D will be moved, but were not contained in F or B. This is trivial and is done by adding them to the back of the relevant list, so that they will be re-inserted last.

3.6 BFGT Dense $O(n^2 \log n)$

As the graphs grow denser, the two-way search method used in BFGT Sparse becomes less efficient. The algorithm uses a similar level strategy as its sparse counterpart but utilizes sophisticated methods to decide what edge to traverse. Each vertex stores an incoming edge (u, x) in a priority queue using the level of u as the key. This is done so that each traversal is more likely to increase the level of a vertex. Additionally the algorithm counts certain span traversals in the graph and uses this information to increase vertex levels more aggressively. The article proves that this maintains correctness of the algorithm. As the algorithm only searches forward, cycles are detected when the algorithm traverses into v. Additionally, the algorithm does not require a reordering phase as the traversal updates the topological order.

4
Supporting Algorithms and Data Structures

In this chapter we describe the detailed inner workings of the supported algorithms mentioned in the previous chapter. These supporting algorithms are either in the form of data structures or used to assist with pruning the search tree.

4.1 Order Maintenance Problem

The HKMST Sparse algorithm requires a data structure that can perform insert, delete and compare the order of two items in constant amortized time. This data structure developed for the Order Maintenance Problem achieves those bounds. These are the requirements of the problem listed explicitly:

- Query(x, y): Given references to x and y, returns true if x appears before y in the total order.
- Insert(x, y): Given a reference to x, inserts y right after it in the total order and returns a reference to the inserted item.
- Delete(x): Given a reference to x, removes it from the order.

Why is this difficult? It is easy to see that an array will not work efficiently, since every time an element is inserted into the middle of the list, all the items right of it have to be shifted. Another possibility would be to use a linked list to implement this. This would solve the problem of shifting elements, however the list would have to be traversed to answer the question if $x > y$.

4.1.1 Implementation

P. Dietz and D. Sleator released a paper in 1987 [5] with a solution to this problem. Our implementation follows their guidelines with a few changes to better fit our problem. In essence we are assigning comparison labels to objects, while maintaining a range of unused labels for new inserts between two existing objects. The structure we implement is simply a binary tree, whose nodes are labeled for comparison. The labels are distributed in such a way that there is approximately an equal amount of empty labels between them in order to reduce the need for relabeling on inserts, and for this we use binary numbers as labels. Each level of the tree splits the binary range in half, thus the depth of the tree is limited to the number of bits used for the labels, and this is covered in more detail in Section 4.1.4. A way to counteract this is to use a self balancing binary tree, however most balanced tree structures use rotation of sub-trees in order to rebalance itself. This does not work well with the labeling scheme since all the nodes in the rotated subtree would have to be relabeled. To reduce relabeling we choose to use a Scapegoat tree, which does not use rotations.

4.1.2 Scapegoat Tree

A Scapegoat tree is a self-balancing binary tree that does not use rotations for balancing operations. This is crucial for the algorithm to minimize the number of elements that have to be relabeled after rebalancing the tree. The Scapegoat tree is not strictly balanced, that is, it does not have half the elements to the left of the root and half to the right of the root. Instead it holds a more relaxed balancing criteria consisting of two limitations:

- $size(left) \leq \alpha * size(node)$ & $size(right) \leq \alpha * size(node)$
- $height(tree) \leq \log_{1/\alpha} NodeCount + 1$

We say that the tree is α balanced, referring to the variable limiting the imbalance that can occur between the left and right children of each node and the overall tree height. With these properties the Scapegoat tree is not guaranteed to be α-weight-balanced at all times but is always α-height-balanced. This is because during inserts we always make sure the tree is α-height-balanced by triggering a rebuild if the invariant is broken. However, a preceding node can become unbalanced. When the tree needs to rebalance to maintain its height, we traverse up the tree looking for a scapegoat, or a node which is not balanced. The tree rooted at the scapegoat is then rebuilt completely, making it perfectly balanced. This can be done in $O(n)$ time, since the nodes are already in sorted order. We traverse the subtree and recursively use the median node as the root of a new subtree.

4.1.3 Choosing α

A low alpha value makes the tree balance more strictly. When $\alpha = 0,5$, the tree balances like a regular binary tree. The way we are using the data structure, we never search the tree. The only operations we perform on the tree are insert and delete. A higher α value slows down uniformly distributed inserts [6], however we will mostly be doing sequential inserts and deletes. Figure 4.1 shows how a higher α value gives better results with sequential inserts. This means that we want to pick a high α value without exceeding depth equal to the number of bits used for the binary labeling. The number of bits used depends on the size of the graph the algorithm will run on. Choosing the α value following these guidelines will give us the best performance as it reduces the number of times the tree will be rebalanced.

