Pushing the Power of Stochastic Greedy Ordering Schemes for Inference in Graphical Models

Kalev Kask, Andrew Gelfand, Lars Otten, Rina Dechter
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697, USA

Abstract

We study iterative randomized greedy algorithms for generating (elimination) orderings with small induced width and state space size - two parameters known to bound the complexity of inference in graphical models. We propose and implement the Iterative Greedy Variable Ordering (IGVO) algorithm, a new variant within this algorithm class. An empirical evaluation using different ranking functions and conditions of randomness, demonstrates that IGVO finds significantly better orderings than standard greedy ordering implementations when evaluated within an anytime framework. Additional order of magnitude improvements are demonstrated on a multi-core system, thus further expanding the set of solvable graphical models. The experiments also confirm the superiority of the MinFill heuristic within the iterative scheme.

Introduction

Having good (i.e., small tree-width) variable orderings in graphical models is known to be of central significance. While it was thought initially that this will impact only exact algorithms, many approximation schemes such as generalized belief propagation, mini-bucket elimination and AND/OR sampling (Yedidia, Freeman, and Weiss 2005; Dechter and Rish 2002; Gogate and Dechter 2008) are highly dependent on the availability of good orderings.

Finding a minimal tree-width ordering is known to be NP-complete (Arnborg, Corneil, and Proskurowski 1987). Therefore, the past two decades have seen serious effort devoted to developing anytime complete algorithms (Gogate and Dechter 2004) and approximation schemes, e.g. (Robertson and Seymour 1983; Bodlaender 2009; 2007). However, the most popular and practical class of approximations are greedy variable-ordering schemes that utilize various variable-ordering heuristics (Dechter and Meiri 1989; Kjaerulff 1992). Greedy schemes are popular because they are relatively fast, leaving ample time for the exponential inference computation that follows. Indeed, most recent research in probabilistic inference and optimization in graphical models has focused on advancing exact and approximate inference methods. This work has resulted in significant computational improvements (Darwiche 2001; Bacchus, Dalmo, and Piassi 2003; Dechter and Mateescu 2007; Kask, Dechter, and Gelfand 2010). However, since inference is exponential in the tree-width, a small reduction in tree-width (say by even by 1 or 2) can amount to one or two orders of magnitude reduction in inference time. Thus, huge computational gains are also possible by simply finding improved variable orderings.

An approach that has proven successful in practice is iteratively executing a greedy ordering algorithm with random tie-breaking and keeping track of the best ordering generated (Fishelson and Geiger 2003). This scheme is appealing because it is anytime, easy to employ and easily parallelized. While empirical evaluations of greedy ordering algorithms have been conducted (Clautiaux et al. 2004; Kjaerulff 1992; Larranaga et al. 1997; Bodlaender 2009), iterative schemes have not been systematically studied.

In this paper we study iterative randomized greedy schemes for finding variable orderings that can minimize both tree-width and state-space measures. We focus on the following ideas:

1. Implementing greedy variable-ordering schemes in a highly efficient manner.
2. Adding stochasticity to the selection process to encourage discovery of a more diverse set of variable orderings (in the spirit of stochastic local search).
3. Aborting iterations of the greedy algorithm that yield unpromising orderings (in the spirit of branch and bound).
4. Exploiting parallel processing.

Specifically, the paper describes the development of a highly efficient, parallel variable ordering algorithm - the Iterative Greedy Variable Ordering (IGVO) algorithm - that can accommodate any greedy ranking function. The IGVO algorithm also employs randomized "pooling" during variable selection (an idea from (Fishelson and Geiger 2003)) and early termination rules to stop unpromising iterations.

In an extensive empirical evaluation, we demonstrate an impressive (5-10x) speedup over a standard greedy variable ordering implementation. We also show that IGVO finds far better orderings than a standard implementation. These gains are attributed to a more efficient implementation, the pooling scheme and, to a lesser extent, early termination. In addition, we show that parallelization of the iterative scheme yields linear speedup using multi-core CPUs.
Background

A graphical model \( \mathcal{R} \) is a 4-tuple \( \mathcal{R} = (X, D, F, \otimes) \) that specifies variables, (finite) domains and functions. The arguments of a function is its scope. It represents the combination of all its functions: \( \otimes_{i=1}^{n} f_i \). For example, for Bayesian or Markov networks, the functions are probabilities and the combination operator is multiplication.

