Solving Large Graph Problems Using Tree Decompositions: A Computational Study

Chris Groër

Computational Math Group
Oak Ridge National Laboratory

October 18, 2011

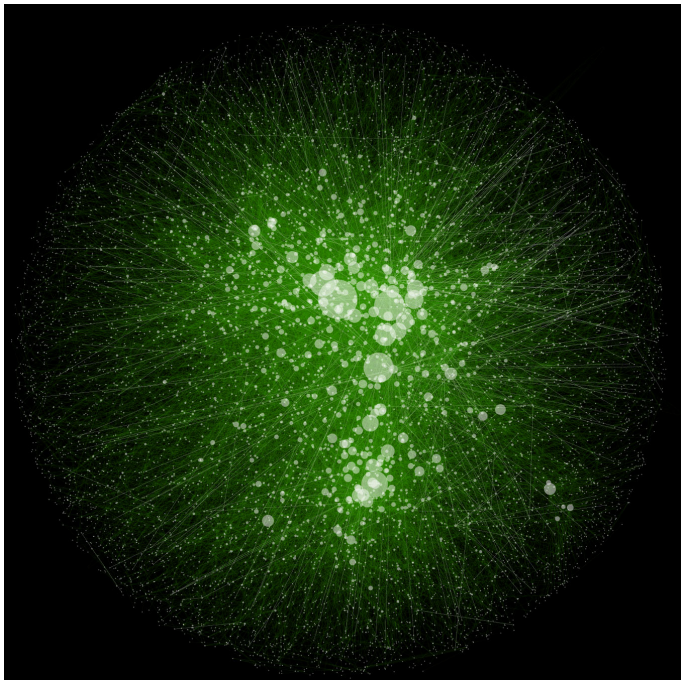
This is joint work with:
Blair Sullivan , Complex Systems Group, ORNL
Dinesh Weerapurage, Complex Systems Group, ORNL

U.S. DEPARTMENT OF ENERGY | Office of Science

OAK RIDGE National Laboratory

Background

- Large graphs are ubiquitous in a variety of diverse domains: biology, sociology, operations research
- When one wishes to solve discrete optimization problems involving such data sets, the scaling is often exponential in the size of the graph, typically thought of as the number of nodes (n) and the number of edges (m) in the graph

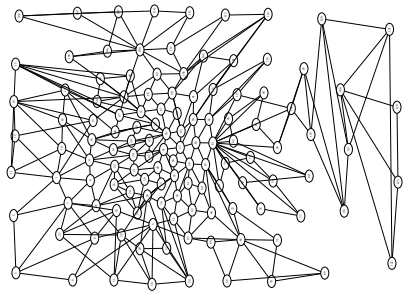


Biological interaction graph, NIH

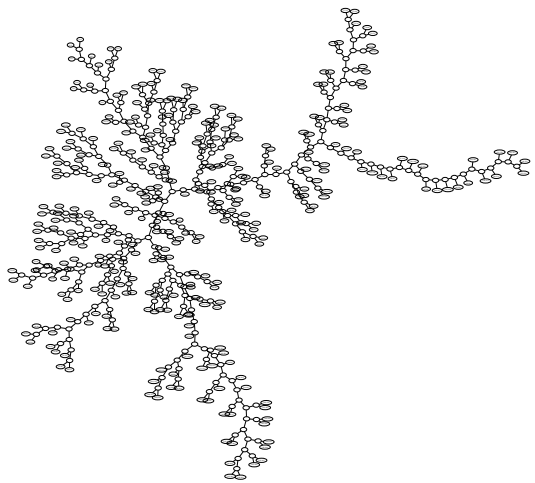
Background

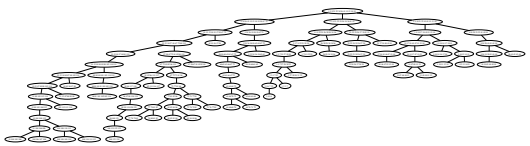
- Theoretical results tell us that tree and branch decompositions provide a framework for solving a wide variety of *NP*-hard problems on graphs in time that is polynomial in graph size and exponential in the *width* of the decomposition
- This change in scaling seems to offer possibilities for solving problems on certain types of graphs where n and m are very large

A graph with 124 nodes and 318 edges



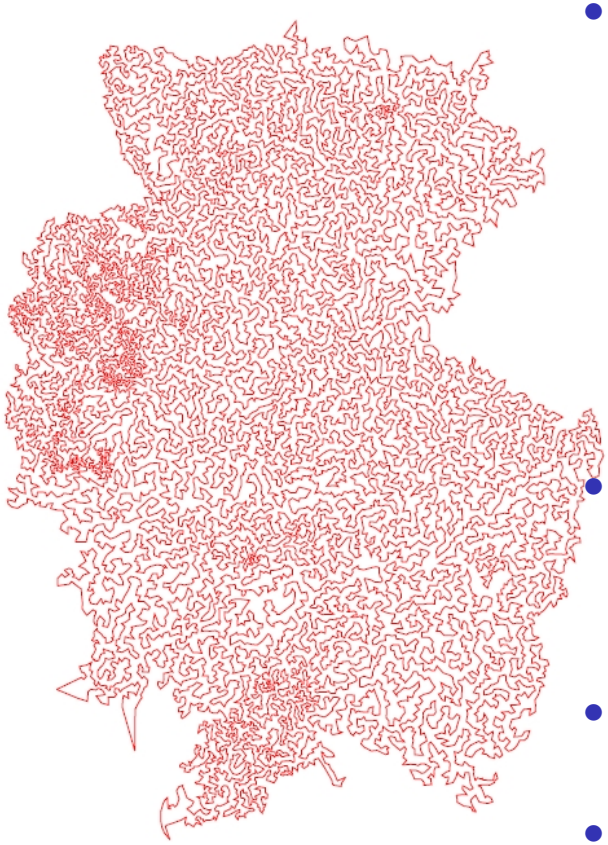
A Branch Decomposition





A Tree Decomposition

Background



A tour of 18512 German cities

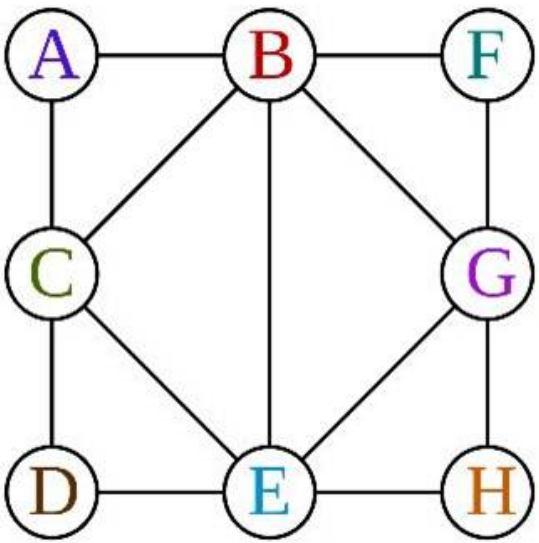
- Despite a large amount of research into methods for creating such decompositions, much less work into utilizing them for large-scale computational optimization problems
  - Cook & Seymour use tour-merging algorithm based on branch decompositions for large TSPLIB instances
  - Fomin, et al, have examined memory-saving strategies in dynamic programming
  - Koster, et al, solved Frequency Assignment Problem
  - Hicks has explored the utility of branch decompositions
- "As a rule of thumb, the typical border of practical feasibility lies somewhere below a treewidth of 20 for the underlying graph." [Hüffner, Niedermeier, Wernicke 2007]
- How do actual implementations of optimization algorithms scale in practice?
- Can they be used to solve large problems where better-known techniques fail?

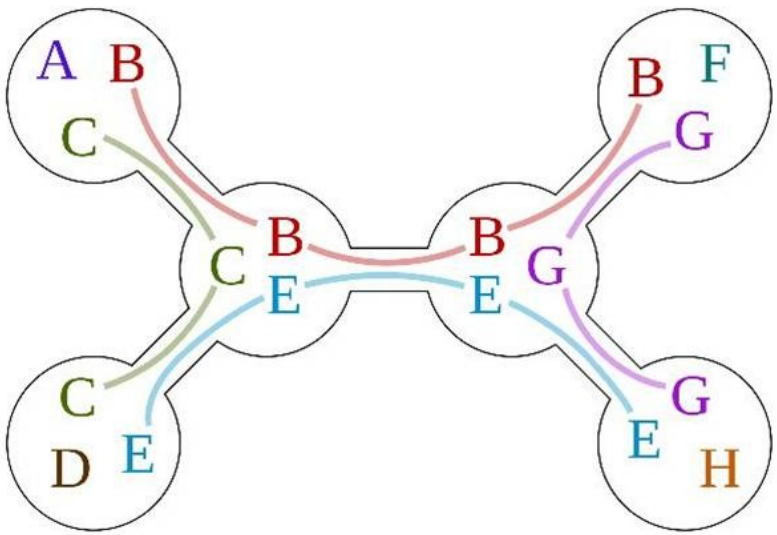
Outline of Talk

- Introduce tree decompositions
- Describe a dynamic programming algorithm for a well-known *NP*-hard problem
- Discuss specific implementation details
- Present computational results

Some Definitions

- We begin with a connected graph $G = (V, E)$ to be decomposed.
- A *tree decomposition* of G is a mapping of the vertices of V onto a tree T where each node $t \in T$ represents a subset or *bag* of vertices, denoted as X_t
  - Each edge from E must appear in some X_t
  - All bags containing some vertex $v \in V$ must form a connected subtree in T
- The *width* of a tree decomposition is the size of the largest bag minus one



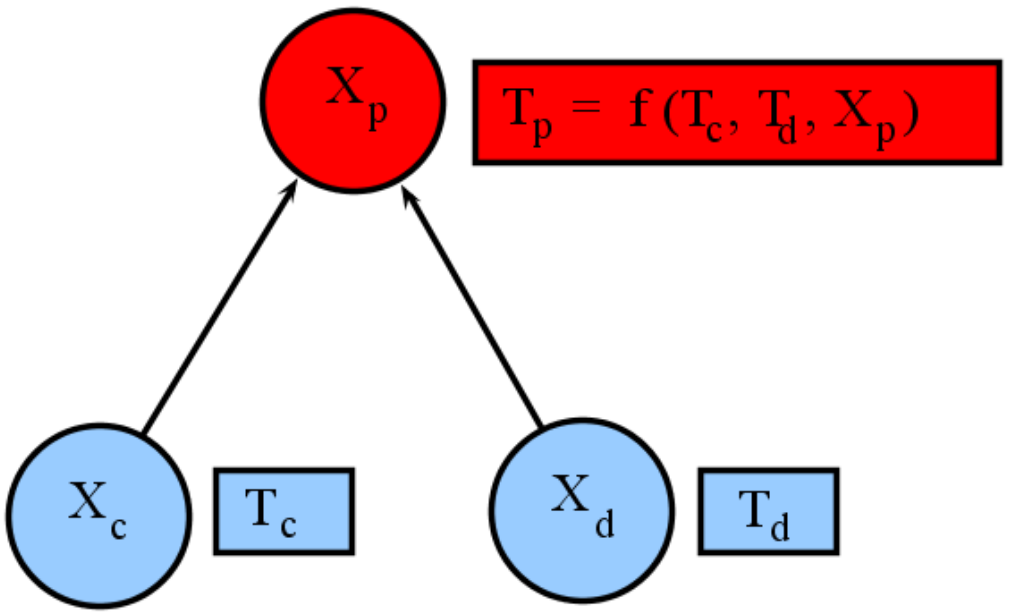


Constructing Tree Decompositions

- We use heuristics that share a link with numerical linear algebra where one often wants to permute the rows/columns of a matrix before computing a factorization so that the resulting factors are as sparse as possible
- This problem can be phrased as graph triangulation and the objective is to minimize the number of fill edges (not max clique size)
- Numerous fast implementations are available (METIS - minimum degree algorithm and nested node dissection, Approximate Minimum Degree (AMD), etc.)
- The time and memory required to construct a tree decomposition is much smaller than for a dynamic programming algorithm on the tree
- The *treewidth* of a graph G is the minimum width across all tree decompositions of G
- Determining the *treewidth* of a graph is generally *NP*-hard

