5

ARTIFICIAL INTELLIGENCE COM S 472/572 LECTURE NOTES WEEK 5

Author: Vasant Honavar
Scribe(s): Rich Miller, Venkatraman Naganathan, Anand Ganapathy

Computer Science Department,
Iowa State University, Ames, IA 50011

1 PROBLEM REDUCTION REPRESENTATION AND SEARCH

Example:

$$\int (2t + sec^2t\ sin^2t + 7\ )dt$$

AND

$$\int 2t\ dt + \int sec^2t\ sin^2t\ dt + \int 7\ dt$$

OR

$$\int sec^2t\ (1 - cos^2t)\ dt \qquad \int (1 + tan^2t) sin^2t\ dt$$

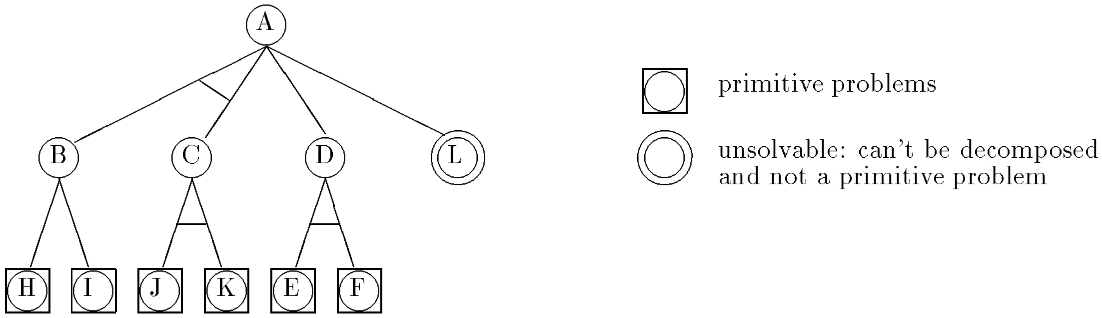
In many problems, it is natural to think of solutions in terms of subproblems. The problem is solved by successively decomposing or transforming it into subproblems until we end up with *primitive* problems (i.e. those problems whose solutions are known or can be looked up in a table or can be obtained using a known algorithm).

Such a decomposition is called problem reduction representation (PRR). A PRR is specified by the 3-tuple (G, O, P), where

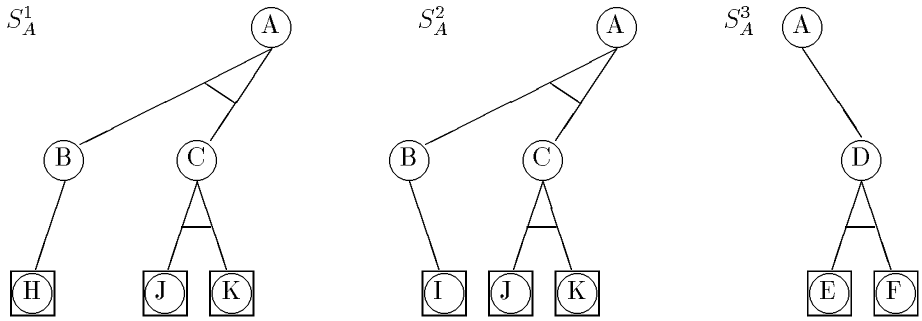
G = Problem to be solved
O = Set of operators for transforming a problem into subproblems using AND or OR decompositions
P = Set of primitive problems (those we know how to solve)

Such problem decompositions can be represented using **AND-OR graphs** in which the nodes represent problems and subproblems to be solved and the arcs have been generalized to AND and OR **connectors**. This was shown with the integral calculus example above.