The primal graph of a graphical model associates a node with each variable and connects any two nodes appearing in the same scope. The set of neighbors of \( v \) in a graph \( G = (V, E) \) is denoted \( N_G(v) = \{ w \in V | \{ v, w \} \in E \} \) and the neighbors including \( v \) itself is denoted \( N_G[v] = N_G(v) \cup \{ v \} \). We assume that the reader is familiar with graph concepts, such as cycles, chords and cliques.

**Definition 1. Variable Ordering, Simplicial variable** - An ordering of a graph \( G = (V, E) \) is a bijection \( d : V \rightarrow \{1, 2, ..., n\} \), where \(|V| = n\). An ordered graph is a pair \((G, d)\), where \( d = V_1, ..., V_n \) is an ordering. A vertex \( v \in V \) is called simplicial, if the set of its higher ordered neighbors \( \{ w | \{ v, w \} \in E , \, d(w) > d(v) \} \) forms a clique. An elimination ordering \( d \) if perfect, if all \( v \in V \) are simplicial.

Note that we often denote the nodes \( V \) by the variables of the graphical model \( X \). A variable ordering \( d \) induces a sequence of supergraphs of \( G \), defined as follows, \( G_0 = G \) and for all \( 1 \leq i \leq n \), \( G_i \) is the graph obtained from \( G_{i-1} \) by adding edges so that all vertices in \( C_i = N_{G_{i-1}}(X_i) \cap \{ X_{i+1}, ..., X_n \} \) are pairwise connected. New edges added during the elimination process are called fill edges. \( F_i = E(G_i) \setminus E(G_{i-1}) \) is the set of fill edges added during step \( i \). Note that graph \( G_n \) is chordal and the set \( C_i \) is a clique. The process of obtaining graph \( G_i \) from \( G_{i-1} \) is referred to as eliminating vertex \( X_i \).

**Definition 2. Induced width, Treewidth** - Given an ordered graph \((G, d)\), the induced width of \( X_i \) is \( |C_i| \). The width of an ordering is \( w(d, \mathcal{R}) = \max_{i=1}^{n} |C_i| - 1 \). The induced width of a graph, \( w^* \), is the minimal induced width over all possible orderings. Treewidth is identical to induced-width and we will use those interchangeably.

**Definition 3. Total State Space** - The total state space of an elimination order \( d \) with respect to a graphical model \( \mathcal{R} \) is: \( s(d, \mathcal{R}) = \sum_{i=1}^{n} s(X_i, G_i) \), where \( s(X_i, G_i) = \prod_{u \in N_{G_i}(X_i)} |D(u)| \) is the space needed to eliminate node \( X_i \) from \( G_i \) and \( D(u) \) is the domain of node \( u \) (Kjaerulf 1992).

Greedy Variable Ordering (GVO) Algorithms

The general greedy variable ordering (GVO) scheme constructs an ordering as follows. A vertex \( v \) is greedily selected according to a heuristic ranking function. This vertex is placed into the first position of the ordering, its neighbors are connected and it is then eliminated. Then a second vertex is selected, placed into the next position of the ordering and eliminated. This process is repeated until all vertices have been eliminated. At each step, more than one vertex may be of minimum cost under the heuristic ranking function. In such cases, the vertex to be eliminated is selected randomly from the set of minimum cost vertices - i.e. random tie-breaking is used.

Vertex selection is guided by a heuristic ranking function. Using different ranking functions amounts to building elimination orders via different greedy algorithms. The following are three common ranking functions.

1. **MinFill Cost** - The number of fill edges added. Specifically, \( MF(v) = |F_v| \), where \( F_v \) is the set of fill edges that would be added if variable \( v \) were to be eliminated.

2. **MinDegree** (also known as MinInduced-Width) Cost - The degree of variable \( v \) in the current fill graph. Specifically, \( MD(v) = \text{deg}(v) \).

3. **MinComplexity** Cost - The complexity of variable elimination. Specifically, \( MC(v) = \prod_{u \in N_{G}(v)} |D(u)| \), where \( d(u) \) is the domain size of \( u \). This ranking function is designed to minimize the total state space.