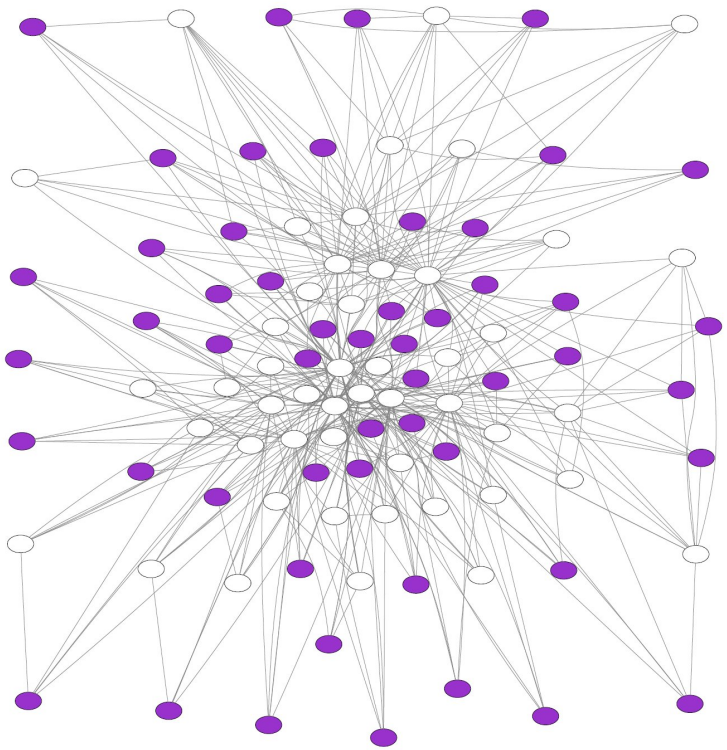
Using a tree decomposition for dynamic programming

- Arbitrarily root the tree and work upwards from the leaves to the root, analyzing the bags of the parent and children in a problem-specific manner
- At a leaf node c with a bag of size $|X_c|$, you typically consider all possible solutions (exponential in $|X_c|$)
- At a non-leaf node p, you must consider all partial solutions in the non-leaf node's bag and execute a dynamic programming step involving the solutions for node p and the tables stored for the child nodes



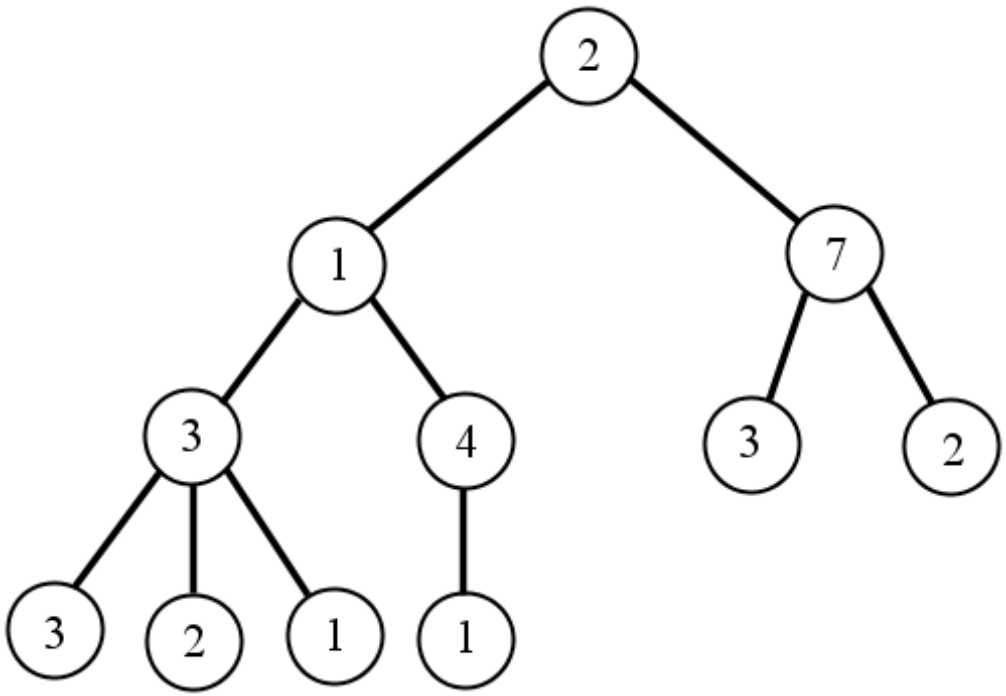
An Example Dynamic Programming Algorithm

- We chose a simple, well-studied problem and developed a careful implementation of
  - *Maximum weighted independent set*: Given a connected, weighted graph, determine the set of vertices with largest total weight such that no two vertices are adjacent



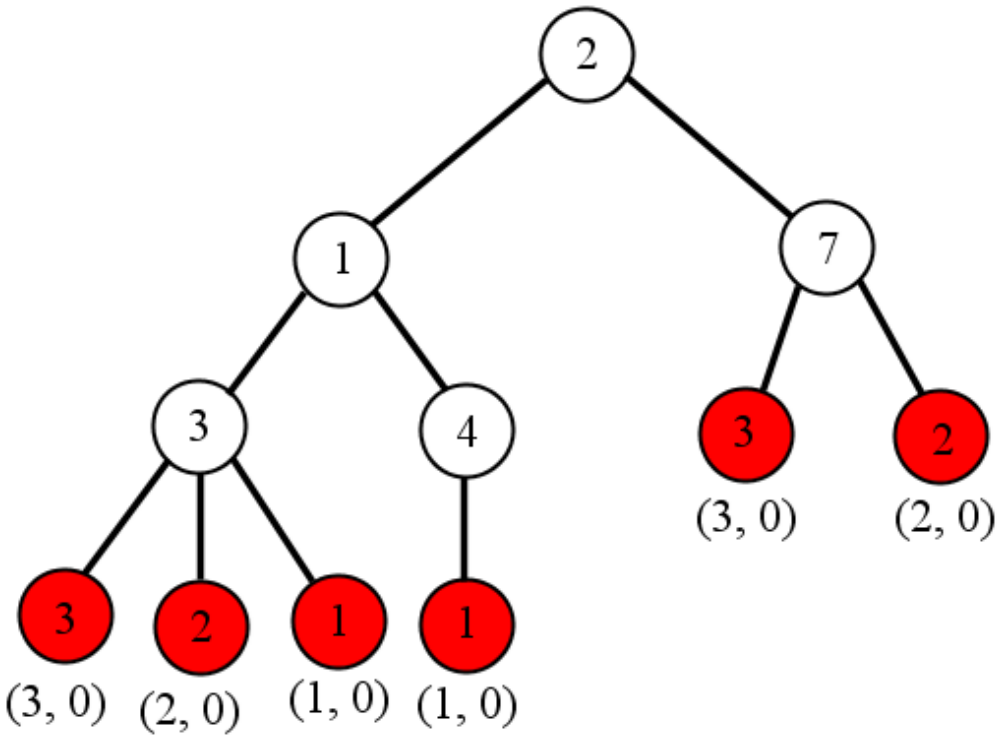
Maximum Weighted Independent Set is Easy on Trees

- At each node q in a rooted tree, we consider the subgraph (subtree) rooted at q
- We keep track of two quantities:
  - The weight of the MWIS in this subgraph that contains q
  - The weight of the MWIS in this subgraph that does not contain q
- The optimal solution is found by considering each case at the root node



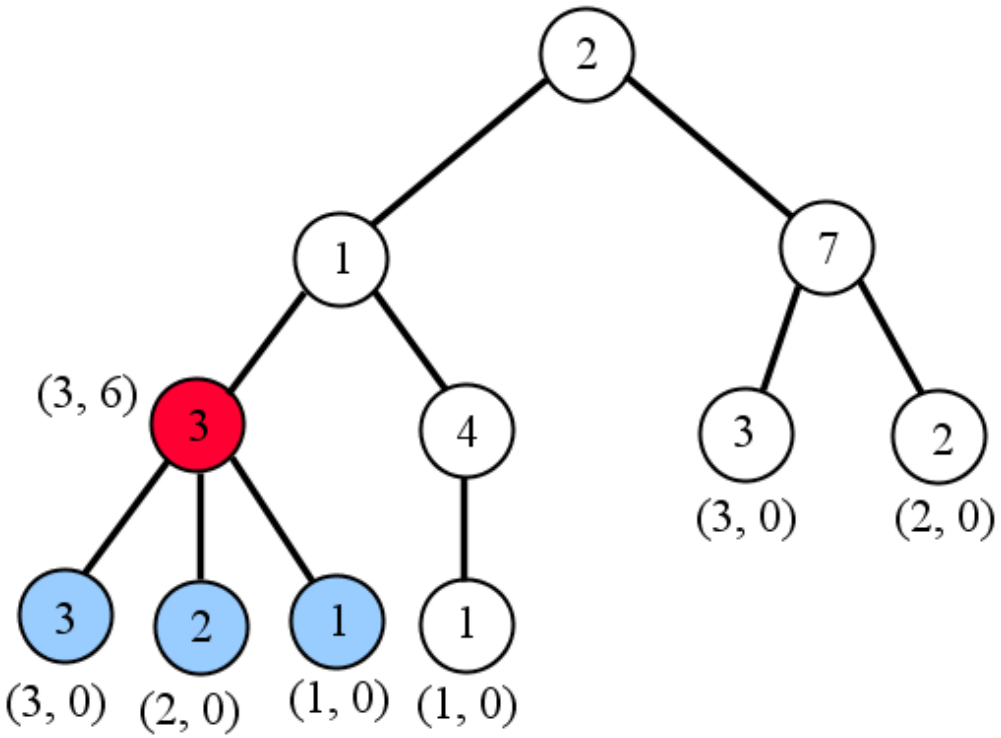
Maximum Weighted Independent Set is Easy on Trees

- At each node q in a rooted tree, we consider the subgraph (subtree) rooted at q
- We keep track of two quantities:
  - The weight of the MWIS in this subgraph that contains q
  - The weight of the MWIS in this subgraph that does not contain q
- The optimal solution is found by considering each case at the root node



