Factor Graphs in Logic and Constraint Satisfaction

Frank Dellaert
College of Computing, Georgia Institute of Technology

February 1, 2012

Factor graphs can be seen as a unifying representation of *information about* and *relationships between* variables. They can represent statements in logic, constraints in constraint satisfaction problems (CSPs), sparse linear systems of equations, probabilistic relationships between variables, etc. Another important graph is the corresponding primal graph: it represents less detailed information than a factor graph but is useful to reason about structure. Below I illustrate both graph types in the context of logic and constraint satisfaction. I also briefly discuss two search-based algorithms to find satisfying assignments in both.

1 Logic

First, let us see how a factor graph can make propositional logic come alive. A **propositional variable**, also called Boolean or binary variable, can take on the value $true$ or $false$ (we often write 1 and 0). A knowledge base (KB) or **propositional theory** φ is a conjunction of logic sentences. An **assignment** or interpretation $\mathbf{x}$ is an assignment of true or false to all propositional variables. A **satisfying assignment** or *model* of the theory φ is an assignment to the propositional variables $\mathscr{X}$ such that the theory $\varphi(X)$ evaluates to true.

1	$M \Rightarrow I$
2	$\neg M \Rightarrow (\neg I \wedge A)$
3	$(I \vee A) \Rightarrow H$
4	$H \Rightarrow G$

Table 1: A simple propositional theory φ about unicorns.

Example. Let us consider a simple knowledge base (KB) about unicorns, from an exercise in AIMA:

"If a unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If a unicorn is either immortal or a mammal, then it is horned. A unicorn is magical if it is horned."

To state this in the language of propositional logic, let us define five propositional variables,

$$\mathbf{X} \stackrel{\Delta}{=} \{M, I, A, H, G\}$$

that correspond to *mythical*, *immortal*, *mammal*, *horned*, and *magical*, respectively. Then our knowledge about unicorns can be stated in the four propositional formulas listed in Table 1. It is easy to check that the assignment $(M, I, A, G, H) = (1, 1, 1, 1, 1)$ is a model for this theory.

Factor Graphs

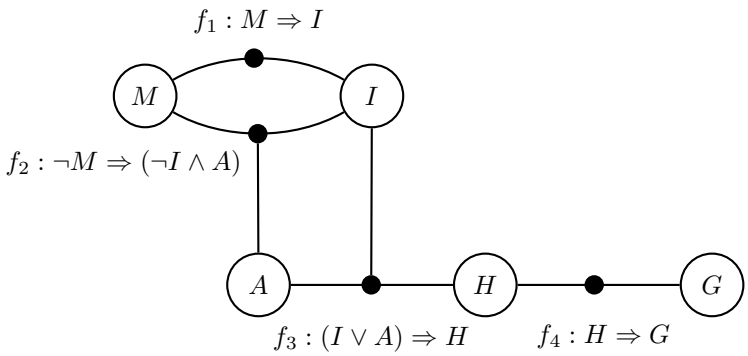

Figure 1: Factor graph representation of the propositional theory φ_1.

The compositional structure of a propositional theory can be made apparent by drawing a **factor graph**, a bipartite graph $G = (X, F, E)$ with one **variable node** $x_j \in X$ for every variable, one **factor node** $f_i \in F$ for every logic sentence. Given a factor graph G and the corresponding factors f_i, the propositional theory $f(X)$ is then obtained by joining all factors using the conjunction operator:

$$f(X) = \bigwedge_i f_i(X_i)$$

where $X_i \subset X$ is defined as the subset of variables connected to factor f_i. The factor graph for the unicorn example is shown in Figure 1.

Primal Graphs

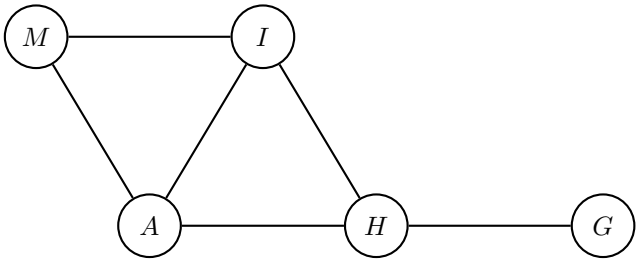

Figure 2: Primal or interaction graph corresponding to the factor graph in Figure 1.