The Iterative GVO (IGVO) Algorithm

In this section, we describe our iterative GVO (IGVO) algorithm. We begin by motivating the need for iterative randomized algorithms. We then give an algorithmic description of the basic randomized GVO algorithm, which is a subroutine of IGVO. Last, we describe the primary differences between IGVO and existing GVO schemes. This includes a description and complexity analysis of two implementation details that yield significant speedup.

Demonstrating Variability

Consider the histogram shown in Figure 1, which contains the empirical distribution of widths over 20000 runs of GVO using the MinFill ranking function on a particular Markov network. It is typical of the variability in both width and state-space size observed while running our greedy scheme on instances from linkage analysis, and shows the induced widths of the orderings found (against the left vertical axis) and the (log of the minimum and maximum state space size for each induced width found (right vertical axis). For most of the width values, there is a wide range of state

![Figure 1: Histogram of induced width and state space size observed over 20,000 iterations of MinFill with random tie-breaking on problem largeFam3-15-54 (3511 variables).](image-url)
space sizes; moreover, there is a significant overlap between state space sizes of neighboring widths. As expected we also see strong correlation between width and state space size - as width gets smaller, the state space size declines. This variability encourages the use of randomness in an iterative search for low width and low state space variable orderings and also demonstrates the need for early termination. The IGVO algorithm is described next.

**Algorithmic Description**

The basic randomized GVO algorithm is presented in Figure 2. The algorithm utilizes any ranking function $VC(X)$, such as the min-fill, min-degree or min-complexity heuristics described above. The algorithm has a total of $n$ iterations, where $n$ is the number of variables in the graphical model. At each iteration a variable is selected for elimination. Any simplicial variables in the graph are always eliminated (item 2a) because their elimination does not introduce any new fill edges and increase the width of an ordering (Robertson and Seymour 1983). If there are no simplicial variables (item 2b), a pool of the $p$ variables having the smallest variable cost (wrt. the given ranking function) are identified. A variable $X$ is randomly selected from this pool with probability proportional to its cost. The variable selected is added to the ordering (step 4) and the graph is updated by removing the variable and adding the necessary fill edges (step 3).

The GVO algorithm is a subroutine in the main iterative algorithm presented in Figure 3. The iterative algorithm uses an objective function $C(d, R)$ to compare different variable orderings. The objective function can be either the induced ordering (Robertson and Seymour 1983). If there are no simplicial variables (item 2b), a pool of the $p$ variables having the smallest variable cost (wrt. the given ranking function) are identified. A variable $X$ is randomly selected from this pool with probability proportional to its cost. The variable selected is added to the ordering (step 4) and the graph is updated by removing the variable and adding the necessary fill edges (step 3).

### Key Enhancements

The iterative GVO algorithm differs from standard GVO algorithms in the following ways:

1. **Randomization by pooling** - Randomization is introduced into the basic variable selection step. This allows our algorithm to select variables that are non-optimal according to the heuristic ranking function. This is similar to the idea used in (Fishelson and Geiger 2003), but their work is based on elimination cost only.

2. **Early termination** - Our algorithm improves upon naive iterative implementations by terminating inferior variable orderings.

### Algorithm Greedy Variable Order (GVO)

**Input:** Graphical model $R$, its primal graph $G=(V,E)$, a ranking function $VC(X)$, pool size $p$, exponent $e$, Objective Function $C(d, R)$ and Upper Bound $UB$

**Output:** A variable ordering $d = (X(1), ..., X(n))$.

- **Initialize:** Set $d = 0, W = V$.

- **For** $k = 1, ..., n$, do, using filled graph $G_{k-1}$:
  1. **If:** $C(d, R) > UB$, Terminate Early
  2. Select a variable $X$ to eliminate:
     (a) **If:** any simplicial variables in $W$, pick one as $X$,
     (b) **Else:** Order the variables from $W$ according to cost $VC$.
        - Let $P$ be a pool of the $p$ lowest cost variables. Select $X$ from $P$ with probability:
          $VC(X)^{r}/\sum_{Y \in P} VC(Y)^{e}$.
  3. Eliminate $X$ from $G_{k}$: connect neighbors of $X$, remove $X$.
  4. Set $W = W \setminus \{X\}$ and $d(k) = X$.