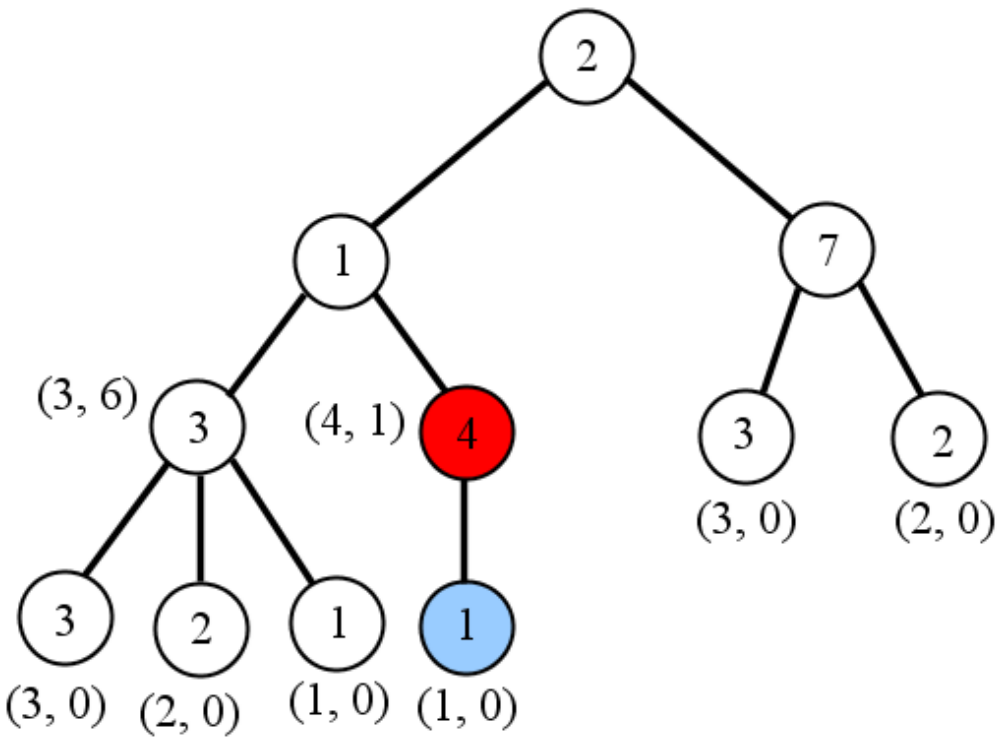
Maximum Weighted Independent Set is Easy on Trees

- At each node q in a rooted tree, we consider the subgraph (subtree) rooted at q
- We keep track of two quantities:
  - The weight of the MWIS in this subgraph that contains q
  - The weight of the MWIS in this subgraph that does not contain q
- The optimal solution is found by considering each case at the root node



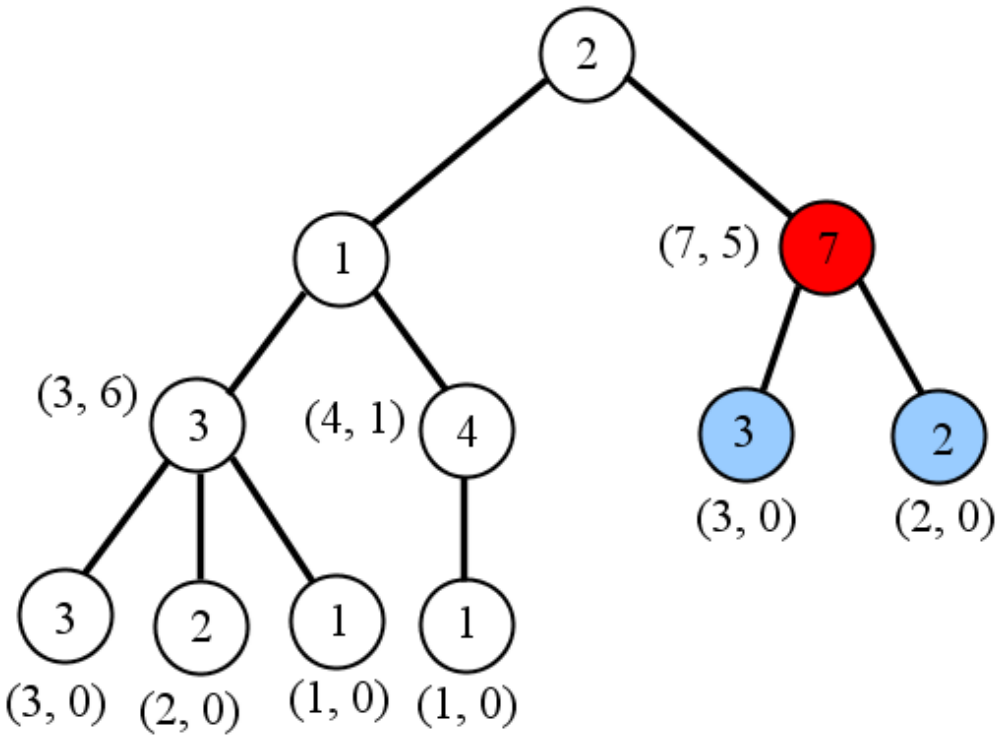
Maximum Weighted Independent Set is Easy on Trees

- At each node q in a rooted tree, we consider the subgraph (subtree) rooted at q
- We keep track of two quantities:
  - The weight of the MWIS in this subgraph that contains q
  - The weight of the MWIS in this subgraph that does not contain q
- The optimal solution is found by considering each case at the root node



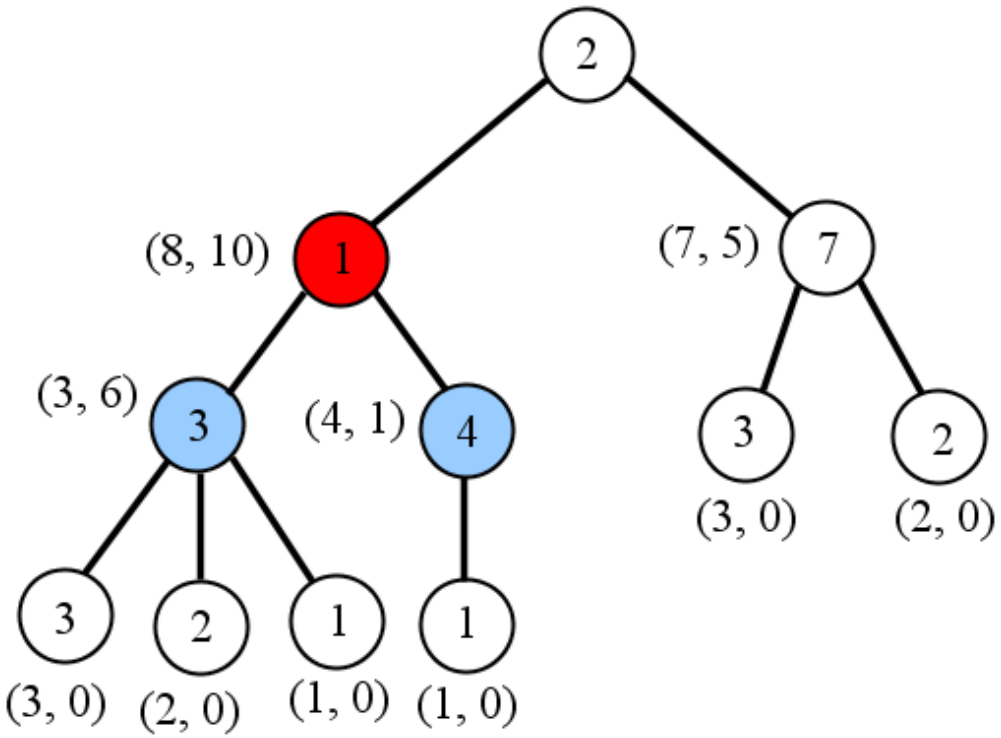
Maximum Weighted Independent Set is Easy on Trees

- At each node q in a rooted tree, we consider the subgraph (subtree) rooted at q
- We keep track of two quantities:
  - The weight of the MWIS in this subgraph that contains q
  - The weight of the MWIS in this subgraph that does not contain q
- The optimal solution is found by considering each case at the root node



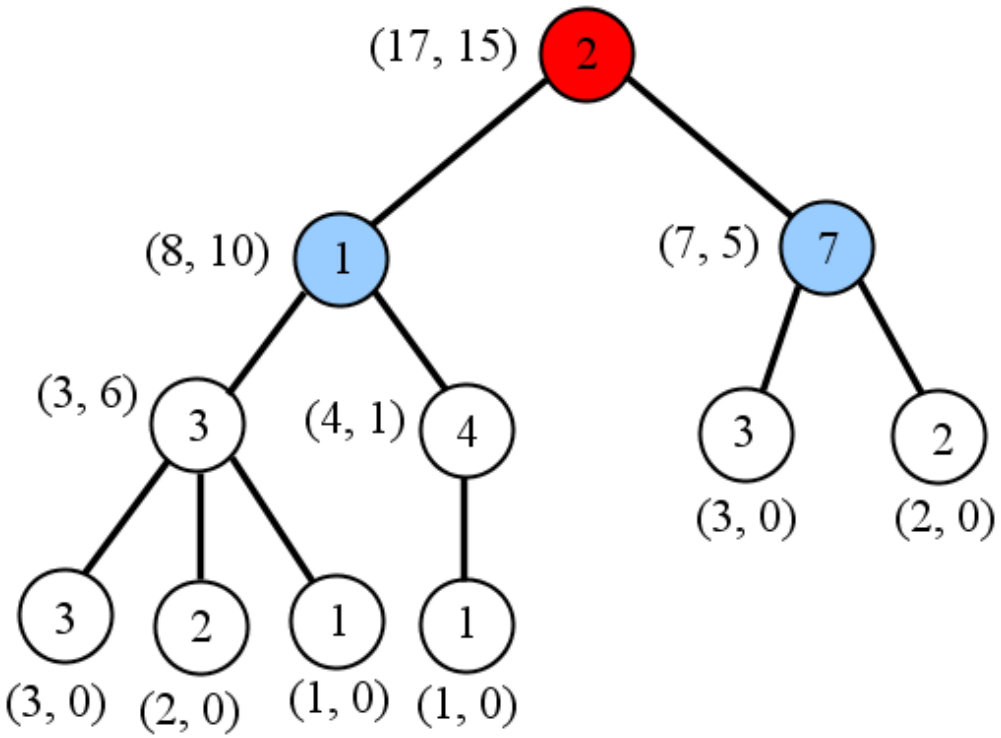
Maximum Weighted Independent Set is Easy on Trees

- At each node q in a rooted tree, we consider the subgraph (subtree) rooted at q
- We keep track of two quantities:
  - The weight of the MWIS in this subgraph that contains q
  - The weight of the MWIS in this subgraph that does not contain q
- The optimal solution is found by considering each case at the root node



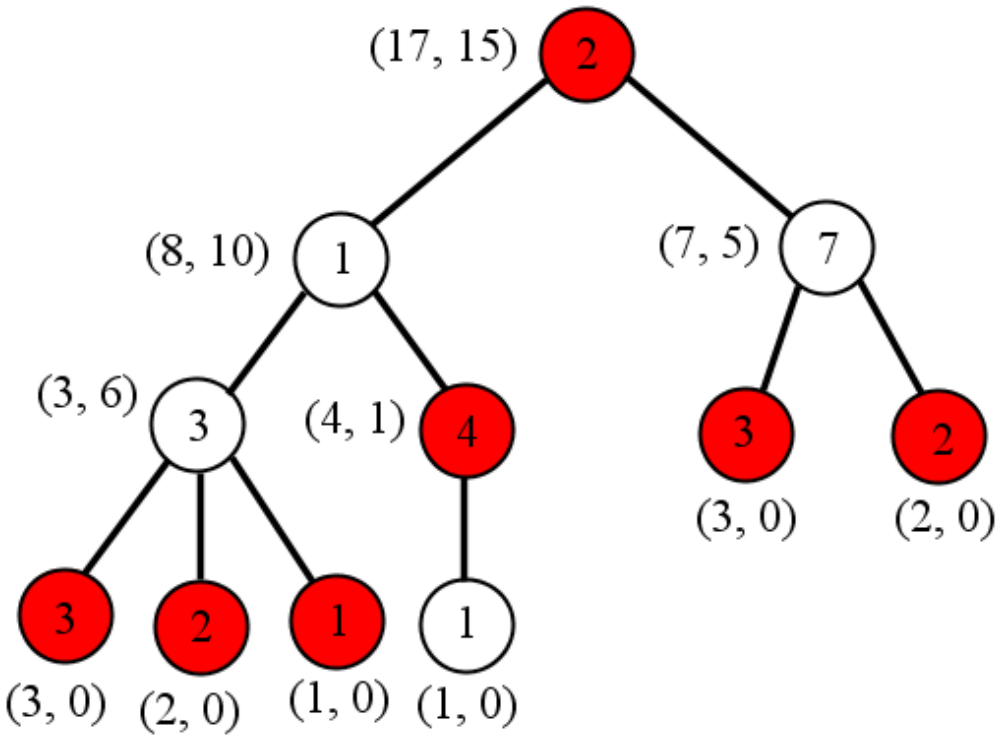
Maximum Weighted Independent Set is Easy on Trees