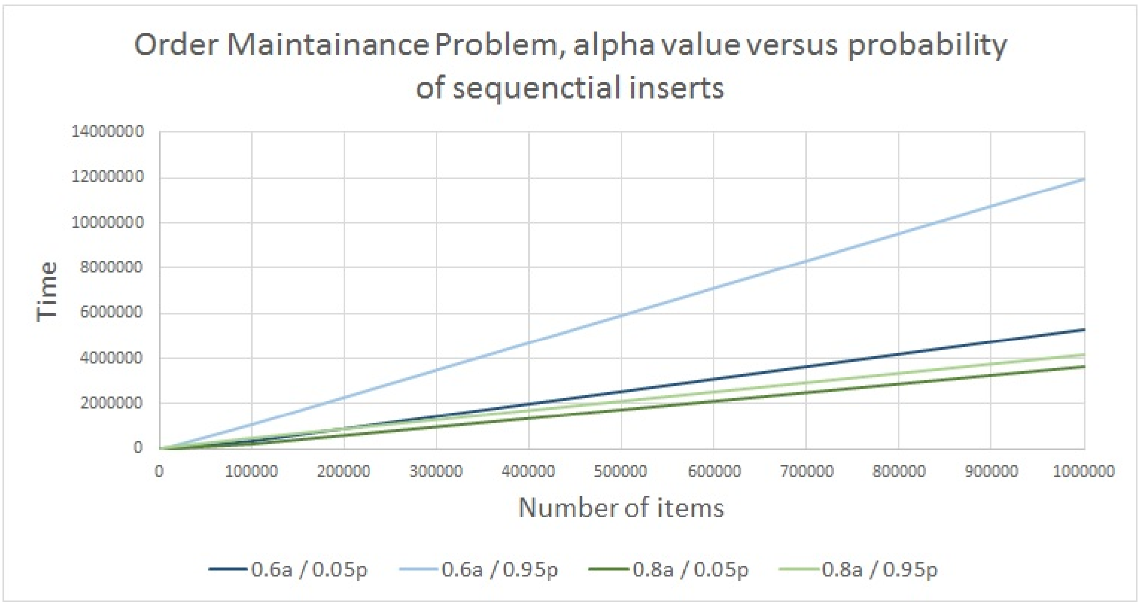


Figure 4.1: A more balanced tree (low alpha) causes sequential inserts to be more expensive.

α	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}	10^{11}	10^{12}	10^{13}	10^{14}	10^{15}	10^{16}
0,5	13	16	19	23	26	29	33	36	39	43	46	49	53
0,55	15	19	23	26	30	34	38	42	46	50	53	57	61
0,6	18	22	27	31	36	40	45	49	54	58	63		
0,65	21	26	32	37	42	48	53	58	64				
0,7	25	32	38	45	51	58							
0,75	32	40	48	56									
0,8	41	51	61										
0,85	56												

Table 4.1: Maximal depth of the tree in relation to α and number of items

4.1.4 Binary Labeling

To answer the question if x appears before y in the total order, we use a strategy that consists of giving each node in the tree a binary label. The root element is positioned in the middle of the total order, thus we give it the label $int.max/2 + 1$ or 1000..0 (where the number of bits depends on the integer type used). From there it is simple to build labels for new nodes with two operations we call $PathLeft$ and $PathRight$. Both operations find the least significant bit that is already set to 1 in the given label. PathLeft shifts the bit to the right and PathRight copies the bit to the right. This creates a new label that is smaller or bigger respectively and is in the middle of the range of labels we have to choose from.

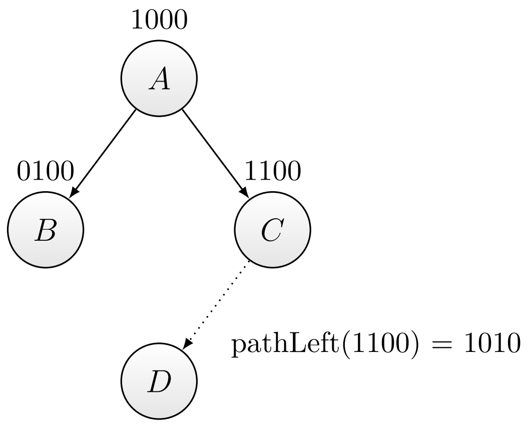


Figure 4.2: A new label is generated for an inserted node from its parent label using bit operators.

These labels are created with minimal work by utilizing bitwise operations. We take advantage of a property of Two's Complement to find the least significant bit that is set to 1. In Two's Complement the most significant bit is the sign bit and negative numbers count backwards from 1...111.

Decimal	Two's Complement
2	0010
1	0001
0	0000
−1	1111
−2	1110

Table 4.2: Example of Two's Complement binary representation of a 4-bit number.

Since zero is represented with the positive numbers, positive and negative numbers are off by one bit. This allows us to use the *AND* operator on a positive and a negative number to isolate the bit we need. The bit can be either shifted or copied to the right to create a smaller or bigger label respectively.

```
// least significant bit moved right
private long pathLeft(long label)
{
    long lastBit = (label AND -label);
    return (label XOR lastBit) | (lastBit >> 1);
}
```

The number of bits puts a constraint on the maximum depth we can allow the Scapegoat tree to reach. This is not a problem since we can use the α value in the scapegoat tree to control how many elements have to be inserted before reaching the maximum depth.

4.2 Finding The Median

HKMST-Sparse uses median finding in order to minimize its worst case. A linear time algorithm is required to not increase the time complexity of the algorithm. In 1973 [4] it was realized that this could be done in linear time. Up until then it was assumed that the set had to be ordered to find the absolute median, or the i-th largest element. There are a few algorithms that accomplish this task with similar methods, such as the Median of Medians (Median of Fives) algorithm which does this in linear time. Another popular solution is QuickSelect, which is used in the well known QuickSort algorithm. QuickSelect is a randomized algorithm resulting in a worse time complexity if compared to Median of Medians. However it has been shown to give better performance in practice.