Another graph that is useful is the variable *interaction graph*, or **primal graph**. This is an undirected graph $G = (X, E)$ with again one variable node x_j per variable, and an edge e_{jk} if two variables x_j and x_k interact. The primal graph for the unicorn example is shown in Figure 2.

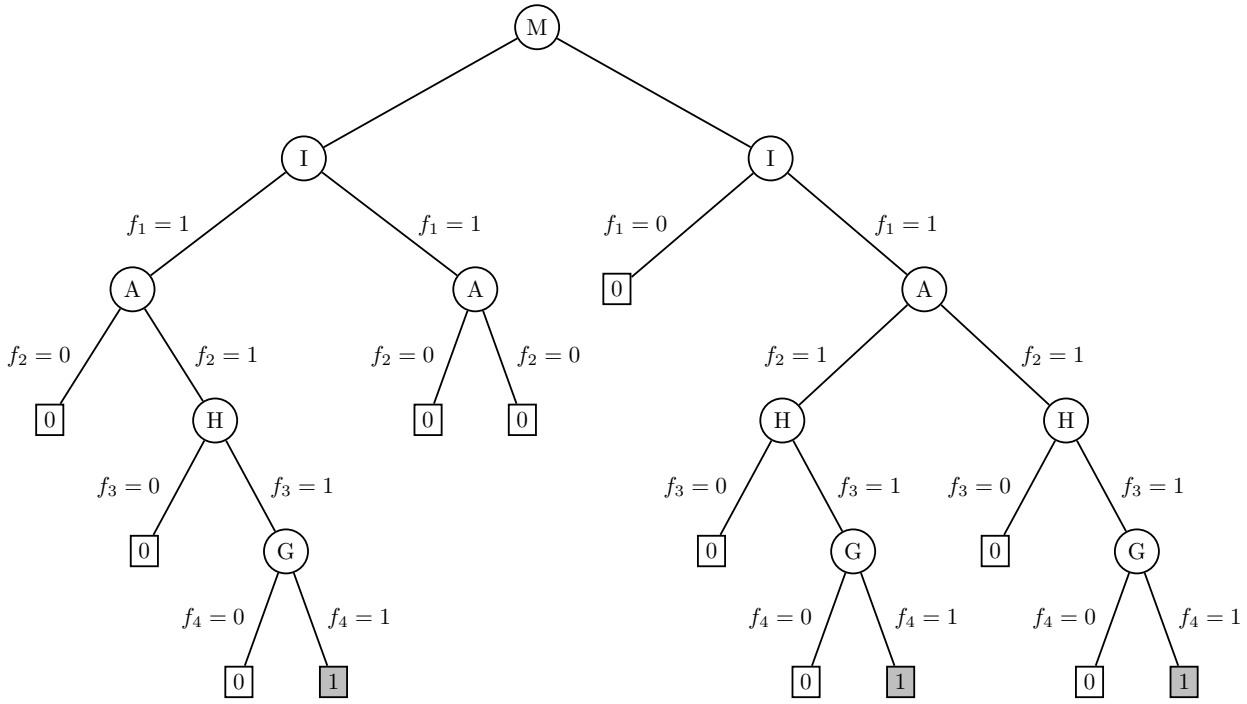

Figure 3: Search tree for satisfiability. At each variable node (e.g., the root) we either assign 0 to the corresponding variable (M for the root) by going left or 1 by going right, pruning subtrees whenever one of the clauses $f_1 \dots f_4$ evaluates to 0.

Satisfiability

One of the key questions to ask about any propositional theory f is whether it is **satisfiable**, i.e., does there exist an assignment of truth values to the variables X such that $f(X) = \bigwedge_i f_i(X_i)$ evaluates to 1. This is essential in reasoning, as any theorem α can be disproved (proved) by adding its negation $\neg\alpha$ to the theory and finding a satisfying counter-example (or proved by finding that the augmented theory is unsatisfiable).

An easy way to search for a satisfying assignment X is simply through **depth-first-search** in a tree that represents the entire truth table $f(X)$, evaluating each factor $f_i(X_i)$ as soon as the variables X_i it depends on have been assigned a value. Entire subtrees can be pruned if a factor $f_i(X_i)$ evaluates to 0, making it so that we do not have to search the entire (exponentially large) search space.