- At each node q in a rooted tree, we consider the subgraph (subtree) rooted at q
- We keep track of two quantities:
  - The weight of the MWIS in this subgraph that contains q
  - The weight of the MWIS in this subgraph that does not contain q
- The optimal solution is found by considering each case at the root node



Maximum Weighted Independent Set is Easy on Trees

- At each node q in a rooted tree, we consider the subgraph (subtree) rooted at q
- We keep track of two quantities:
  - The weight of the MWIS in this subgraph that contains q
  - The weight of the MWIS in this subgraph that does not contain q
- The optimal solution is found by considering each case at the root node



Solving Maximum Weighted Independent Set

- A similar idea allows us to develop a dynamic programming algorithm using tree decompositions
- For a leaf node, we record each independent set contained in the bag along with its weight
- Given a non-leaf tree node t, consider an independent set $S \subseteq X_t$ with weight $w(S)$
- For each such S, record its value $f_t(S)$ in the table to be the weight of the maximum independent set in the subgraph induced by the vertices beneath t in the tree whose intersection with X_t is S
- If t has children tree nodes $1, 2, \ldots, c$, then $f_t(S)$ can be calculated via dynamic programming:

$$f_t(S) = w(S) + \sum_{i=1}^{c} \max(f_i(T_i) - w(T_i \cap S) | T_i \cap X_t = S \cap X_i)$$

Some Implementation Details

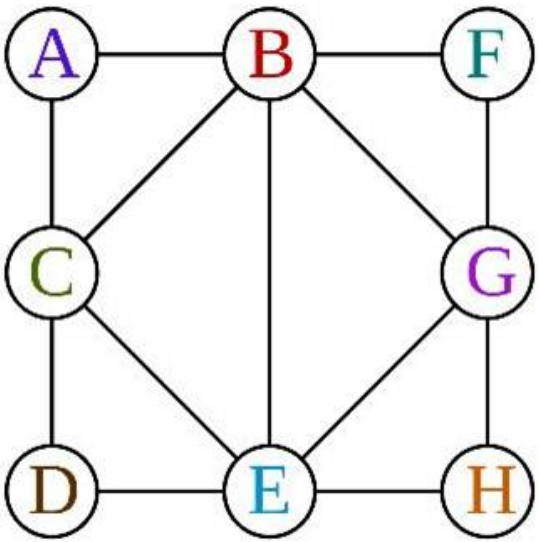
- Extensive use of hash tables to look up $f_t(S)$ values in child tables
- We represent a subset $S \in X_t$ as a bit vector composed of (potentially multiple) 64-bit words where a 1-bit in position i indicates that the i-th entry in X_t is in S
- The binary representation is convenient to rule out many of the $2^{|X_t|}$ possibilities in a single stroke
  - Suppose the edge $(7, 11)$ exists in G so that any set S containing both 7 and 11 is not independent

X_t	13	11	7	5	3	2	1
Mask	0	1	1	0	0	0	0

  - Any mask of the form $011xxxx$ cannot represent an independent set and we eliminate 2^4 possibilities at once
- This and other tricks typically allow us to skip many possibilities in a for loop from 0 to $2^{|X_t|}$

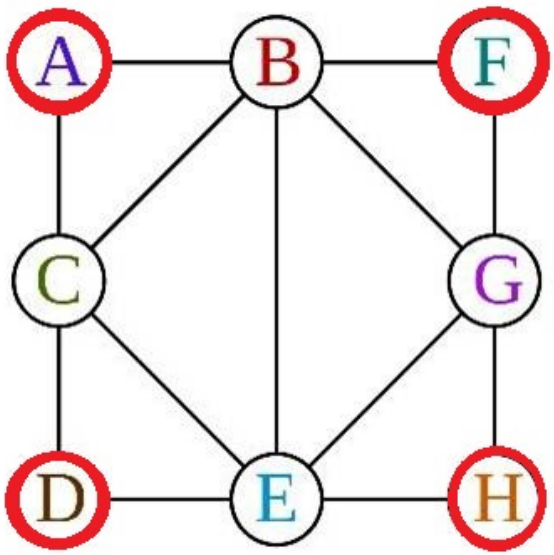
Some Implementation Details

- We also developed some techniques for saving memory by limiting what is stored in the dynamic programming tables
- Rather than storing all independent sets in each treenode's table, we only store information about the independent sets' intersection with the parent's bag since that is all that matters in the recursion
- Instead of storing all dynamic programming tables in memory, we delete them as we move up the tree
- Then we have some partial information about the optimal solution when we arrive at the root node
- We use this information to *refine* the tree and decrease the width by removing vertices from consideration



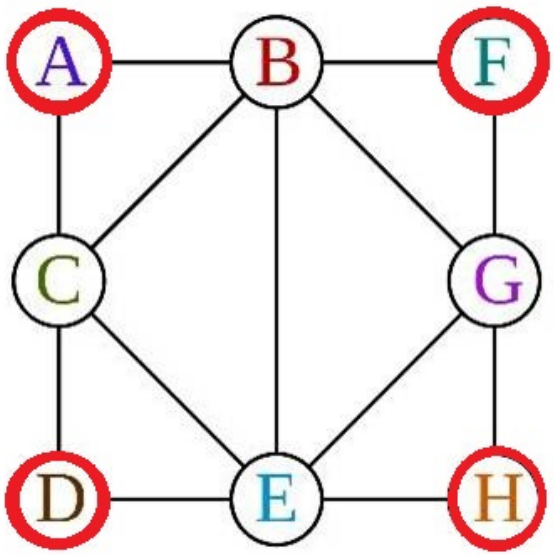
Some Implementation Details

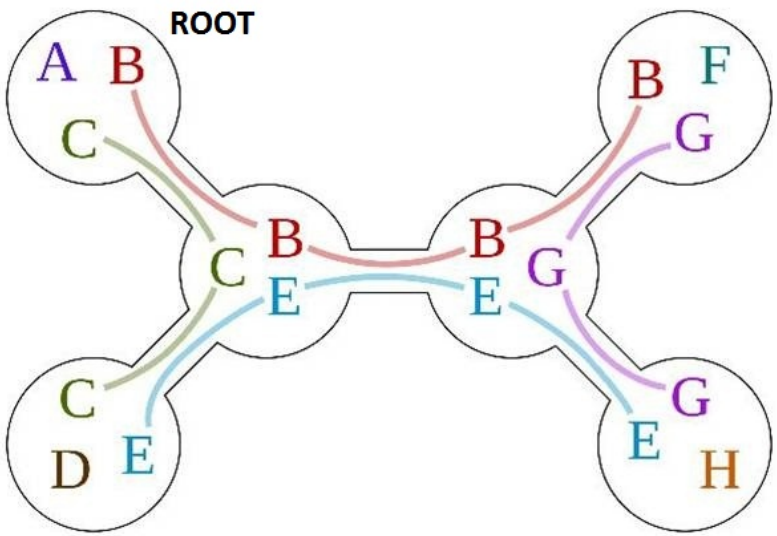
- We also developed some techniques for saving memory by limiting what is stored in the dynamic programming tables
- Rather than storing all independent sets in each node t's table, we only worry about the intersection of t's bag with the parent bag
- Rather than store all dynamic programming tables in memory, we delete them as we move up the tree
- Then we have some partial information about the optimal solution
- We use this information to *refine* the tree and decrease the width by removing vertices from consideration



Some Implementation Details

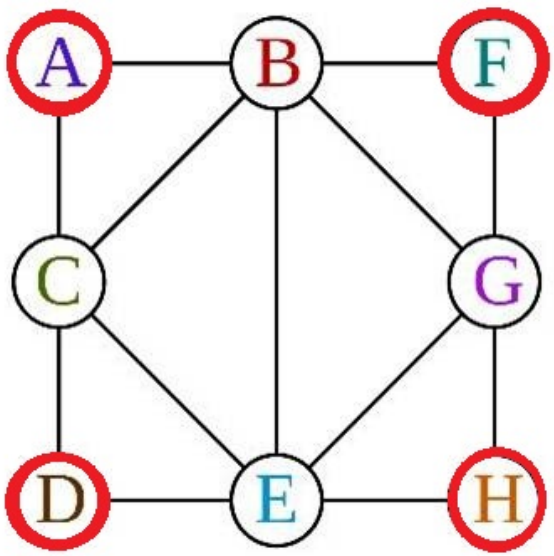
- We also developed some techniques for saving memory by limiting what is stored in the dynamic programming tables
- Rather than storing all independent sets in each node t's table, we only worry about the intersection of t's bag with the parent bag
- Rather than store all dynamic programming tables in memory, we delete them as we move up the tree
- Then we have some partial information about the optimal solution
- We use this information to *refine* the tree and decrease the width by removing vertices from consideration

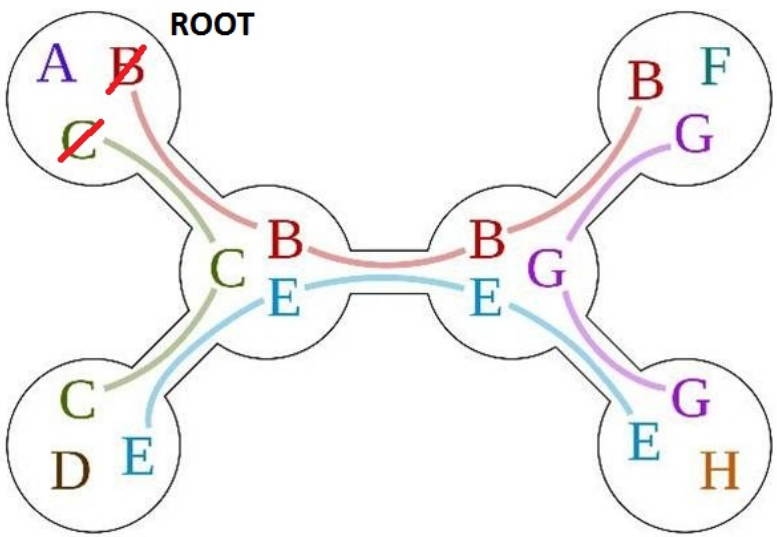




Some Implementation Details

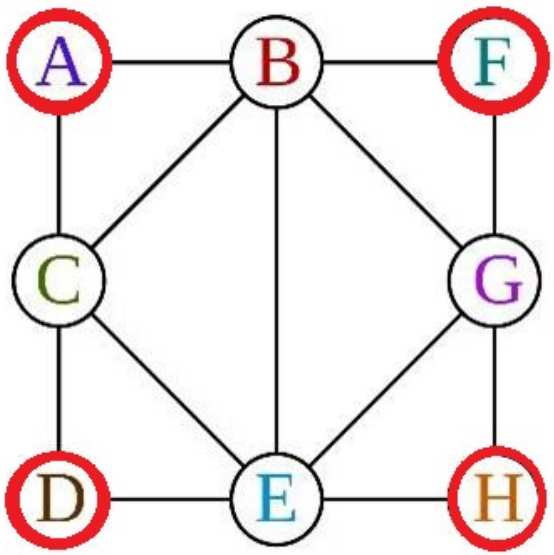
- We also developed some techniques for saving memory by limiting what is stored in the dynamic programming tables
- Rather than storing all independent sets in each node t's table, we only worry about the intersection of t's bag with the parent bag
- Rather than store all dynamic programming tables in memory, we delete them as we move up the tree
- Then we have some partial information about the optimal solution
- We use this information to *refine* the tree and decrease the width by removing vertices from consideration