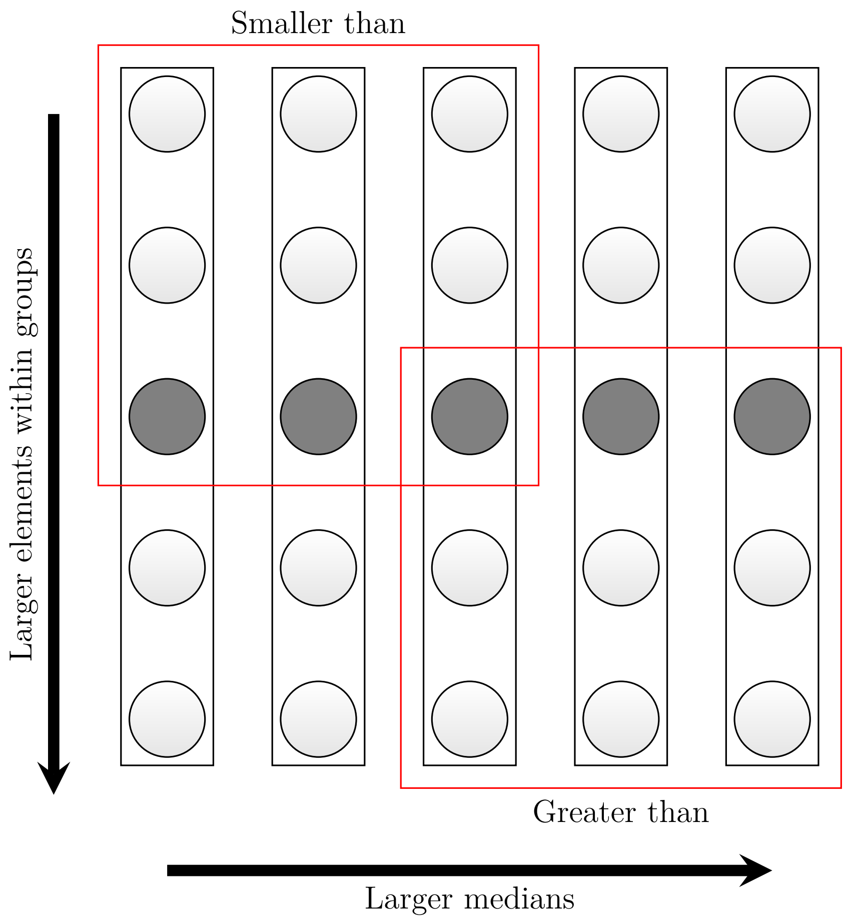


Figure 4.3: Each bucket contains a median shown as a gray node, buckets are sorted by medians.

4.2.1 Median of Medians

The median of medians algorithm is relatively simple in its design. We have a list of items and want to find the item at a certain rank or position. We split the list of items into buckets containing 5 elements each; these buckets are then sorted

internally to find the median of each bucket. As we see in Figure 4.3, the buckets are then sorted by the medians. Now we can compare the rank or position of the median of medians with the rank we are looking for. If the item we are looking for is smaller than the current median of medians, we repeat the process on the relevant items, "Smaller than" box in Figure 4.3. If the item is larger we perform the same operation on the "Greater than" box. Finally, if the item we are looking for is the median of medians, we have now found its rank in the list.
It should be noted that Figure 4.3 only shows 5 buckets, in practice there are $n/5$ buckets for the first iteration. Without diving into the detailed analysis of the algorithm, the main requirement is that we reduce the search space in each iteration. We examine that the median of medians is larger than half of the two top rows, or $2n/10$, additionally it is also larger than half of the medians or $1n/10$. This shows that each iteration reduces the search space to $3n/10$.

4.2.2 Median of Medians With Random

The biggest drawback of using Median of Medians is the increase of constant factors it introduces. This is the reason most practical implementation use a random pivot strategy. As an experiment and proof of concept we made a hybrid version of the algorithm which picks a pivot from the group of medians, or the middle row in Figure 4.3. As expected, this performed slightly better than the original implementation as seen in the results in Section 5 .

4.2.3 QuickSelect

QuickSelect is a randomized algorithm widely used for this type of problems in practice. In essence, it selects an item at random and pivots the list around it, repeating this process on the relevant side of the pivoted list until the i-th largest item is found. As expected, this method proved to be quite a lot quicker in practice. However its worse case, is quadratic, this can occur when the random pivot only eliminates one element at a time.

5 Results

The results show that the choice of algorithm is very dependent on the density of the graph. In most real world scenarios imaginable, graphs are most likely super sparse, in the $0 - 2\%$ density range. As the results below highlight, BFGT-Sparse and HKMST-Sparse seem to be the fastest algorithms for this particular range. The graphs below are of two types: the first type shows the time it takes to add the edges in the density range between the data points; the second one shows the percentage of edges added within a specific amount of time per edge. The latter can be helpful to see which algorithms are more suited to specific time limits per edge addition. It should be noted that in the following graphs HKMSTV1 is the basic version of HKMST-Sparse, using priority queues instead of soft-threshold search.

5.1 HKMST selection method

One of the methods used by HKMST-Sparse to reduce its worst case is a median selection algorithm to select compatible edges for traversal. According to our findings, selecting this edge at random gives the best results, as seen in Figure 5.1. However, the difference is relatively small, which adds value to the median selection method.