- **Return:** $d$

### Algorithm Iterative Greedy Variable Ordering (IGVO)

**Input:** Graphical model $R$, a cost function $VC(X)$, pool size $p$, exponent $e$, number of threads $m$, Objective Function $C(d, R)$ and Upper Bound $UB$, timeout/# iterations.

**Output:** A partial variable order $d_0 = (X(1), ..., X(m))$.

- **Initialize:** Let $d^*$ be the best ordering known at any point, and $UB = C(d^*, r)$ be its cost. Let $G$ be the primal graph of $R$.

- **On** $m$ threads execute :
  1. Compute $d = GVO(R, G, VC, p, c, UB)$.
  2. If $C(d, R) < UB$, set $d^* = d$ and set $UB = C(d, R)$.

- **Return:** $d^*$

3. **Optimizing the efficiency of GVO** - A significant reduction in run-time is obtained due to the following algorithmic improvements.

   - Adding fill edges to the graph has complexity of $O(deg^2)$ because the adjacency of all pairs of neighbors of $X$ must be checked. By keeping all adjacency lists sorted, we can reduce this to $O(2 \cdot deg^2)$, where $deg$ is the degree of the variable being eliminated.

   - When using the MinFill ranking function, the number of fill in edges (MinFill removal cost) must be updated every time a variable is eliminated. Rather than recomputing the MinFill cost of all vertices, this is typically done by recomputing the MinFill cost of only the neighbors of $X$ and the neighbors of $N_G(X)$. Updating costs in this fashion is wasteful, since there are only 3 cases where the MinFill cost actually changes. Assuming that $X$ is being eliminated, the 3 updates are:
     (a) For every variable $w$ and $u$, such that $(w, u) \in E$, if
(u, X) ∈ E, (w, X) /∈ E, subtract 1 from u.

(b) For every fill edge (u, v) added, for every w such that (w, u) ∈ E, (w, v) /∈ E, add 1 to u.

(c) For every fill edge (u, v) added, for every w such that (w, u) ∈ E and (w, v) ∈ E were not added, subtract 1 from w.

The first case can be handled as fill edges are added at no overhead. The second and third add complexity $O(nf \cdot 2 \cdot \deg)$ where $nf$ is the number of fill edges added.

4. Combined objectives - The algorithm can consider both tree-width and state space size as objectives in searching for variable orderings that minimize both the time and space of inference computations.

5. Parallelism - The algorithm is clearly massively parallel and can exploit the multi-core architecture of modern CPUs.

The above enhancements were incorporated into our implementation of the randomized GVO algorithm with MinFill ranking function. The complexity gains resulting from point 4 is summarized in the following theorem:

**Theorem 1.** The complexity of GVO is $O(nw^2 + n \cdot \log(n)) + NF \cdot w$, where $n$ is the number of variables, $w$ is the width, $NF$ is the number of fill edges. The complexity of standard MinFill is $O(n(n + w^3))$.

**Proof.** The complexity of a single iteration of GVO is $O(deg^2 + nf \cdot deg + n \cdot \log(n))$, where $n \cdot \log(n)$ is the cost of constructing the pool. Over $n$ iterations this is bounded by $O(n(w^2 + n \cdot \log(n)) + NF \cdot w)$. Without pooling, the complexity would be $O(n(w^2 + n) + NF \cdot w)$. In a standard MinFill algorithm, eliminating a variable $X$ is an $O(deg^2)$ operation, requiring enumeration of all pairs of neighbors adding edges where necessary. It requires updating MinFill ranking values for all neighbors and neighbors of neighbors of $X$, at a cost of $O(deg^3)$. Selecting a variable is $O(n)$, for a total single iteration cost of $O(n + deg^2 + deg^3)$. Over $n$ iterations this is bounded by $O(n(n + w^3))$.

**Experiments**

We conducted an extensive empirical evaluation of the suggested scheme and its parameters. The bulk of our experiments were performed on 242 largeFam problems modeling haplotype and linkage queries on biological pedigree data from the domain of computational genetics. The problems have between 2000 and 6000 variables with domain sizes from 2-6 and induced width ranging from the teens to over 100.

Comparing Greedy Ranking Functions

We first evaluated the impact of different ranking functions on the induced width and state space objectives.