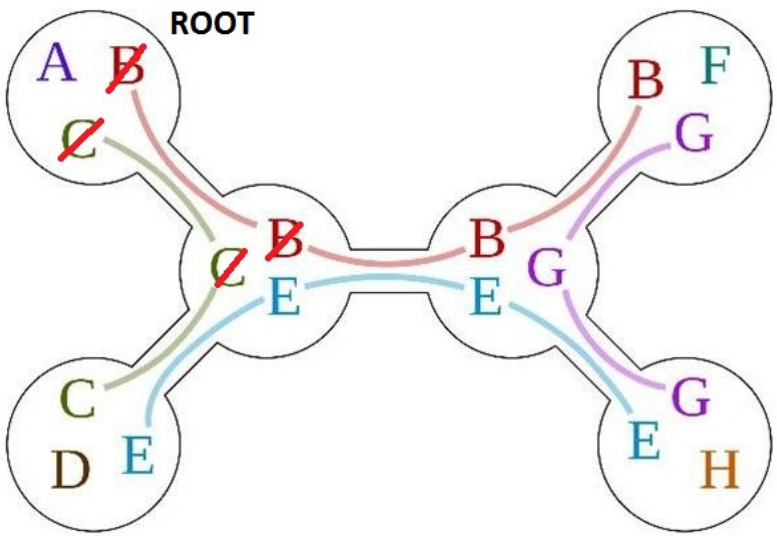




Some Implementation Details

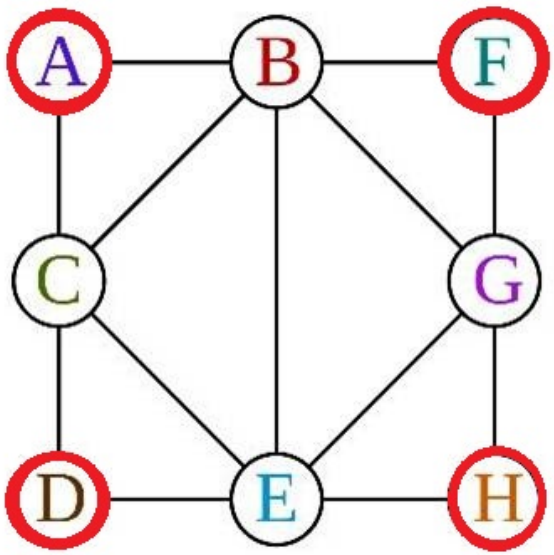
- We also developed some techniques for saving memory by limiting what is stored in the dynamic programming tables
- Rather than storing all independent sets in each node t's table, we only worry about the intersection of t's bag with the parent bag
- Rather than store all dynamic programming tables in memory, we delete them as we move up the tree
- Then we have some partial information about the optimal solution
- We use this information to *refine* the tree and decrease the width by removing vertices from consideration

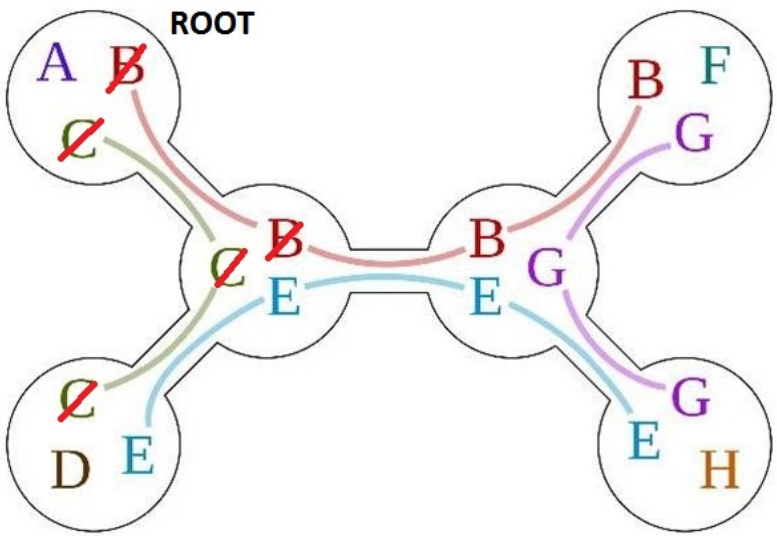




Some Implementation Details

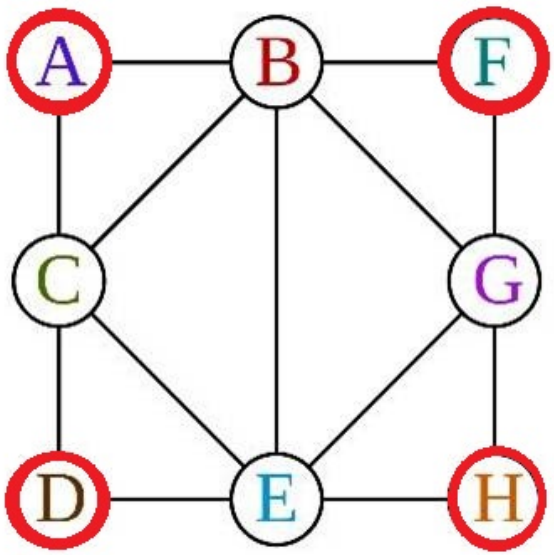
- We also developed some techniques for saving memory by limiting what is stored in the dynamic programming tables
- Rather than storing all independent sets in each node t's table, we only worry about the intersection of t's bag with the parent bag
- Rather than store all dynamic programming tables in memory, we delete them as we move up the tree
- Then we have some partial information about the optimal solution
- We use this information to *refine* the tree and decrease the width by removing vertices from consideration

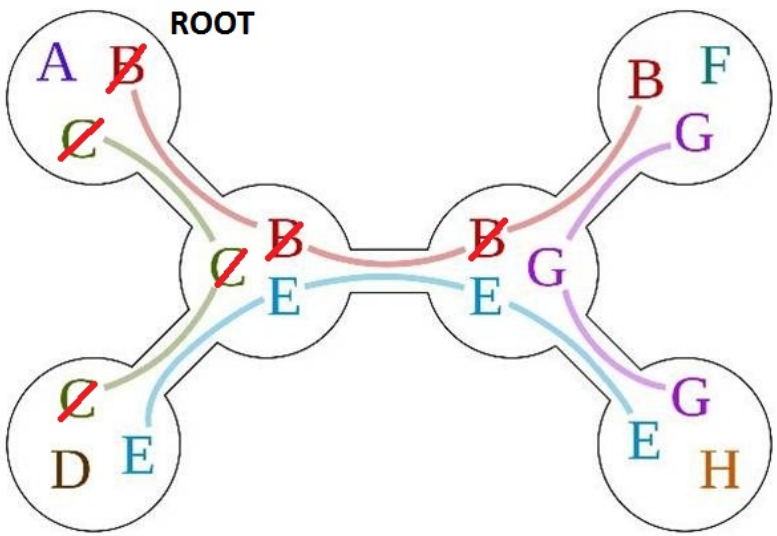




Some Implementation Details

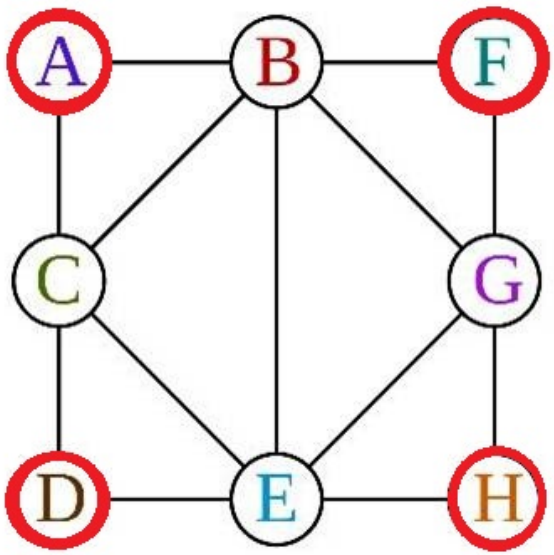
- We also developed some techniques for saving memory by limiting what is stored in the dynamic programming tables
- Rather than storing all independent sets in each node t's table, we only worry about the intersection of t's bag with the parent bag
- Rather than store all dynamic programming tables in memory, we delete them as we move up the tree
- Then we have some partial information about the optimal solution
- We use this information to *refine* the tree and decrease the width by removing vertices from consideration

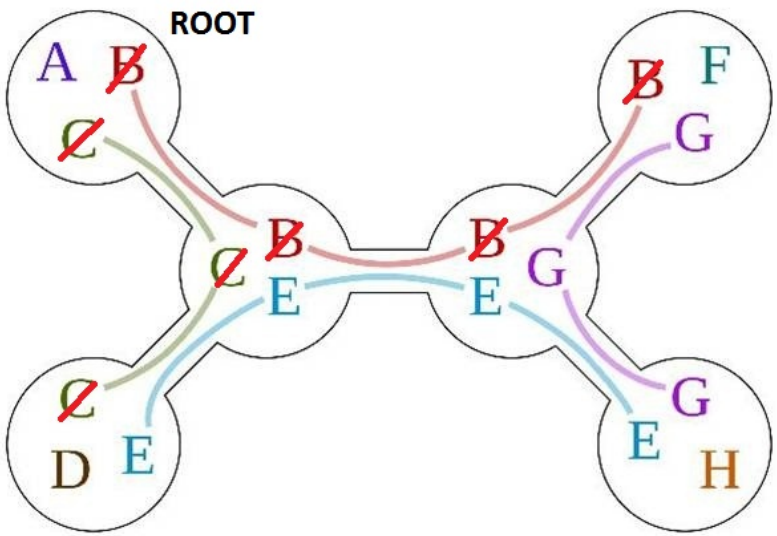




Some Implementation Details

- We also developed some techniques for saving memory by limiting what is stored in the dynamic programming tables
- Rather than storing all independent sets in each node t's table, we only worry about the intersection of t's bag with the parent bag
- Rather than store all dynamic programming tables in memory, we delete them as we move up the tree
- Then we have some partial information about the optimal solution
- We use this information to *refine* the tree and decrease the width by removing vertices from consideration