The process is illustrated for the unicorn example in Figure 3. The factor graph in Figure 1 acts as a guide throughout. Here, in a depth-first traversal, the leftmost subtree is pruned after assigning $M = 0$, $I = 0$, and $A = 0$ because $f_2(M,I,A) = \neg M \Rightarrow (\neg I \wedge A)$ evaluates to false. At the leaves we find that there are exactly three models, i.e., $(x_M, x_I, x_A, x_H, x_G) \in \{(0,0,1,1,1), (1,1,0,1,1), (1,1,1,1,1)\}$.

The best algorithms for satisfiability, which solve problems with millions of variables in reasonable time, are based on search. Industry-strength solvers use a variety of SAT-specific tricks, such as efficiently propagating the effects of assigning a value through the constraints, and remembering *nogood* assignments to subsets of variables. Interestingly, SAT problems typically fall into two classes: hard ones, and easy ones. The easy ones can often be solved by random guessing, the basis for algorithms like WalkSAT.

2 Constraint Satisfaction

Constraint satisfaction is a simple generalization of logic where variables are allowed to take on more values than just true or false. Problems such as Sudoku puzzles or graph coloring problems are examples of **constraint satisfaction problems** (abbreviated CSP), where a set of variables have to be assigned values while satisfying a number of application-specific constraints. CSP's have numerous applications such as scheduling and planning, and Boolean satisfiability is a special case.

Definitions

Formally, a CSP is defined by

1. a set of **variables** $X = \{x_j | j \in 1 \dots n\}$;
2. their **domains** $D = \{D_j | j \in 1 \dots n\}$;
3. a set of **constraints** $f_i : X_i \to \{0,1\}$, with $i \in 1 \dots m$.

An **assignment** A is a set of individual assignments (x_j, a_j) of a value $a_j \in D_j$ to variable x_j. Each constraint f_i is defined over its scope $X_i \subseteq X$, and returns 1 for a set of assignments A_i that are said to satisfy f_i, and 0 otherwise. A **satisfying assignment** A for the CSP is an assignment such that $\prod_i f_i(A_i)$=1.

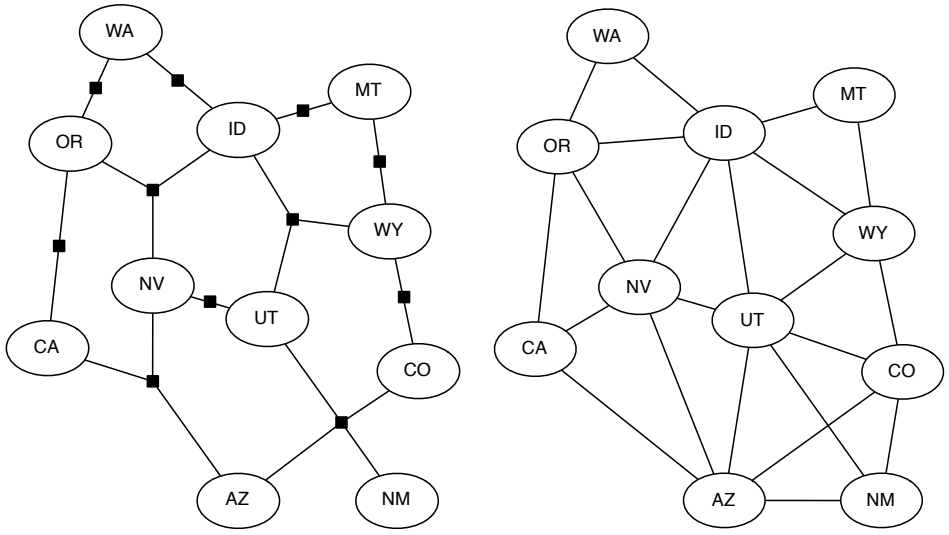

Figure 4: The factor graph and primal graph for a map coloring example, showing the western U.S.

Example. The factor graph and primal graph for a simple map coloring example are shown in Figure 4. In this example there are 11 variables and 11 constraints. The variables correspond to states on a map, and the constraints indicate that neighboring states should be assigned different colors. For example, suppose $D_{CA} = \{Red, Green, Blue\}$ and $D_{OR} = \{Blue, Green, Yellow\}$, then the constraint between CA and OR would include $f(x_{CA} = Red, x_{OR} = Green) = 1$ and $f(x_{CA} = Green, x_{OR} = Green) = 0$. Note that both binary, ternary, and quaternary factors (constraints) are used. In the primal graph, however, that information is lost: an edge only indicates that two variables are connected through *some* constraint.