Example of AND-OR graph:



There are 3 solutions to the above example:



Definition: A **subgraph** S_x of an AND-OR graph (G, O, P) is said to be a solution of problem x iff:

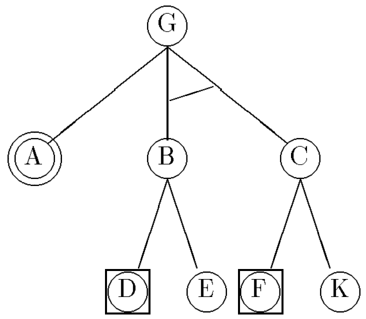
1. x is the root of S_x
2. for every non-leaf node $y \in S_x$, exactly one connector k going out of y along with all the nodes at the tip of the connector is in S_x
3. all leaf nodes in S_x are elements of P (the set of primitive problems)

Definition: We say that a node x is **solvable** iff

1. $x \in P$ (i.e., x is a primitive problem) **or**
2. x is a non-leaf node such that all its children at the tip of at least one of its connectors are solvable.

Essentially, solving a problem G using AND-OR graphs entails finding a solution to S_G. Now how can we generalize our search algorithms to include AND-OR graphs? Instead of placing nodes on a list maintained by the search algorithm, we place connectors or partial solution graphs on the list.

Example:

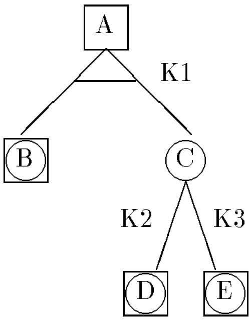


BFS
(G)
(A) (B & C)
if you find out A can't be solved then you get:
(B & C)
(D & F) (E & F) (D & K) (E & K)
Note that when the search terminates in this example, nodes E and K have not yet been checked; they may be primitive, but we do not yet know.

Exercise: Solve same example using DFS.

2 HEURISTIC SEARCH OF AND-OR GRAPHS

Example:

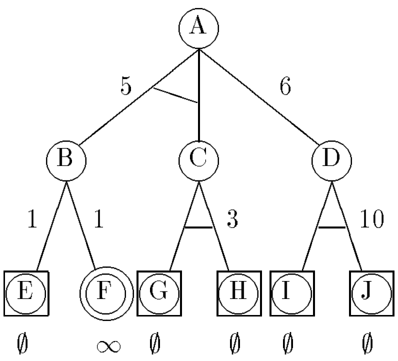


$Cost(S_A) = Cost(k_1) + Cost(S_B) + Cost(S_C)$
$Cost(S_C) = Min\{(Cost(k_2) + Cost(S_D), (Cost(k_3) + Cost(S_E)\}$

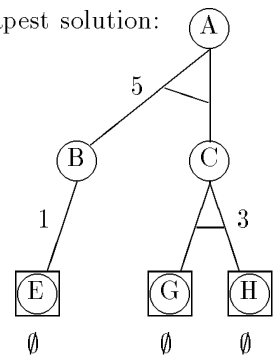
Cost of connectors and primitive problems are assumed to be positive and bounded.

$Cost(n) = \infty$ if n is unsolvable.

Example:



Cheapest solution:

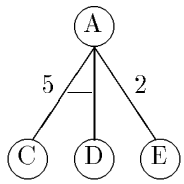


$S_{Amin} = 5 + 1 + 3 = 9$

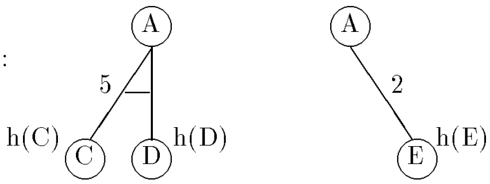
3 SEARCHING FOR AN OPTIMAL SOLUTION

We consider defining a heuristic function $h(x)$ for a solution graph rooted at x. If $h(n) \leq h^*(x)$ (where $h^*(x)$ is the cost of the cheapest (optimal) solution rooted at n), we say that $h(n)$ is **admissible** and we can easily adapt A* to an analogous algorithm AO* to work on AND-OR graphs.