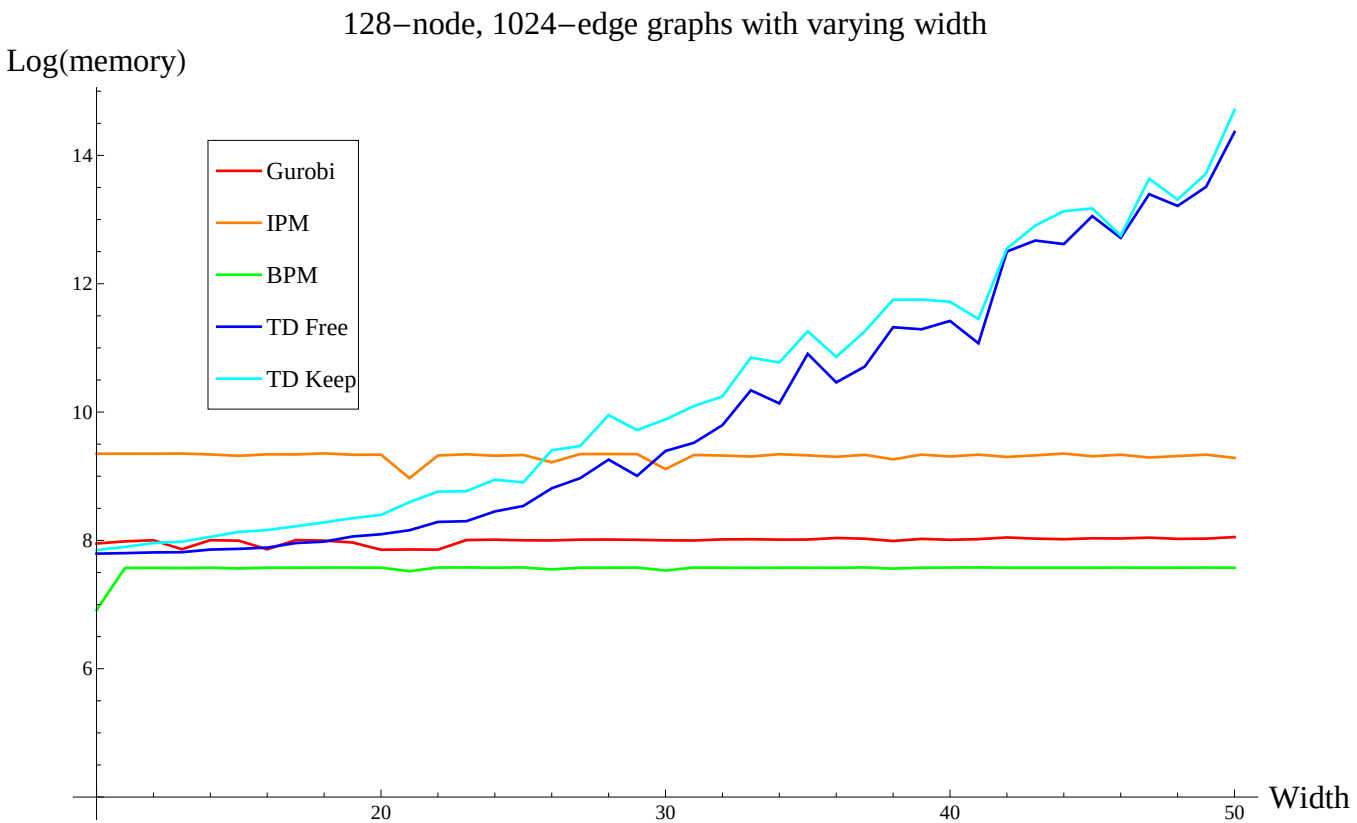


Comparison with other techniques

- One of the main goals of our project is to determine whether or not dynamic programming using tree decomposition is competitive with other state-of-the-art methods
  - *Gurobi*: Commercial mixed integer programming solver using standard integer programming formulation of MWIS (used Amazon EC2 AMI)
  - *IPM*: Uses an Interior Point Method along with semi-definite programming via branch-and-bound (Brian Borchers and Aaron Wilson, New Mexico State)
  - *BPM*: Uses a Boundary Point Method along with Semi-Definite Programming via branch-and-bound (Brian Borchers and Aaron Wilson, New Mexico State)
- In the next set of slides, we compare the performance and scalability of these approaches on a variety of graphs

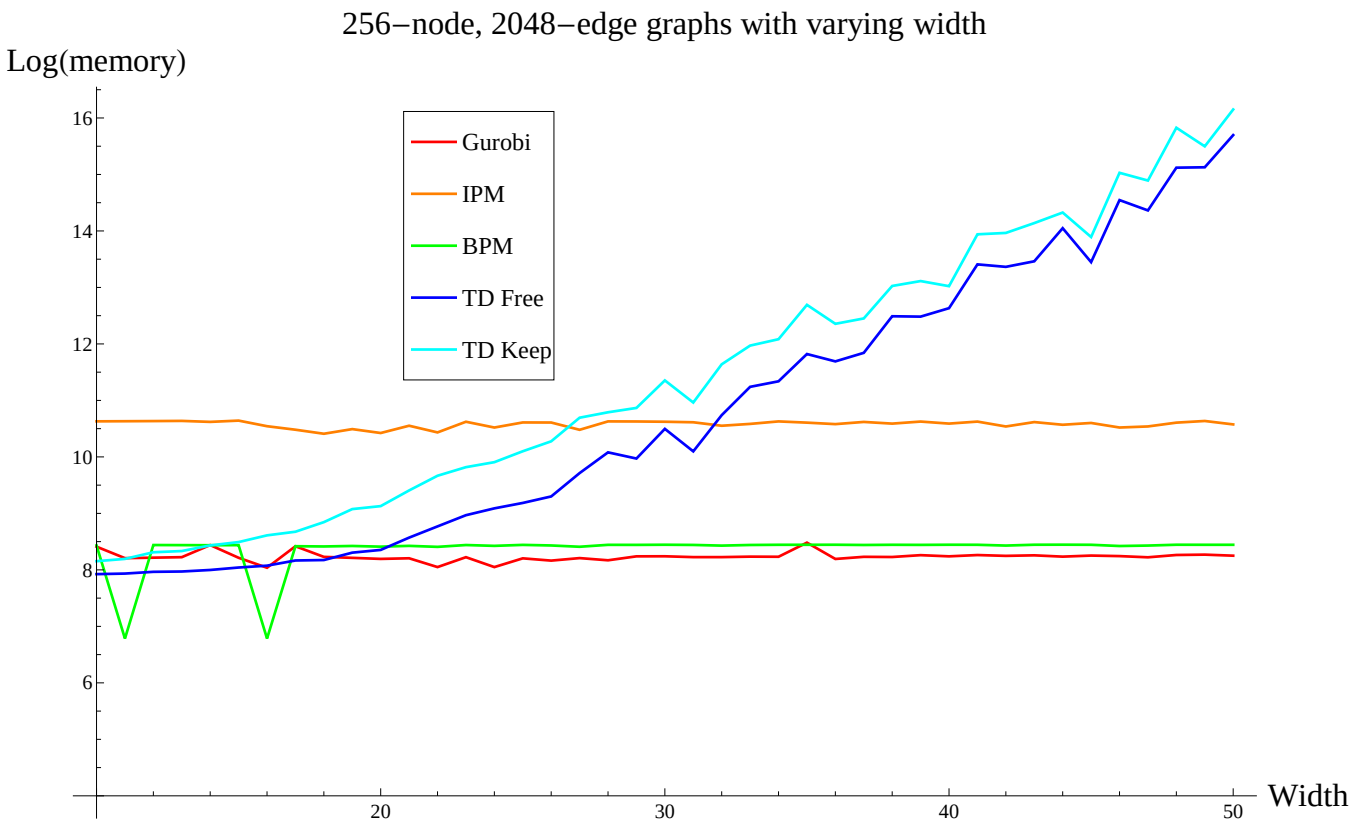
The Importance of Width

- We generated a set of random partial k-trees with the same number of nodes and roughly the same number of edges but varying k
- These graphs are useful for testing as we can control both n and m and maintain a lower bound on the treewidth
- Goal is to confirm belief that the width of a graph is not an important parameter for the performance of other methods



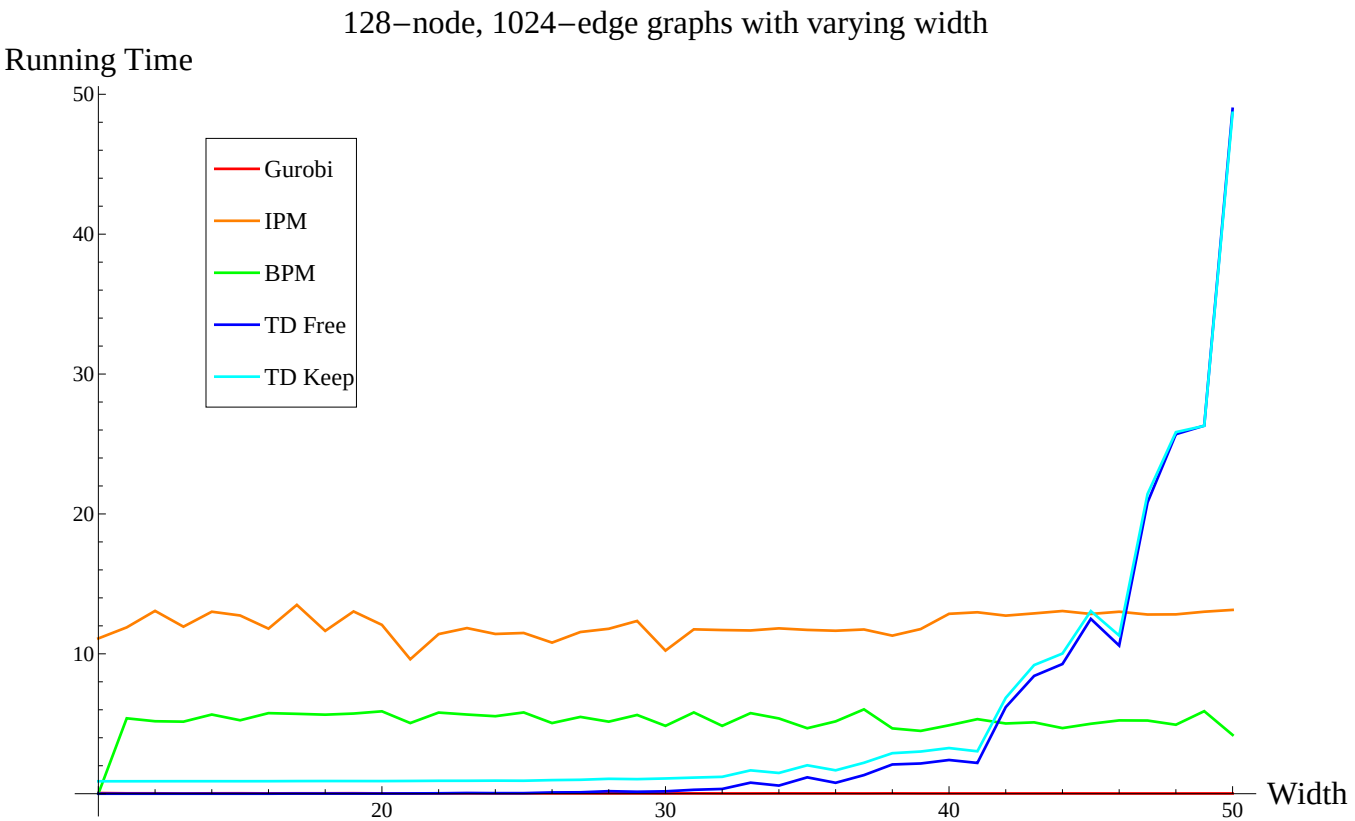
The Importance of Width

- We generated a set of random partial k-trees with the same number of nodes and roughly the same number of edges but varying k
- These graphs are useful for testing as we can control both the density and a lower bound on the treewidth while keeping n and m roughly constant
- Goal is to confirm belief that the width of a graph is not an important parameter for the performance of other methods