Minimizing Induced Width. IGVO was run on each instance with each ranking function for 1 hour and the best width recorded. Figure 4 shows the results in a cumulative manner: For a given width ($x$-axis) the curves depict the number of instances (out of 242) having width lower or equal to the one specified in the $x$-axis, for each ranking function. Clearly MinFill outperforms the other ranking functions and returns lower-width orderings across the problem set.

For lack of space we don’t include the full set of results, but we can state that on 186 problems MinFill is strictly better than both other schemes. On average, the best orderings found by MinFill have width 3 lower than MinDegree and 7 lower than MinComplexity (corresponding to up to 3 and 7 orders of magnitude better algorithmic performance). Furthermore, on average MinFill finds the best ordering after 836 seconds, 88 seconds before MinDegree and 717 seconds before MinComplexity.

Minimizing State Space Size. Results of using the three ranking functions with the state space objective are provided in Table 1. In the top half, problems have their original domain sizes (2-6), while in the bottom half we adjusted the domains to be either 2 (2/3 of the time) or 30 (1/3 of the time). We thus see that MinFill performs best by far on the original configuration, finding the ordering with the smallest state space size in the vast majority of cases (185 out of 242). However, once the domains are very uneven (bottom

<table>
<thead>
<tr>
<th>MinFill</th>
<th>MinDeg</th>
<th>MinCompl</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain sizes 2-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td># wins</td>
<td>log of min SS</td>
<td>w of min SS</td>
</tr>
<tr>
<td>185</td>
<td>23.6</td>
<td>59</td>
</tr>
<tr>
<td>3</td>
<td>26.4</td>
<td>62</td>
</tr>
<tr>
<td>25.4</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>domain sizes 2-30</td>
<td></td>
<td></td>
</tr>
<tr>
<td># wins</td>
<td>log of min SS</td>
<td>w of min SS</td>
</tr>
<tr>
<td>108</td>
<td>37.6</td>
<td>61</td>
</tr>
<tr>
<td>7</td>
<td>40.5</td>
<td>64</td>
</tr>
<tr>
<td>39.1</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Results when using state space size (SS) as objective. All values except number of wins are averaged across 242 largeFam problems (100,000 iterations per run).
half of the table), MinComplexity ranking function outperforms MinFill in finding smallest state space size in more cases (127 vs 108), while MinFill has smaller average state space size.

The Effect of Randomization
Strictly greedy algorithms can get stuck in local minima, even with random tie-breaking. This is particularly true when few ties occur while running GVO on a problem instance. To escape such minima, IGVO is augmented with the option of making a greedily suboptimal choice, where we don’t pick a variable with lowest cost, but instead choose among a pool (size $p$) of lowest-cost variables. In particular, the probability of picking variable $X$ is $VC(X)^e/\sum_{Y \in P} VC(Y)^e$, where $e$ is a weighing constant and $P$ is the pool. In this form, $e = 0$ yields a uniform distribution, $e < 0$ gives preference to variables with lower cost, and $e > 0$ leans towards variables with higher cost.

Figure 5 contrasts a standard MinFill implementation against IGVO with exponent $e = -1$ and varying pool sizes, focusing on a small part of the cumulative plot for readability. We observe that performance improves as the pool size increases from 1 to 4 to 8, but not much after that (if at all).

Varying the value of $e$ does not have a major impact on the overall solution, but we found that $e = -1$ discovered the best width sooner than $e = 0$ or $e = 1$. For instance, with $p = 8$, the minimum width ordering was found after an average of 478 seconds with $e = -1$, compared to 888 seconds and 946 seconds for $e = 0$ and $e = -1$, respectively.

Comparison with Standard MinFill
We furthermore conducted a comparison of IGVO against a standard MinFill implementation\(^1\). In addition to the 242

\(^1\)An earlier version of our standard MinFill was used in the UAI-2010 competition with solver(s) that won first place in 4 categories, and was subsequently improved/enhanced, yielding the standard MinFill used in this paper. It eliminates all simplical vertices as described in Figure 2 and uses random tie-breaking without

Figure 5: Magnified cumulative plot comparing various pool sizes ($e = -1$, timeout 30 minutes, $largeFam$ problems).