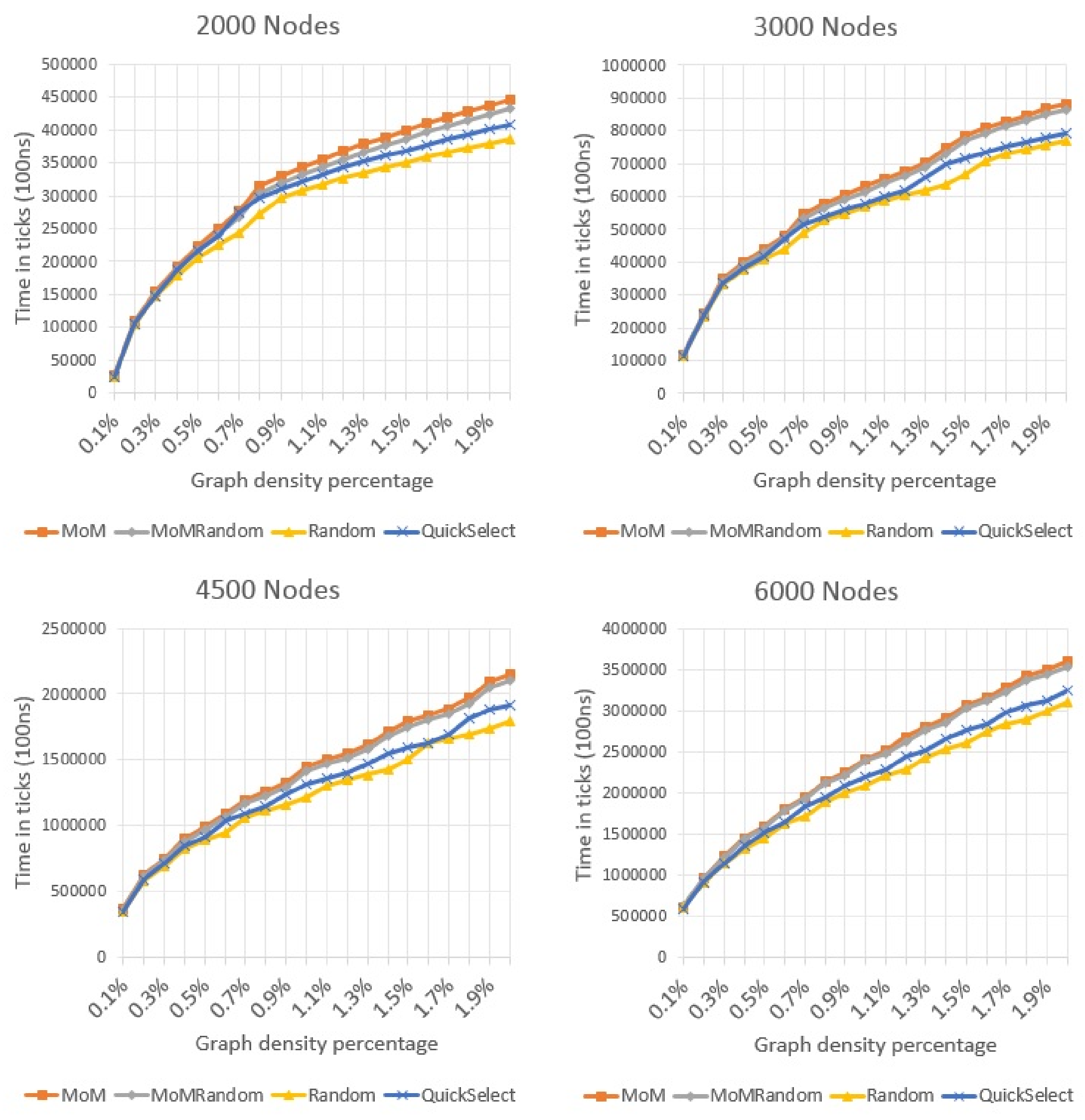


Figure 5.1: HKMST-Sparse selection methods on super sparse graphs. The Random method (yellow triangles) is the fastest for super sparse graphs

5.2 Super sparse graphs

BFGT-Sparse and HKMST-Sparse seem to be the fastest algorithms for super-sparse graphs. However HKMST-Sparse seems to pull ahead as the graphs become larger. BFGT-Sparse is still a good choice as its implementation is a lot simpler than that of HKMST-Sparse. It comes as a surprise that HKMSTV1 seems to perform better in this range than HKMST-Sparse as the latter is supposed to be an improvement of the former. This shows that, as with many other algorithms, lower theoretical time complexity does not always produce better results in practice.

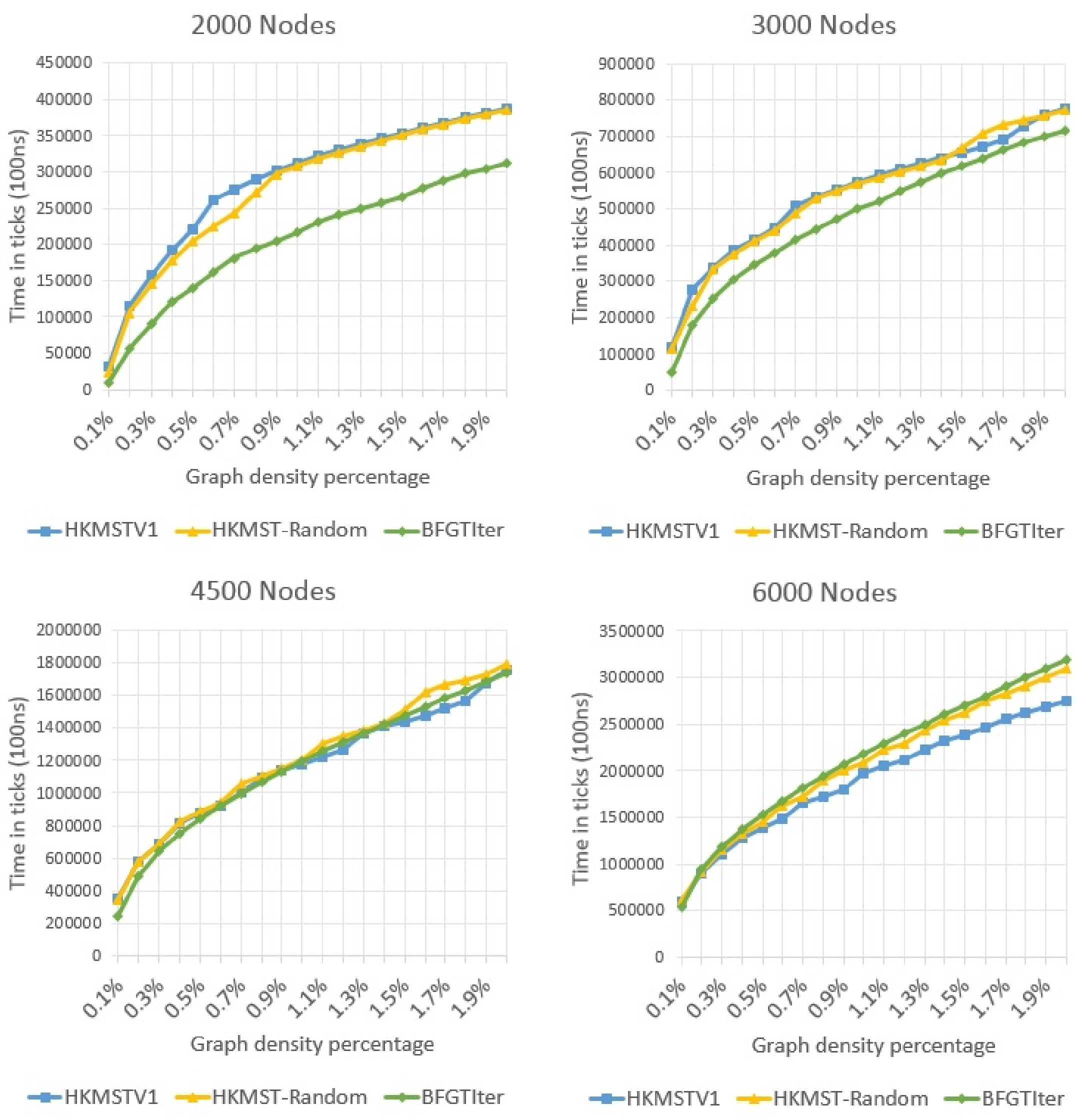
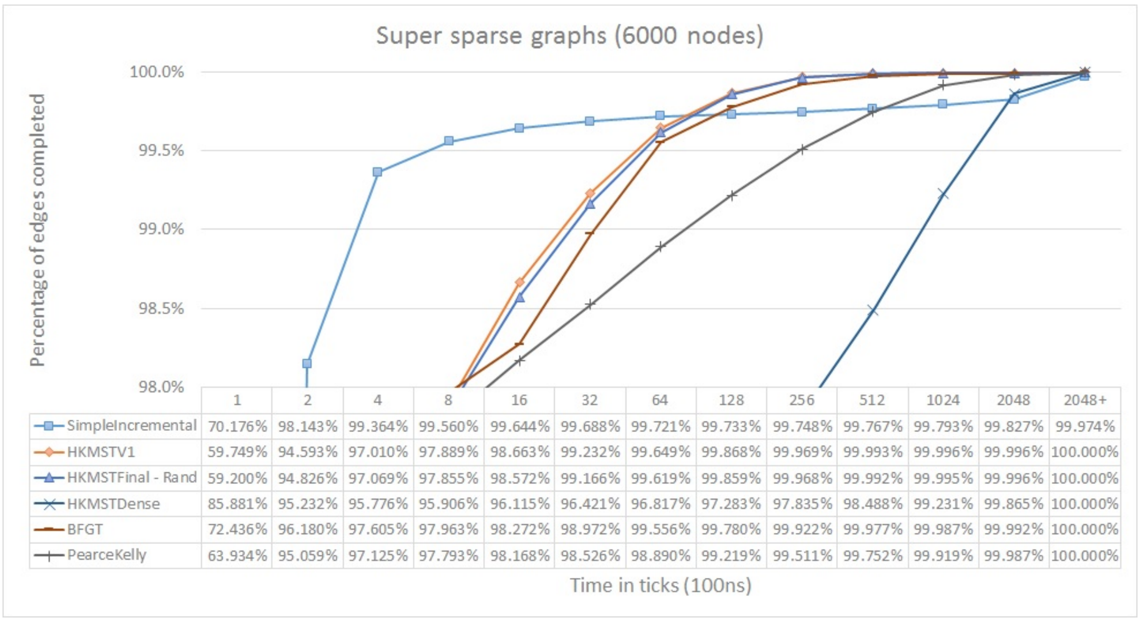