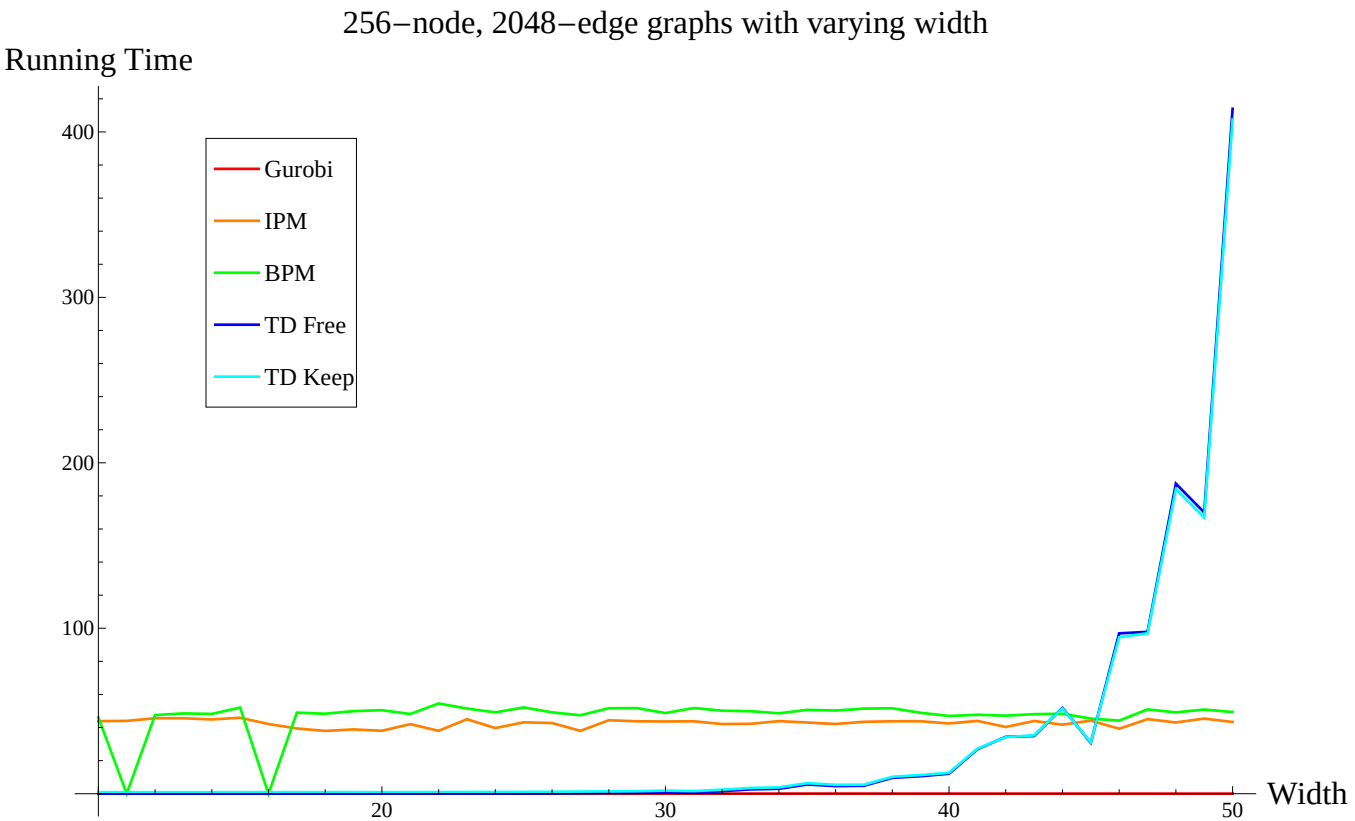
The Importance of Width

- We generated a set of random partial k-trees with the same number of nodes and roughly the same number of edges but varying k
- These graphs are useful for testing as we can control both the density and a lower bound on the treewidth while keeping n and m roughly constant
- Goal is to confirm belief that the width of a graph is not an important parameter for the performance of other methods



The Importance of Width

- We generated a set of random partial k-trees with the same number of nodes and roughly the same number of edges but varying k
- These graphs are useful for testing as we can control both the density and a lower bound on the treewidth while keeping n and m roughly constant
- Goal is to confirm belief that the width of a graph is not an important parameter for the performance of other methods



Performance Comparison

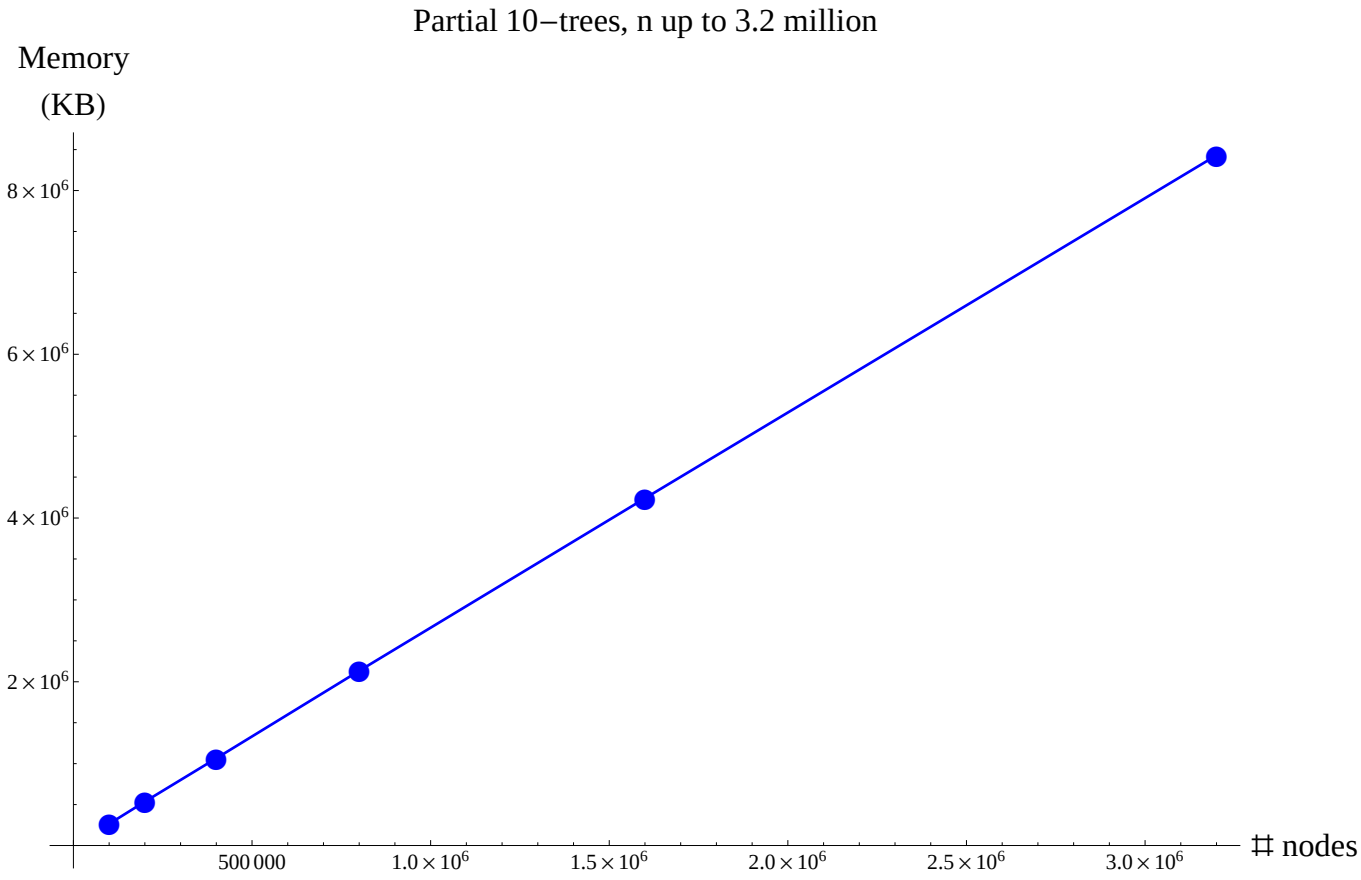
- Partial k-trees with $n \in \{1000, 2000, 4000, 8000\}$; $k \in \{15, 30, 60, 90, 120\}$; $p \in \{20, 40, 60, 80\}$
- Number of edges up to $750,000$

	# Completed	Max time	Max Memory
Gurobi	80	500 s.	1.2 GB
TD Free	62	2400 s.	17.5 GB
TD Keep	56	800 s.	24.2 GB
BPM	51	120 hours	0.9 GB
IPM	32	122 hours	16 GB

- Tree Decomposition approach fastest on 23/80 problems and is up to 5× faster on some denser graphs with low widths. This gap widens as n grows larger
- Tree Decomposition approach requires more memory and more time on sparse problems - reverse is true for other methods
- Freeing child tables offers memory savings of up to 8×
- Gurobi, BPM, and IPM can benefit from a good heuristic solution as it can accelerate the branch-and-bound process, but we have not devised a way to take advantage of good initial solution

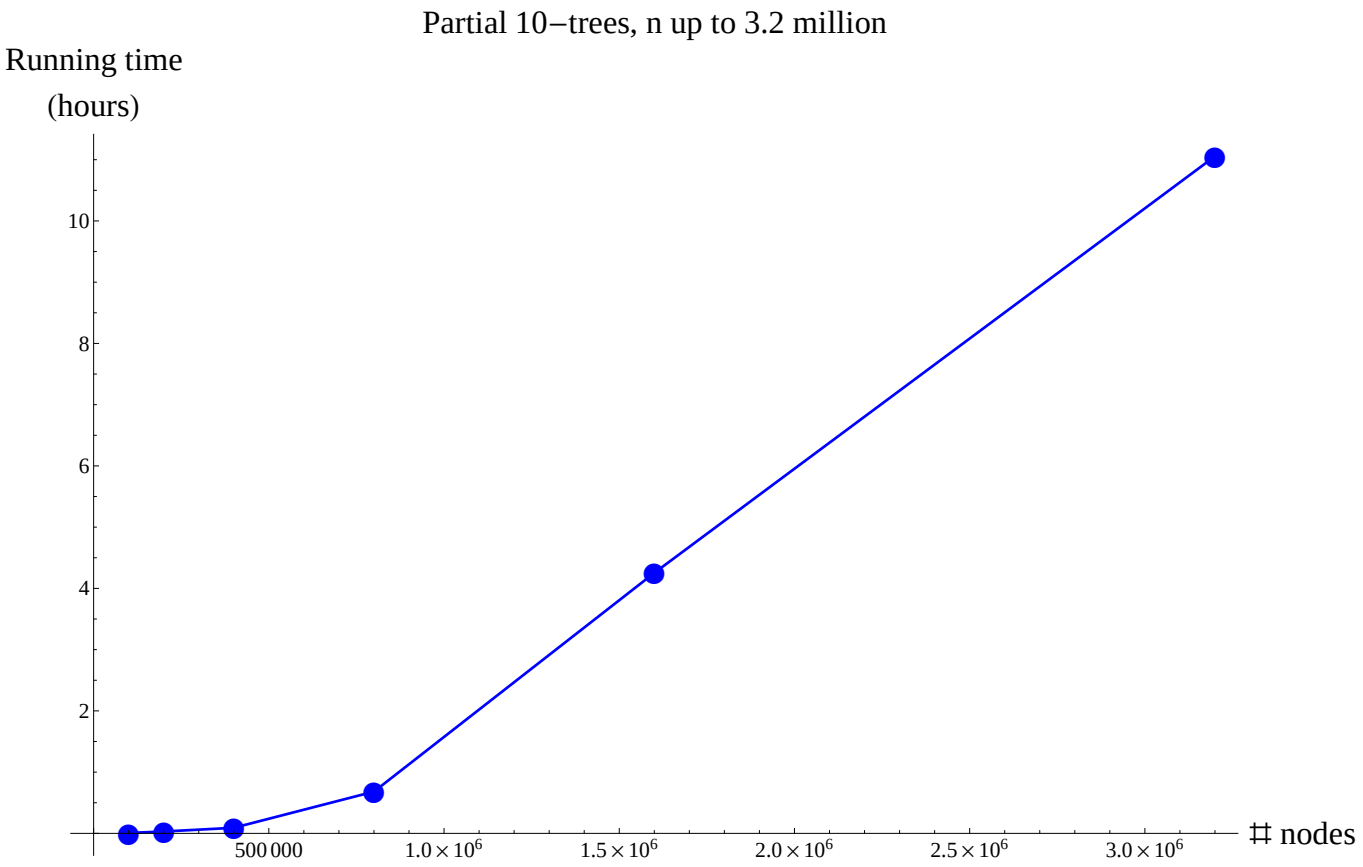
Dynamic Programming Scalability

- To test scalability for larger, low-width graphs, we generated partial k-trees with $n \in \{100K, 200K, 400K, 800K, 1.6M, 3.2M\}$, $k = 10$,