Divide and Conquer for CSPs

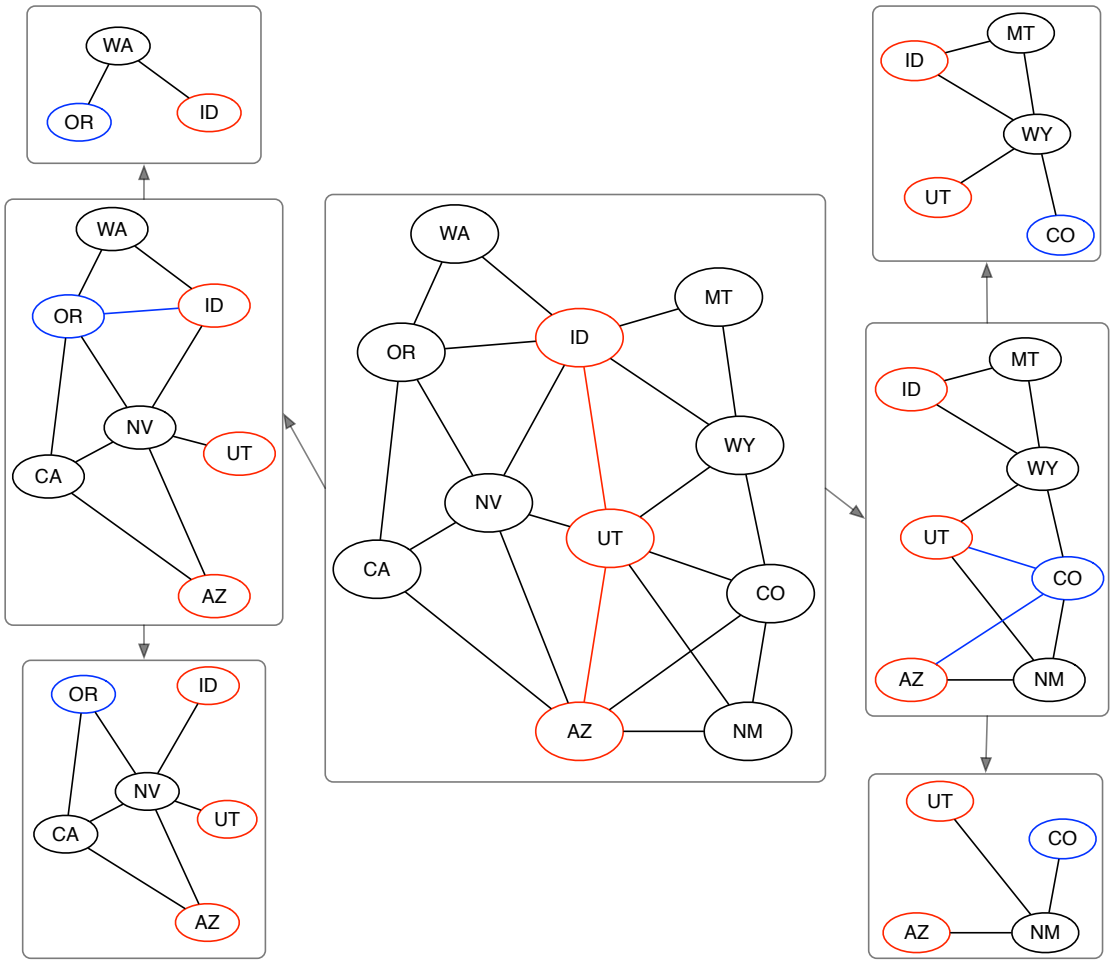

Figure 5: The nested dissection process on the example map coloring problem.

A simple recursive algorithm uses a divide and conquer approach to solve CSPs. In a nutshell, each recursive call to the algorithm partitions the given CSP into two smaller problems, conditioned on a separator in the primal graph which we enumerate over. The algorithm is illustrated in Figure 5 for the simple map coloring example. In the figure, the first separator is indicated by red nodes, and the second by blue nodes.

There are some ways to dramatically speed up this algorithm if you can afford some time (constraint consistency pre-processing and constraint propagation) or memory (by memoizing). In addition, the algorithm can be trivially modified to solve Boolean Satisfiability (a special case of CSP) or Constraint Optimization Problems (COP). I call this (unpublished sketch for an) algorithm a "nested dissection solver" as it is based on the so-named technique in sparse linear algebra, which was analyzed by our own Richard Lipton.

Acknowledgements

Collating this note was prompted by my giving a guest lecture in Merrick Furst's algorithms class. Thanks!