Figure 5.2: Performance on graphs of density varying from 0-2%.



	1	2	4	8	16	32	64	128	256	512	1024	2048	2048+
SimpleIncremental	70.176%	98.143%	99.364%	99.560%	99.644%	99.688%	99.721%	99.733%	99.748%	99.767%	99.793%	99.827%	99.974%
HKMSTV1	59.749%	94.593%	97.010%	97.889%	98.663%	99.232%	99.649%	99.868%	99.969%	99.993%	99.996%	99.996%	100.000%
HKMSTFinal - Rand	59.200%	94.826%	97.069%	97.855%	98.572%	99.166%	99.619%	99.859%	99.968%	99.992%	99.995%	99.996%	100.000%
HKMSTDense	85.881%	95.232%	95.776%	95.906%	96.115%	96.421%	96.817%	97.283%	97.835%	98.488%	99.231%	99.865%	100.000%
BFGT	72.436%	96.180%	97.605%	97.963%	98.272%	98.972%	99.556%	99.780%	99.922%	99.977%	99.987%	99.992%	100.000%
PearceKelly	63.934%	95.059%	97.125%	97.793%	98.168%	98.526%	98.890%	99.219%	99.511%	99.752%	99.919%	99.987%	100.000%

Figure 5.3: Performance on graphs of density varying from 0-2%.

We see how the simplicity of Simple Incremental makes it extremely fast for the majority of edges or around 98%, however the remaining 2% are very slow resulting in terrible performance. The other algorithms are very even, as expected from the previous graph.

5.3 Sparse graphs

The trends seen in the super sparse graphs continue in the sparse graphs. We see that HKMSTV1 increases its gap as the density grows. Additionally, we see that the Pearce and Kelly algorithm was not a contender for the super-sparse graphs but it starts performing better in relation to the other algorithms as the graphs grow denser.