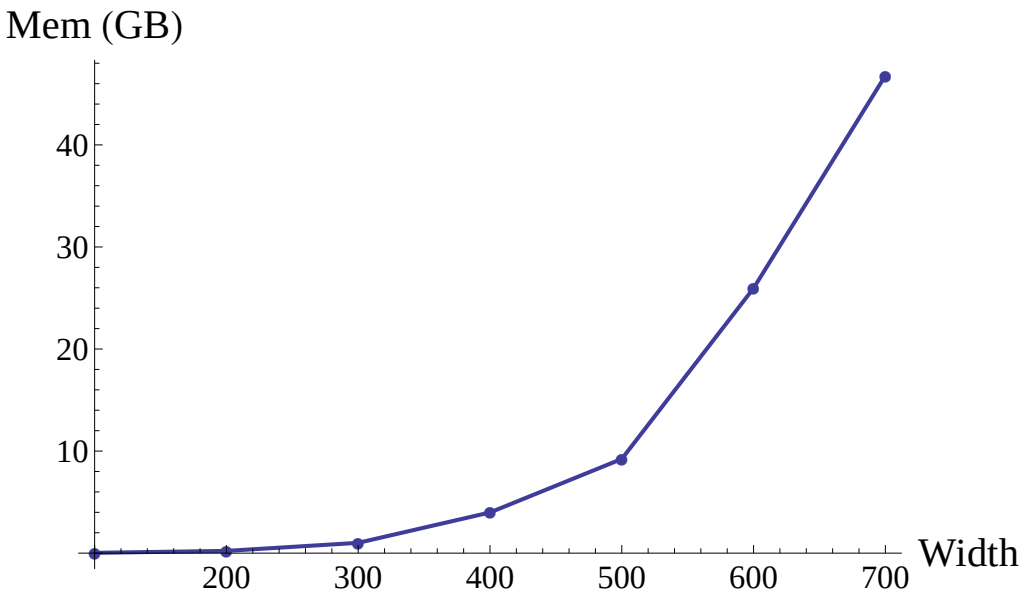
Dynamic Programming Scalability

- To test scalability for larger, low-width graphs, we generated partial k-trees with $n \in \{100K, 200K, 400K, 800K, 1.6M, 3.2M\}$, $k = 10$, $p = .7$



Is it Sufficient to Focus on Width Alone?

- Conventional wisdom has held that due to the $O(2^w)$ factor in the theoretical complexity, low width is essential for this kind of dynamic programming to be possible
- However, we have succeeded in generating optimal solutions to 10,000 node graphs with widths as high as 700 and roughly 6 million edges

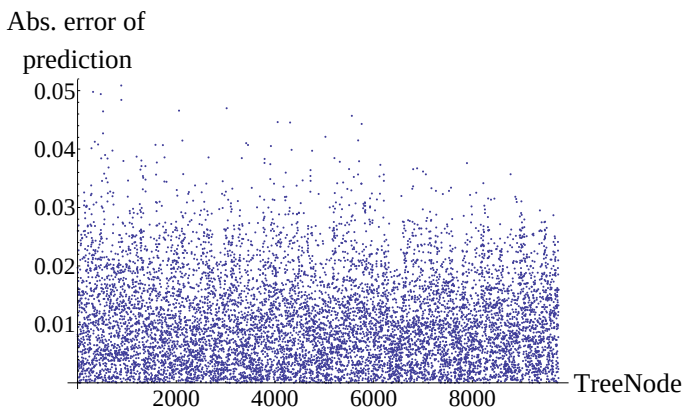


Is it Sufficient to Focus on Width Alone?

- For maximum independent set, the density of the graph is critical to both the running time and memory usage
- If we consider each bag of vertices as a random set of vertices from the graph, then we can estimate the number of independent sets at each node in the tree
- For a tree node whose bag contains w vertices and where the induced subgraph contains q edges, let $\rho = 2q/\binom{w}{2}$:

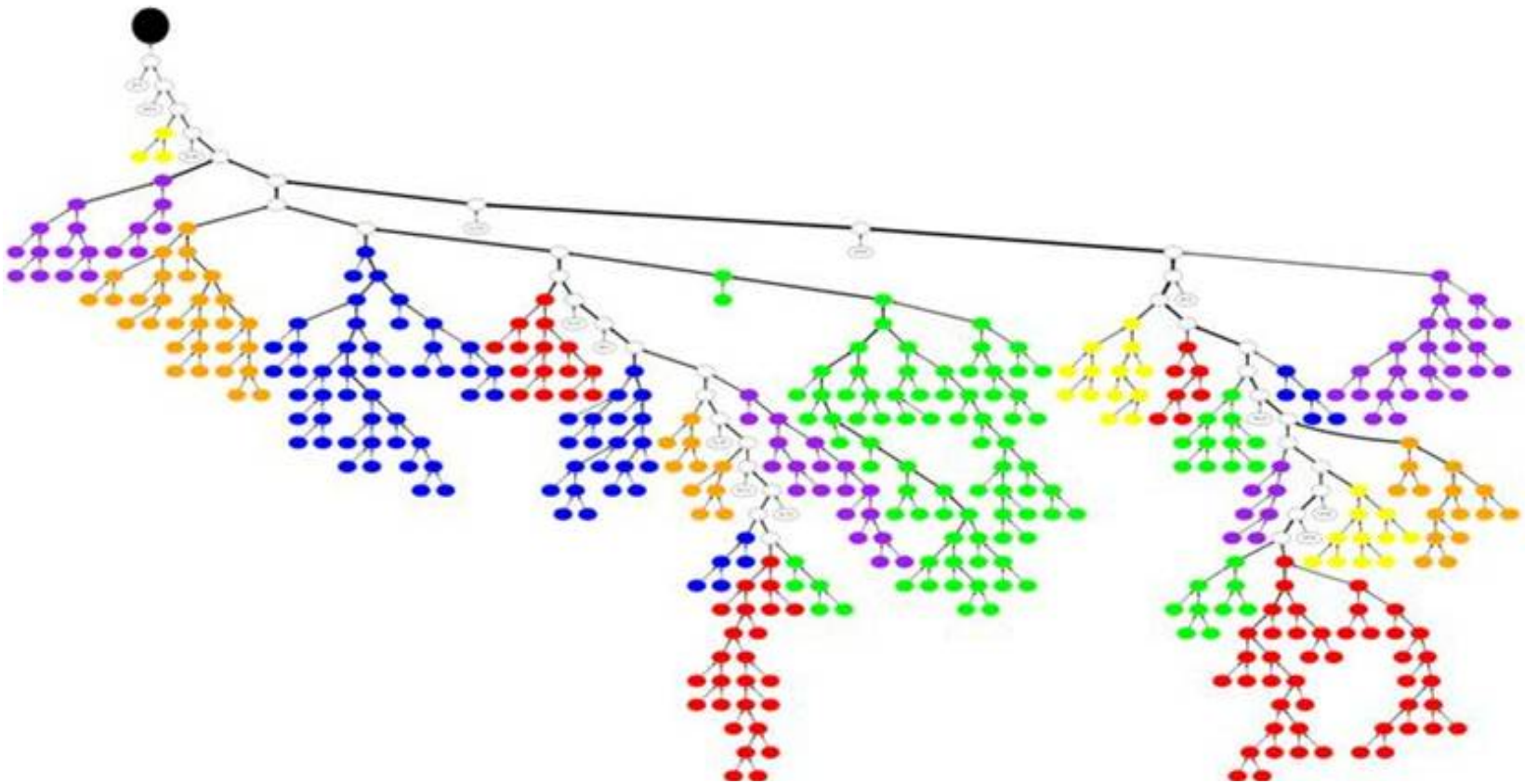
$$E[\#ind.sets] = \sum_{k=1}^{w} \binom{w}{k} (1-\rho)^{\binom{k}{2}}$$

- This estimate matches well with empirical data and provides a means to accurately estimate memory usage since the relevant quantities are known prior to running the dynamic programming



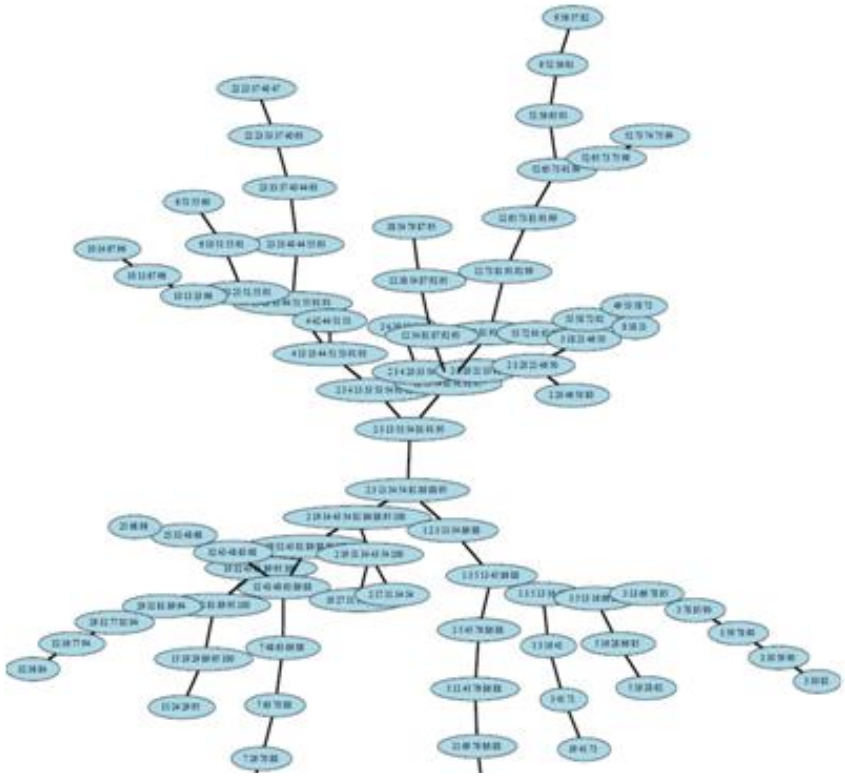
Opportunities for Parallelization

- Computation of the tables for the leaf nodes is embarrassingly parallel - often more than half the treenodes are leaves, especially for sparse graphs
- MPI implementation parallelizes work in computing solution table for single tree node, useful for large width graphs, but limited scalability
- Communication patterns are very irregular and unpredictable
- Exploring possibility of using MADNESS (computational chemistry code) runtime environment for parallelization and automated load-balancing



Conclusion

- Our project attempts to bridge the gap between theory and practice
- Large graphs with low width that are not too sparse seem to be the sweet spot for our dynamic programming algorithm as we outperform other methods
- Scaling behavior very different from other branch-and-bound based methods, and only computational operations are ANDs, ORs, shifts, memory reads and writes
- Memory consumption depends on both density and width. Conventional beliefs regarding width are not always valid
- Opportunities for parallelism exist, but communication is irregular



Acknowledgements

- Thanks to Brian Borchers and Troy Hanson for help with their software
- This work was supported by a grant from the Applied Mathematics Program of the US Department of Energy Office of Science.

U.S. DEPARTMENT OF ENERGY | Office of Science