Scribe Notes: 2/6/2013
Presenter: Dr. Berthe Choueiry
Scribe: Tony Schneider
Reading: Chapter 9 of Dechter's Textbook

1. Student Questions

Daniel Dobos:
Question: Why don't we apply tree decompositions to binary networks?
Answer: Tree Clustering (TC) is not limited to binary CSPs. However, in TC to have backtrack-free search, you need to generate a global constraint over common variables between clusters (i.e., the variables of the seperators). The generated constraint is likely to be non-binary, so if a graph is initially binary, it might not be after applying TC. Note that you don't *need* to add a non-binary constraint, but if you don't, you may be forced to backtrack by the size of the largest component. In particular, Jegou & Terrioux (2003),[1] who exploit a tree decomposition during backtrack search, run all their experiments on binary CSPs. (Note: this paper would be a good presentation.)

Nate Stender:
Question: Regarding ATC time complexity (theorem 9.11, page 267 from Dechter): If it's possible to get rid of the dependency on the degree in the time complexity with no increase in space complexity, why not use that modification in the first place?
Answer: The CTE, which has higher complexity, the same computation is repeated for each neighbor because there is no directionality in the edges. ATC proposes to order the buckets and send the messages in an orderly way, once bottom-up and once top-down, thus saving repeating the computation (which is not stored) for all neighbors (-1). Please check Piazza for more discussion.

2. Callback to TCSPs

We briefly went over one slide from Angelo Montanari that listed some applications and research areas concerned with temporal reasoning.[2]

3. Dechter's Slides[3] for Chapter 4

Note: Several of these concepts were discussed previously, most commonly through student questions. Refer to previous scribe notes for more information.

[1] Jegou & Terrioux (2003) paper describing BTD
[2] Slides from Montanari
[3] Dechter's Chapter 4 Slides

Slide 2 – Backtrack-Free Search

It is important to note that backtrack-free search here requires that it is relative to a specific ordering of variables, and back checking is still necessary because domains are not minimal (not all of the values in a given variable are consistent with previous instantiations). You are only guaranteed that at least one value in the current variable extends the current partial solution to a complete solution.

Slide 6 – Algorithm for directional arc-consistency (DAC)

Note: skipped slides 3, 4, and 5 as they had already been covered in previous lectures.

One important point here is to not confuse the *instantiation* ordering (top down, direction of the search tree) with the *elimination* ordering (bottom up). The textbook is rather confusing and inverts the two orderings in many places.

Main idea: Given some ordering of variables, process each variable in a reverse ordering (e.g., if your instantiation ordering is x_1 to x_n, begin by processing x_n). The process consists of revising the domains of current variable's parents with respect to constraint between the parent and the current variable. Once all of the current variable's parents have been revised, move onto the next variable in the ordering. See slide 6 for the pseudocode of this algorithm. Because DAC typically doesn't make the network fully arc consistent (does not revise the domains of the children given the constraints with the parents), DAC only requires $O(ek^2)$ to compute, where k is the size of the domain (since when? Wasn't the domain denoted d or a?) and e is the number of the constraints.

Example 4.4 in Decther's book is a brief demonstration of DAC.

Slide 8 (missing slide 7) – Directional Path Consistency (DPC)

DAC may not yield a strong enough consistency, so we may need to use DPC, which involves "moralizing" the graph (i.e., adding an edge between the parents of every node in an ordering if one doesn't exist). The pseudocode is on slide 9, but a description of the algorithm follows:

1. First, DPC enforces DAC on the graph as described above.
2. Then, again going bottom up (as in DAC), it revises all the constraints between every pair of parents of the current variable. If the constraint between the parents doesn't exist, it is added (moralizing). If it does, the existing constraint will be intersected with the projection of the existing relation onto the join of the two relations connecting the current variable to its parents (line 5 of the pseudocode). Additionally, it includes (and strongly recommended by Prof. Choueiry) the domain of the current variable in the join.