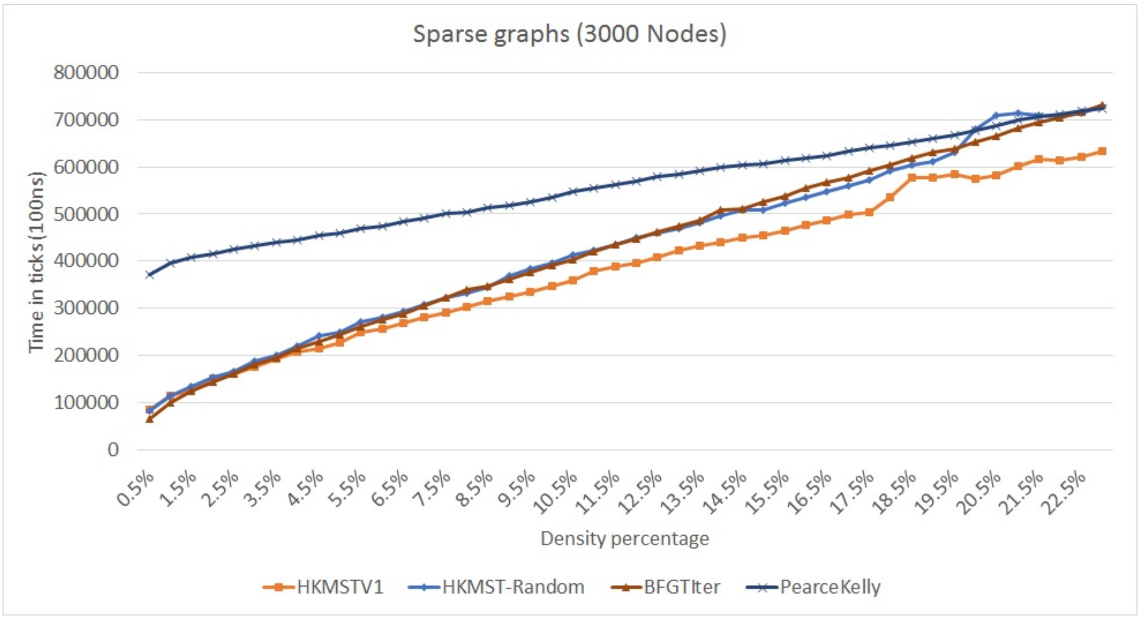
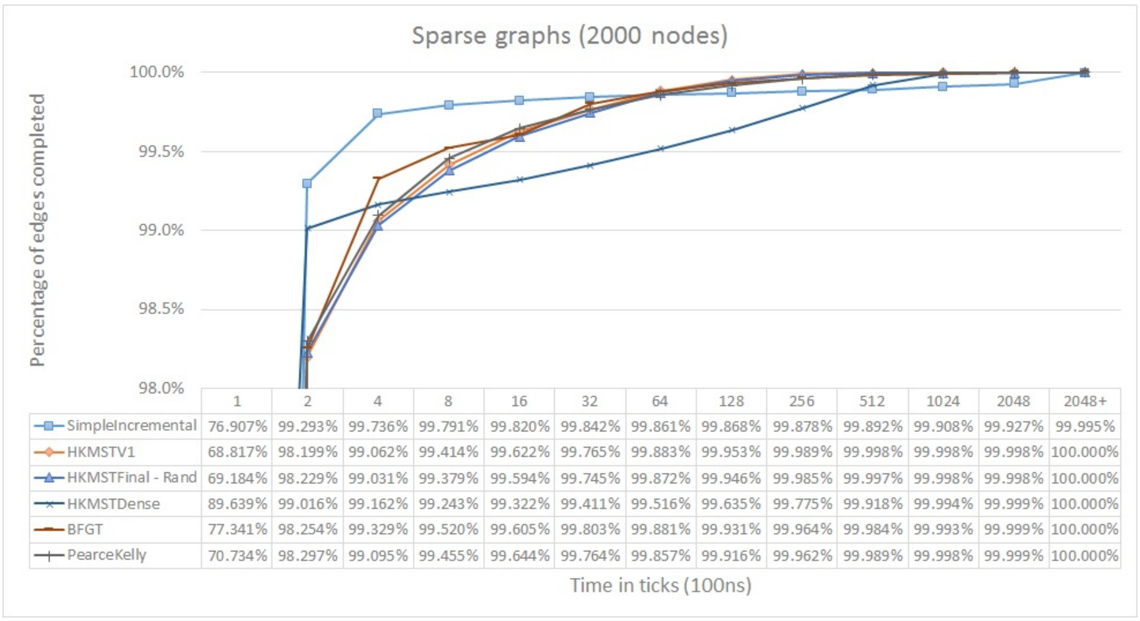


Figure 5.4: Performance on graphs of density varying from 0-23%.



	1	2	4	8	16	32	64	128	256	512	1024	2048	2048+
SimpleIncremental	76.907%	99.293%	99.736%	99.791%	99.820%	99.842%	99.861%	99.868%	99.878%	99.892%	99.908%	99.927%	99.995%
HKMSTV1	68.817%	98.199%	99.062%	99.414%	99.622%	99.765%	99.883%	99.953%	99.989%	99.998%	99.998%	99.998%	100.000%
HKMSTFinal - Rand	69.184%	98.229%	99.031%	99.379%	99.594%	99.745%	99.872%	99.946%	99.985%	99.997%	99.998%	99.998%	100.000%
HKMSTDense	89.639%	99.016%	99.162%	99.243%	99.322%	99.411%	99.516%	99.635%	99.775%	99.918%	99.994%	99.999%	100.000%
BFGT	77.341%	98.254%	99.329%	99.520%	99.605%	99.803%	99.881%	99.931%	99.964%	99.984%	99.993%	99.999%	100.000%
PearceKelly	70.734%	98.297%	99.095%	99.455%	99.644%	99.764%	99.857%	99.916%	99.962%	99.989%	99.998%	99.999%	100.000%

Figure 5.5: Performance on graphs of density varying from 0-23%.

We see that the Pearce and Kelly algorithm falls in line with the other sparse algorithms, this shows that it doesn't have a higher number of bad cases but an overall slower performance on sparse graphs. This is particularly interesting as its performance gets better on medium density graphs.

5.4 All-round

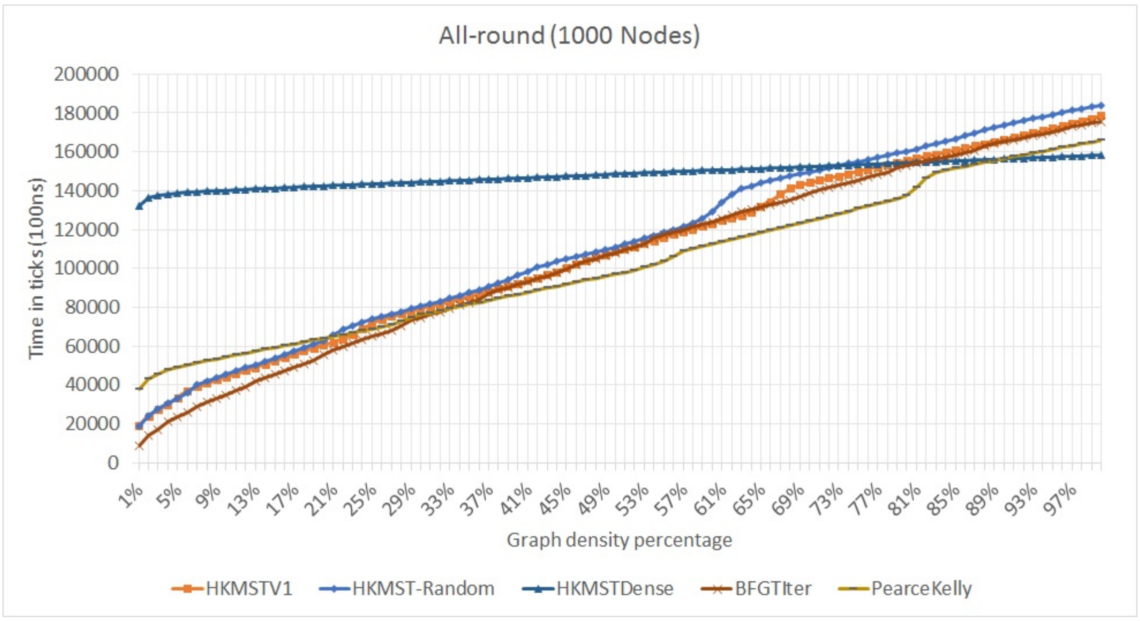


Figure 5.6: Performance on graphs of density varying from 0-99%.