Figure 6: Magnified cumulative plot comparing standard MinFill, single-threaded IGVO, and 12-threaded IGVO on type4 problems. Timeout 1 hour.

<table>
<thead>
<tr>
<th>instance</th>
<th>$n$</th>
<th>$iter$</th>
<th>$w$</th>
<th>$iter$</th>
<th>$w$</th>
<th>$w$</th>
<th>spd</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-18</td>
<td>7,435</td>
<td>6,430</td>
<td>51</td>
<td>26,689</td>
<td>48</td>
<td>324,664</td>
<td>48</td>
</tr>
<tr>
<td>110-19</td>
<td>7,303</td>
<td>3,852</td>
<td>54</td>
<td>13,005</td>
<td>52</td>
<td>158,806</td>
<td>51</td>
</tr>
<tr>
<td>120-18</td>
<td>8,656</td>
<td>6,594</td>
<td>47</td>
<td>17,604</td>
<td>45</td>
<td>211,830</td>
<td>44</td>
</tr>
<tr>
<td>120-25</td>
<td>9,171</td>
<td>3,789</td>
<td>57</td>
<td>14,576</td>
<td>56</td>
<td>176,156</td>
<td>54</td>
</tr>
<tr>
<td>130-20</td>
<td>9,328</td>
<td>3,167</td>
<td>60</td>
<td>12,541</td>
<td>58</td>
<td>154,647</td>
<td>57</td>
</tr>
<tr>
<td>130-22</td>
<td>10,271</td>
<td>3,747</td>
<td>56</td>
<td>13,107</td>
<td>52</td>
<td>168,635</td>
<td>52</td>
</tr>
<tr>
<td>140-23</td>
<td>10,998</td>
<td>2,318</td>
<td>61</td>
<td>7,654</td>
<td>60</td>
<td>91,576</td>
<td>57</td>
</tr>
<tr>
<td>150-22</td>
<td>11,799</td>
<td>2,636</td>
<td>57</td>
<td>8,423</td>
<td>54</td>
<td>99,949</td>
<td>53</td>
</tr>
<tr>
<td>170-18</td>
<td>12,186</td>
<td>2,202</td>
<td>59</td>
<td>6,913</td>
<td>55</td>
<td>82,756</td>
<td>55</td>
</tr>
<tr>
<td>170-22</td>
<td>14,641</td>
<td>2,795</td>
<td>58</td>
<td>8,147</td>
<td>56</td>
<td>97,423</td>
<td>54</td>
</tr>
<tr>
<td>190-19</td>
<td>15,433</td>
<td>3,044</td>
<td>56</td>
<td>6,473</td>
<td>54</td>
<td>77,287</td>
<td>52</td>
</tr>
<tr>
<td>190-21</td>
<td>15,125</td>
<td>5,284</td>
<td>43</td>
<td>9,545</td>
<td>42</td>
<td>115,048</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 2: Exemplary results comparing standard MinFill with single- and 12-threaded IGVO ($p = 8$, $e = -1$) on type4 problems. $n$ denotes the number of problem variables, $iter$ is the number of ordering iterations performed within 1 hour, $w$ the best width found, spd the parallel speedup of IGVO(12).

Figure 7: Time savings relative to standard MinFill across a subset of 100 $largeFam$ instances (20,000 iterations each).
Table 3: Four type4 problems that were previously infeasible because of their space requirements, but are now solvable (n is number of problem variables, k max. domain size).

<table>
<thead>
<tr>
<th>instance</th>
<th>n</th>
<th>k</th>
<th>w</th>
<th>space</th>
<th>w</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>110-21</td>
<td>7,675</td>
<td>5</td>
<td>37</td>
<td>16 TB</td>
<td>33</td>
<td>215 GB</td>
</tr>
<tr>
<td>140-20</td>
<td>9,355</td>
<td>5</td>
<td>35</td>
<td>10 TB</td>
<td>28</td>
<td>4 GB</td>
</tr>
<tr>
<td>180-21</td>
<td>14,157</td>
<td>5</td>
<td>38</td>
<td>9 TB</td>
<td>31</td>
<td>67 GB</td>
</tr>
<tr>
<td>200-18</td>
<td>15,319</td>
<td>5</td>
<td>36</td>
<td>19 TB</td>
<td>30</td>
<td>41 GB</td>
</tr>
</tbody>
</table>

Figure 6 summarizes the results in the same cumulative manner as before; it includes standard MinFill as well as single- and 12-threaded IGVO (p = 8, e = −1). Table 2 also lists detailed results for a number of instances. Focusing first on single-threaded execution, we observe the following:

- IGVO benefits from the strategy of pooling non-optimal choices and consistently finds orderings with lower width than standard MinFill, evidenced by the difference between the two respective curves in Figure 6 and the examples in Table 2.