The algorithm will create the largest strong DPC network relative to some ordering *d*. The time and space complexity of the algorithm is $O(n^3k^3)$, where k bounds the domain size.

Here a brief discussion arose over the time complexity.

- Constraint composition (matrix multiplication) on line 5 is k^3
- For every variable, we consider every pair of parents. Thus, we consider every combination of 3 variables, which is $O(n^3)$.

Fikayo pointed out that lines 2 and 3 in the pseudocode require n^2 operations (as DAC is quadratic), and lines 4-6 require n^3 operations. She also mentioned that the parents are married on line 6 of the algorithm.

Slide 10 – Directional i-consistency (DiC)

Main idea: for every variable *k*, look at all of its parents, and instead of adding binary constraints (as in the DPC), add a constraint over every combination i-1 of the parents. So, for example, directional 4-consistency adds an arity-3 constraint between every combination of 3 parents for a variable (and for every combination of 3 variables, you can extend a solution to a 4^{th} variable). In general, you may need to add $\binom{m}{i-1}$ constraints to the graph to enforce DiC.

Discussion

Robert wondered how the algorithm for DiC on slide 11 is strong when line 5 of the algorithm only revises each subset of i-1 variables instead of all subsets of 2, 3, … (i-1) variables. The instructor suggested that strong consistency may be implicitly enforced by the in the revise function as follows: the constraints of arity (*i*-1) ensure that only tuples that can be extended to an i^{th} variable are kept. So, if we choose a tuple of size less than *i*-1, if it cannot be extended to a partial solution of size *i*, it cannot appear in any of the constraints of size *i*-1. The proof is Exercise 9 (page 113). The instructor asked Robert to look into it further.

Slide 12 – Graph aspects of DPC

The various terms in the slide were briefly reviewed because all had been previously covered, I believe, so I won't belabor them here. Of note: The instructor mentions that the usage of "induced graph" here is slightly bothersome as it doesn't specify "how" it was induced. It may be preferable to write: graph induced by triangulation.

Slide 13 – The induced-width

Correction: on the last bullet, "max" should be "min" (i.e., the induced width of a graph is the *minimum* induced width over all orderings).

4. Discussion/review of triangulation heuristics

When eliminating a vertex from a graph, we must connect all every two of its neighbors that are not already connected. (This operation is an example of *a graph reduction operation*.) Connecting the neighbors ensures that the information of the eliminated vertex and its constraints are now represented by the new constraints between its neighbors. The added edges are called '*fill-in edges*.' A good triangulation minimizes the number of fill-in edges. If you choose any ordering of vertices and apply the above vertex-elimination operator, you're guaranteed a chordal or triangulated graph as output.

Triangulation heuristic

Minimizing then number of fill-in edges is NP-hard, but there are some reasonable heuristics for approximating a minimal number of added edges.

1. *Simple heuristic*: Choose the smallest degree vertex, repeat. (Remember the min width algorithm of CSCE421/821?)
2. A Better Heuristic Min-fill: Remove the vertex that, after removal, adds the smallest number of fill-in edges. See Shant's working note[4] for an efficient implementation of this heuristic in $O(n^4)$.

Relation to DPC/DiC

With any triangulation heuristic, the fill-in edges yield a triangulated graph: the fill-in edges correspond exactly to the edges you would need to add when moralizing the graph with respect to the order used to remove vertices in the heuristic (elimination order). The ordering at the end of min-fill is a *perfect elimination ordering*, in the sense that if you follow that ordering you will not need to add any more edges. As a corollary, if the graph is triangulated, the reverse of the max-cardinality ordering is a perfect elimination ordering. Using min-fill is a reasonable way to keep the size of the *induced-width (w*) of the ordering* small, thus a reasonable approximation of the *induced width of the graph* (which is NP-hard).

[4] Shant's Min-fill Heuristic Implementation