The best performing algorithm on medium density graphs is the Pearce and Kelly algorithm. HKMSTV1 is also a good all-round algorithm, as it is not much slower than Pearce and Kelly's on medium density graphs but performs quite a lot better on the sparse graphs.

5.5 Dense graphs

The Pearce and Kelly algorithm proves to be quite fast on all densities of graphs, however HKMST-Dense proves to be faster as the graphs become both bigger and extremely dense. We see that once the number of nodes is increased to 2000, HKMST-Dense becomes faster around 83% density as opposed to 90% for 1000 nodes. It should be noted that the results of BFGT Dense are not included as we could not get the algorithm to perform within expected limits. It performed about 20 times slower than the other algorithms, but by a constant factor. It is uncertain whether it is a fault in the implementation or an error in the article.

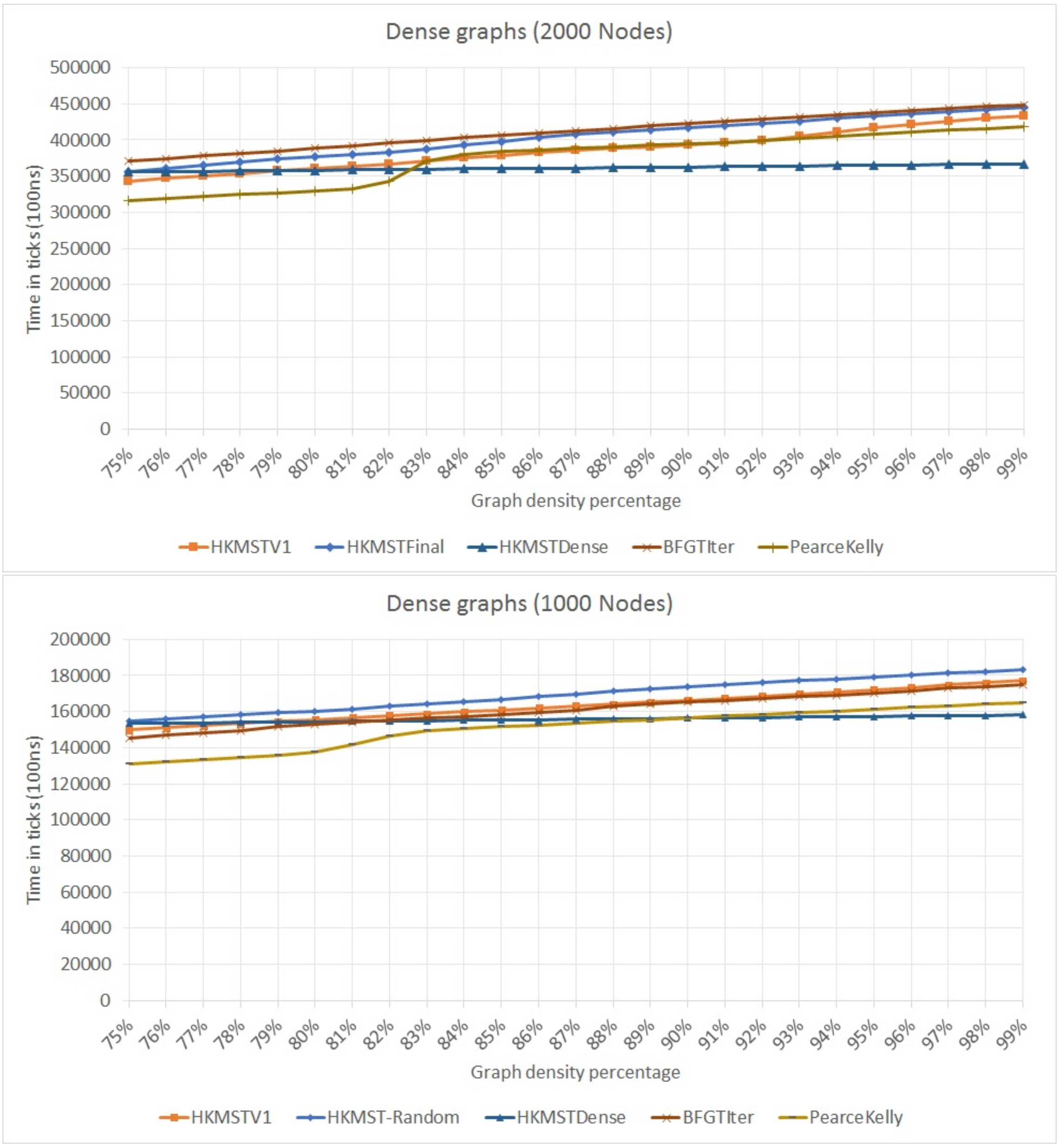
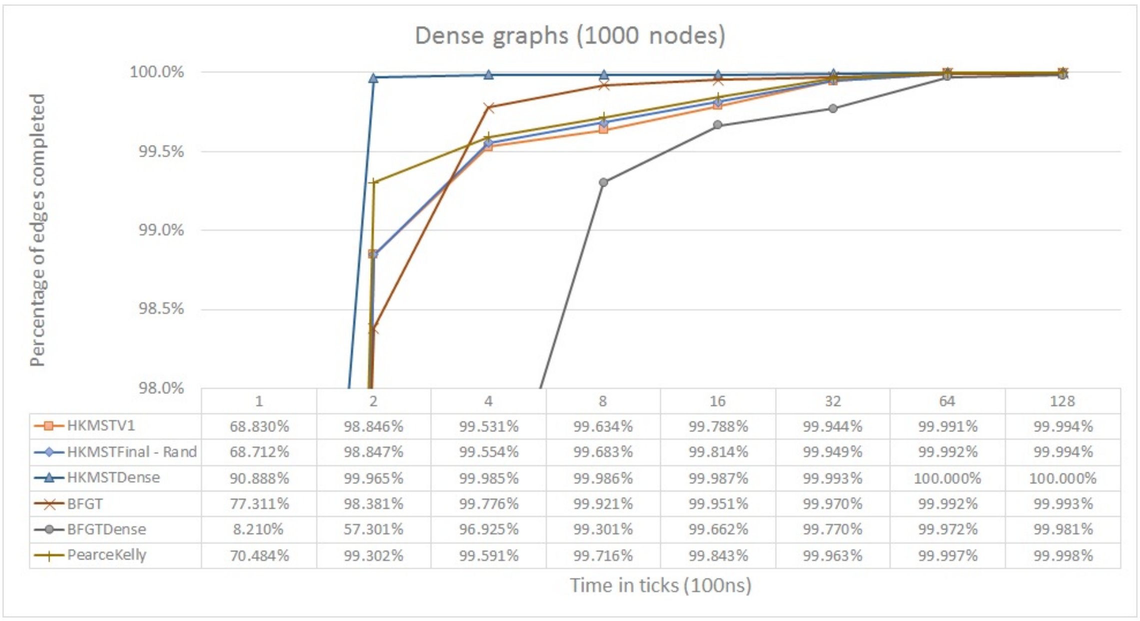