- Efficient data structures and implementation allow IGVO to perform many more iterations – often more than 3 times as many – in the same time as the standard implementation (cf. Table 2), in spite of the additional overhead from pooling. Figure 7 provides further evidence of this speedup, with more than 80% time savings across a subset of 100 problem instances.

Parallelization. Table 2 and Figure 6 also include the results of running IGVO with 12 parallel threads (on dual 6-core CPUs, i.e. 12 cores). Apart from the fact that it returns further improved orderings, we see that the parallel algorithm completes roughly 12 times as many iterations as the single-threaded one, confirming fairly linear scaling as expected. (Speedups greater than 12 can be explained by the increasing impact of early termination as the minimum width improves with the number of iterations.)

Early Termination. Figure 7 also shows the relative impact on time of early termination. We see that it gives a significant speedup (between 20 and 40%) when applied to standard MinFill. In case of IGVO, however, which already incorporates a host of other optimizations and extensions, it only provides a small additional benefit – about 5% on average.

Table 4: Exemplary results comparing standard MinFill with single- and 12-threaded IGVO (p = 8, e = −1) on protein folding problems. Timeout 1 hour.

<table>
<thead>
<tr>
<th>instance</th>
<th>n</th>
<th>iter</th>
<th>w</th>
<th>iter</th>
<th>w</th>
<th>spd</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdb1e18</td>
<td>618</td>
<td>1.72E+5</td>
<td>37</td>
<td>8.61E+5</td>
<td>32</td>
<td>1.02E+7</td>
</tr>
<tr>
<td>pdb1gmt</td>
<td>443</td>
<td>3.28E+5</td>
<td>33</td>
<td>1.53E+6</td>
<td>26</td>
<td>1.82E+7</td>
</tr>
<tr>
<td>pdb1i24</td>
<td>337</td>
<td>4.46E+5</td>
<td>33</td>
<td>2.20E+6</td>
<td>27</td>
<td>2.62E+7</td>
</tr>
<tr>
<td>pdb1m6i</td>
<td>375</td>
<td>4.88E+5</td>
<td>32</td>
<td>2.06E+6</td>
<td>25</td>
<td>2.45E+7</td>
</tr>
<tr>
<td>pdb1nqe</td>
<td>457</td>
<td>3.82E+5</td>
<td>34</td>
<td>1.56E+6</td>
<td>31</td>
<td>1.81E+7</td>
</tr>
<tr>
<td>pdb1qpk</td>
<td>332</td>
<td>4.13E+5</td>
<td>32</td>
<td>2.38E+6</td>
<td>26</td>
<td>2.87E+7</td>
</tr>
<tr>
<td>pdb1c3r</td>
<td>636</td>
<td>1.48E+5</td>
<td>40</td>
<td>7.01E+5</td>
<td>32</td>
<td>8.33E+6</td>
</tr>
<tr>
<td>pdb1e3d</td>
<td>1298</td>
<td>4.09E+5</td>
<td>59</td>
<td>1.83E+5</td>
<td>53</td>
<td>2.30E+6</td>
</tr>
<tr>
<td>pdb1eg5</td>
<td>609</td>
<td>1.63E+5</td>
<td>37</td>
<td>7.46E+5</td>
<td>32</td>
<td>8.91E+6</td>
</tr>
<tr>
<td>pdb1fnn</td>
<td>658</td>
<td>1.82E+5</td>
<td>27</td>
<td>7.57E+5</td>
<td>24</td>
<td>8.99E+6</td>
</tr>
<tr>
<td>pdb1gnl</td>
<td>866</td>
<td>1.38E+5</td>
<td>34</td>
<td>4.94E+5</td>
<td>30</td>
<td>5.95E+6</td>
</tr>
<tr>
<td>pdb1h80</td>
<td>744</td>
<td>1.92E+5</td>
<td>33</td>
<td>6.69E+5</td>
<td>30</td>
<td>7.92E+6</td>
</tr>
<tr>
<td>pdb1i2m</td>
<td>919</td>
<td>1.07E+5</td>
<td>39</td>
<td>4.35E+5</td>
<td>34</td>
<td>5.09E+6</td>
</tr>
<tr>
<td>pdb1l7n</td>
<td>530</td>
<td>2.79E+5</td>
<td>29</td>
<td>1.17E+6</td>
<td>26</td>
<td>1.41E+7</td>
</tr>
<tr>
<td>pdb1jet</td>
<td>457</td>
<td>2.40E+5</td>
<td>36</td>
<td>1.25E+6</td>
<td>31</td>
<td>1.46E+7</td>
</tr>
</tbody>
</table>