Example: Calculation of f(x)



Two partial solutions:



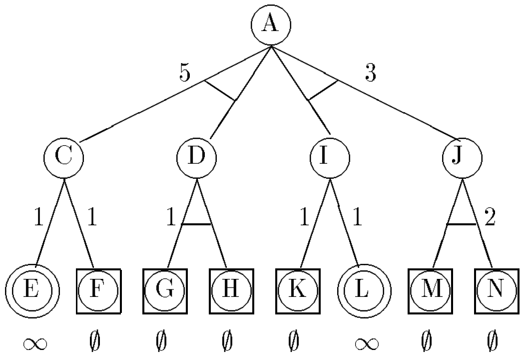
$f(A) = min[f_1(A), f_2(A)]$ is the estimated cost of the cheapest solution rooted at A.

The above graph has two partial solutions (as shown) with the following costs:

$f_1(A) = 5 + h(C) + h(D)$
$f_2(A) = 2 + h(E)$

Example:

$h(C) = h(D) = h(I) = h(J) = 1$



Search List:

(A)
(I & J) (C & D)
5 7
(K & M & N) (C & D)
6
(M & N) (C & D) because K is terminal
(N) (C & D) because M is terminal
() (C & D) because N is terminal
Algorithm terminates with empty list.

Results:

- AO* is admissible given an admissible heuristic h.
- AO* is an optimal search algorithm.
- Therefore, AO* is an optimal admissible algorithm.
- Any algorithm with an f function which is of the form, $f(x) = c_1 g(x) + c_2 h(x)$ cannot do better than AO*.

4 CONSTRAINT SATISFACTION PROBLEMS (CSP)

4.1 Background Information

Examples of CSPs:

(1) 8-queens problem
(2) Job shop scheduling
(3) Scene labeling
(4) Printed circuit board (PCB) layout

CSP procedures are search problems with some special properties:

- The search space is finite.
- The depth of the search tree is fixed (the number of variables).
- Subtrees have similar topologies, so experience obtained in searching one subtree may be exploited when others are searched.
- The search space has minimal size under certain conditions.

4.2 Definitions

Formal Definition of CSP:

The **domain** D_x of a variable x is the set of possible values that can be assigned to x.

A **label** is a variable-value pair (x,v) which represents the assignment of value $v \in D_x$ to variable x.

A **k-compound** label assigns values to variables:
$(x_1, v_1)(x_2, v_2) \cdots (x_k, v_k)$

A **solution** to a CSP with N variables is an N-compound label satisfying all the problem constraints.

A **CSP** is specified by the 3-tuple (Z, C, D), where
Z = set of variables.
C = set of constraints.
D = $\{ D_x | x \in Z \}$

4.3 Examples

In this section we will examine the 8-queens problem and the scene labeling problem in greater detail to illustrate how the general definition we have just given for CSPs can be applied to specific problems.

The 8-Queens Problem

This problem, first posed in a German chess magazine almost 150 years ago, requires that 8 queens be placed on an 8×8 chessboard in such a way that no two queens attack each other, i.e. no two queens occupy the same row, column, or diagonal.

This problem can be formally represented as a CSP in several different ways, and as we shall see, the choice of formal representation has a significant impact on the computational complexity of the problem.

Suppose, for example, that we choose to represent the problem by numbering the queens 1 through 8 and then defining a set Z of 64 variables, one for each square of the chessboard, where each variable has domain $D = \{0, 1, 2, \cdots, 8\}$ and a variable z having a value of i means that queen i occupies the corresponding square if $1 \leq i \leq 8$ or that the square is unoccupied if $i = 0$. Although this representation is formally valid once the constraints have been appropriately formulated, the cardinality of the state space is $9^{64} \approx 1.18 \times 10^{61}$, making it impractical from a computational perspective.

Consider instead a representation with a set Z of 8 variables $Q_1, Q_2, \cdots Q_8$ representing the 8 rows of the board, with the domain of each variable Q_i being the set $D_i = \{1, 2, \cdots, 8\}$; in this case the value of the variable Q_i represents the column in the i^{th} row in which a queen is placed. The constraints can then be specified as logical expressions of the form
$i \neq j \Rightarrow Q_i \neq Q_j$ (i.e., no two queens are in the same column) and
$i \neq j \Rightarrow |Q_i - Q_j| \neq |i - j|$ (i.e., no two queens are in the same diagonal).