Figure 5.7: Performance on graphs of density varying from 75-99%.



	1	2	4	8	16	32	64	128
HKMSTV1	68.830%	98.846%	99.531%	99.634%	99.788%	99.944%	99.991%	99.994%
HKMSTFinal - Rand	68.712%	98.847%	99.554%	99.683%	99.814%	99.949%	99.992%	99.994%
HKMSTDense	90.888%	99.965%	99.985%	99.986%	99.987%	99.993%	100.000%	100.000%
BFGT	77.311%	98.381%	99.776%	99.921%	99.951%	99.970%	99.992%	99.993%
BFGTDense	8.210%	57.301%	96.925%	99.301%	99.662%	99.770%	99.972%	99.981%
PearceKelly	70.484%	99.302%	99.591%	99.716%	99.843%	99.963%	99.997%	99.998%

Figure 5.8: Performance on graphs of density varying from 75-99%.

We see that for HKMST-Dense no edge takes more than 4 ticks to add. The Pearce and Kelly algorithm begins to struggle when the graphs are almost complete, but remains the quickest, except for HKMST-Dense.

6
Conclusion

Very little data exists on the practical performance of new algorithms, therefore it is important to perform tests of this kind. In my opinion the Pearce & Kelly [8] algorithm stands out for its overall good performance over a wide variety of graph densities and exceptional simplicity. As Pearce & Kelly stated in their article, the algorithm does not have the best asymptotic time bound as they designed it for practical performance. This claim proves to be quite accurate, according to the results. HKMST-Sparse is the fastest algorithm for super sparse graphs in the range $0-2\%$, BFGT-Sparse is a strong contender and in my opinion much simpler to implement as it does not require a sophisticated data structure and supporting algorithms. For average to high density graphs, PK is the fastest algorithm. However, when graphs become extremely dense or around 80-85%, HKMST-Dense takes the lead. We hope the this thesis can serve as a guide to selecting the right algorithm for the job.

Density	Algorithm
Super sparse	HKMST-Sparse
Sparse	HKMST-Sparse
Dense	HKMST-Dense
Allround	Pearce & Kelly

Table 6.1: Choice of algorithm by density

6.1 Source code

The algorithms were all implemented and tested in C# 6.0. The source code is available at the url below and can be used to repeat the experiments.

Source code:
http://github.com/ragnarls08/EmpiricalTester

7 Discussion

Due to time constraints on the project, some aspects could not be explored, mainly parallelism and distributed processing. For the most part, the algorithms we covered are not designed for distributed processing. In the case of HKMST Dense, the re-ordering phase can be run in parallel as the backward and forward sets are completely independent. This could give a small performance increase if the algorithm is running on sufficiently large graphs to cover the overhead cost of parallelism.
As the rise in processing speed has started leveling out in recent years with multi-core processing and distributed systems becoming the way forward, this is an interesting factor for future research in the field.

Bibliography

[1] Alpern, B., Hoover, R., Rosen, B. K., Sweeney, P. F., and Zadeck, F. K. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 1990), SODA '90, Society for Industrial and Applied Mathematics, pp. 32–42.

[2] Belik, F. An efficient deadlock avoidance technique. *IEEE Transactions on Computers 39*, 7 (1990), 882–888.

[3] Bender, M. A., Fineman, J. T., Gilbert, S., and Tarjan, R. E. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms 12*, 2 (Dec. 2015), 14:1–14:22.

[4] Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. Time bounds for selection. *Journal of Computer and System Sciences 7*, 4 (1973), 448–461.

[5] Dietz, P., and Sleator, D. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1987), STOC '87, ACM, pp. 365–372.

[6] Galperin, I., and Rivest, R. L. Scapegoat trees. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 1993), SODA '93, Society for Industrial and Applied Mathematics, pp. 165–174.

[7] Haeupler, B., Kavitha, T., Mathew, R., Sen, S., and Tarjan, R. E. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms 8*, 1 (Jan. 2012), 3:1–3:33.

[8] Pearce, D. J., and Kelly, P. H. J. A dynamic topological sort algorithm for directed acyclic graphs. *J. Exp. Algorithmics 11* (Feb. 2007).

[9] Pearce, D. J., Kelly, P. H. J., and Hankin, C. Online cycle detection and difference propagation for pointer analysis. IEEE, pp. 3–12.

[10] Ploeg, A. v. d., and Claessen, K. Practical principled frp: Forget the past, change the future, frpnow! *SIGPLAN Not. 50*, 9 (Aug. 2015), 302–314.