Table 3 lists four problem instances whose massive memory requirements of many terabytes made solving them previously impossible, even for powerful schemes utilizing external memory (Kask, Dechter, and Gelfand 2010). Running IGVO(12) for one hour, however, yielded good orderings that allow for solving these instances with mere gigabytes of hard disk space, a significant improvement.

**Pushing the Boundaries of Feasibility**

Table 4: Exemplary results comparing standard MinFill with single- and 12-threaded IGVO (p = 8, e = −1) on protein folding problems (max. domain size k = 81). n denotes the number of problem variables, iter is the number of ordering iterations performed within 1 hour, w the best width found, spd the parallel speedup of IGVO(12).

**Protein Folding Problems**

We conducted another set of experiments on 138 protein folding / side-chain prediction problems modeled as Markov networks (Yanover and Weiss 2002) – here the max. do-
main size is $k = 81$, which makes finding a low-width ordering particularly important. Figure 8 shows a cumulative summary plot while Table 4 presents select results in detail. Again we see that IGVO is able to perform many more iterations than the standard implementation and consistently finds orderings with lower width, often by a considerable margin. However, the problems are small enough (mostly less than 1,000 variables) that the move to parallel IGVO does not significantly improve the width of the returned orderings.

Conclusions

The paper studies the iterative application of stochastic greedy ordering schemes showing that it may currently be the primary (and perhaps the only) practical scheme for finding low-treewidth decompositions. Such schemes are appealing because of their simplicity, flexibility, anytime nature, and trivial parallelization. Within this class of schemes, we presented the IGVO algorithm and demonstrated its superiority to current variants in terms of both run-time and finding improved orderings. We showed empirically, on hard instances from computational biology, that IGVO’s superiority can be attributed to: 1) Implementation efficiency - by utilizing new data-structures and good software engineering, IGVO can execute many more iterations in a given amount of time. This is particularly impressive since the baseline (standard) implementation is by itself highly efficient. 2) Randomization via pooling - the parametrized scheme for introducing stochasticity to the selection process allows IGVO to explore more diverse orderings than standard randomized GVO. For certain parameter settings this yields better results, despite the pooling overhead. 3) Early termination - this was shown to have a marginal effect on IGVO but much greater impact when the basic greedy scheme is less efficient.

We demonstrated additional speedup and found improved orderings when running IGVO on a multi-core machine. This additional gain allows us to perform exact computations on quite a few problems that could not be solved even when using external memory (i.e., solved by BEEM (Kask, Dechter, and Gelfand 2010)).

The IGVO algorithm can accommodate any greedy ranking function. Using IGVO we verified that MinFill is indeed a superior greedy heuristic when seeking low-treewidth decompositions. For problems with non-uniform domains, we observed that the MinComplexity ranking function is often superior to the MinFill heuristic. We also compared (but do not report here) the performance of IGVO to exact anytime algorithms (e.g., (Gogate and Dechter 2004)) and stochastic local search schemes and found IGVO to superior to these schemes as well.

The contribution of this paper is really two-fold. First, it brings many known ideas into a single, anytime variable ordering framework. Second, it demonstrates benefit of iterative randomized greedy schemes and underscores the importance of certain design choices when using such schemes.

Acknowledgments

This work was partially supported by NSF grants IIS-0713118, IIS-1065618 and NIH grant 5R01HG004175-03.

References