Note that we have implicitly incorporated one of the problem constraints (that no two queens occupy the same row) as well as our reasoning about the consequences of this constraint (it implies that each row must have precisely one queen in it) into our representation. Note too that the cardinality of the search space is now reduced to $8^8 \approx 1.68 \times 10^7$ states.

This example clearly illustrates the need to carefully select a representation for a CSP from among the many formally valid possibilities, bearing in mind the computational complexity which each one entails.

The Scene Labeling Problem

This is a subproblem of the **vision problem**, which seeks to deduce facts about the physical world from visual representations of it. In the scene labeling problem we are given a 2-D line drawing of a 3-D scene and must label the components of the drawing in such a way as to make explicit the 3-D structure the drawing represents.

This problem has been studied extensively since the early 1970s; we will consider a simplified version which was studied by David Waltz as part of his Ph.D. thesis at M.I.T. during this period. The assumptions we make to achieve this simplification are:

1. Every vertex is **trihedral** (formed by the junction of three planar surfaces; implicitly, we assume that all surfaces are planar).
2. Surfaces do not contain 0-width **cracks** (lines formed by the junction of coplanar surfaces).
3. No **shadows** (lines which represent projections of edges onto surfaces along the direction of light sources) are allowed.
4. The scene is viewed from a **general viewing position** (roughly, for some sufficiently small $\epsilon > 0$ we can move our viewing point in an arbitrary direction by a distance of ϵ without any topologically significant change in the resulting 2-D representation).

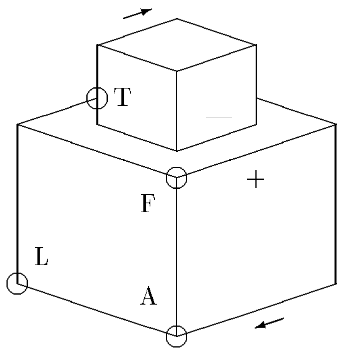
It actually turns out, as was revealed by later work on the problem, that assumption 3 is not really a simplification, since the information which can be deduced from the presence and positions of shadows in some sense outweighs the potential for confusion with actual edges they bring with them, but we will follow Waltz' path here and let the assumption stand.

Although a detailed analysis of the problem is beyond the scope of this lecture, one of the fundamental results of this analysis is that a relatively simple taxonomy of 4 classes of vertices and 4 classes of lines suffices to cover all physical possibilities, modulo the assumptions we have made. While we will not give formal definitions of the classes here, they may be intuitively described as follows:

1. **L-Junction Vertices**: vertices at which only 2 visible lines meet.
2. **T-Junction Vertices**: vertices at which one line segment ends in the middle of another.
3. **Fork Vertices**: vertices at which three lines meet, with no angles greater than or equal to 180° between any pair of them.
4. **Arrow Vertices**: vertices at which three lines meet, with an angle greater than 180° between one pair of them.

1. **Convex Lines**: lines formed by the junction of two visible surfaces for which a line segment connecting the two surfaces would be inside the solids they bound.
2. **Concave Lines**: lines formed by the junction of two visible surfaces for which a line segment connecting the two surfaces would be outside the solids they bound.
3. **Right/Up Arrow Lines**: lines formed by the junction of one visible and one obscured surface, with the visible surface lying below/to the right of the line.
4. **Left/Down Arrow Lines**: lines formed by the junction of one visible and one obscured surface, with the visible surface lying above/to the left of the line.

Representative examples of each of these classes may be seen in the following drawing:



L (L-Junction Vertex)

T (T-Junction Vertex)

F (Fork Vertex)

A (Arrow Vertex)

$+$ (Convex Line)

— (Concave Line)

$\nearrow$ (Right/Up Arrow Line)

$\swarrow$ (Left/Down Arrow Line)

As we shall see, the classification of drawing features in this way allows us to pose the scene labeling problem as a CSP, with vertices as variables to be labeled with values from the list of classes above; since each class of vertex can only have certain classes of lines meet to form it, the restrictions imposed by the classification of the two vertices at the endpoints of each line must be compatible, which then provides us with the constraints.