Constraint Satisfaction Problems: A Deeper Look

The last problem set covered the topic of constraint satisfaction problems. CSP search and solution algorithms are directly applicable to a number of AI topics so it's important to have a good grasp of what's going on underneath the hood. As we shall see, CSP is just another state search problem in the end. However, problem state representation plays a major role in how easy it is to implement the search. In the following document, we'll talk about different representations of a smaller version of the Zebra problem and various state tree traversals that take place when searching over the potential solutions to this problem.

The Problem:

Consider three houses in a row, each house of a different color and each house preferring a different type of donut.

1) The left house is red.
2) Winchell's donuts are eaten in the yellow house.
3) Krispy Kremes are eaten in the house next to the blue house.

In which house do they eat Top Pot donuts?

Representation:

There are a number of ways to represent the problem. The choice of representation doesn't directly affect the back-tracking search algorithm, but does directly affect how constraints will be modeled and checked.

Representation #1

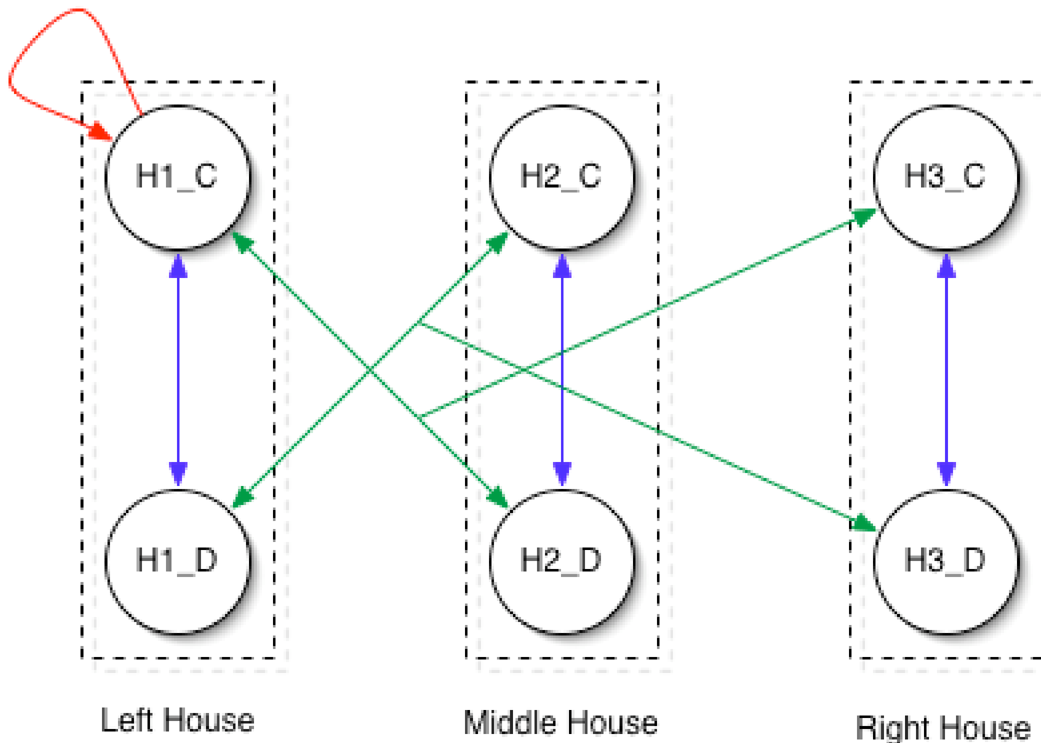
Consider a representation that has a variable to each houses' color (C) and each houses' donut preference (D). That might make the set of variables:

VARIABLES = { H1_C, H1_D, H2_C, H2_D, H3_C, H3_D }

with the following values:

H#_C = { red, yellow, blue } = {R,Y,B}
H#_D = { Krispy Kreme, Top Pot, Winchell's } = { KK,TP,W }

We can draw a constraint graph for this representation where each variable gets a node and constraints between variables are represented with arrows.



In this graph, constraint #1 is represented with the red arc, constraint #2 with blue arc, and constraint #3 with the green arc. As you can see, we apply constraint #1 only to the left house. Constraint #2 is applied to the Color / Donut variable pairing of each house. Constraint #3 is by far the most complicated. The donut choice of of House #2 potentially constraints the house color of both House #1 and House #3 and vice versa. The same connections apply for the H1_D, H2_C, H3_D constraint arc.

Representation #2

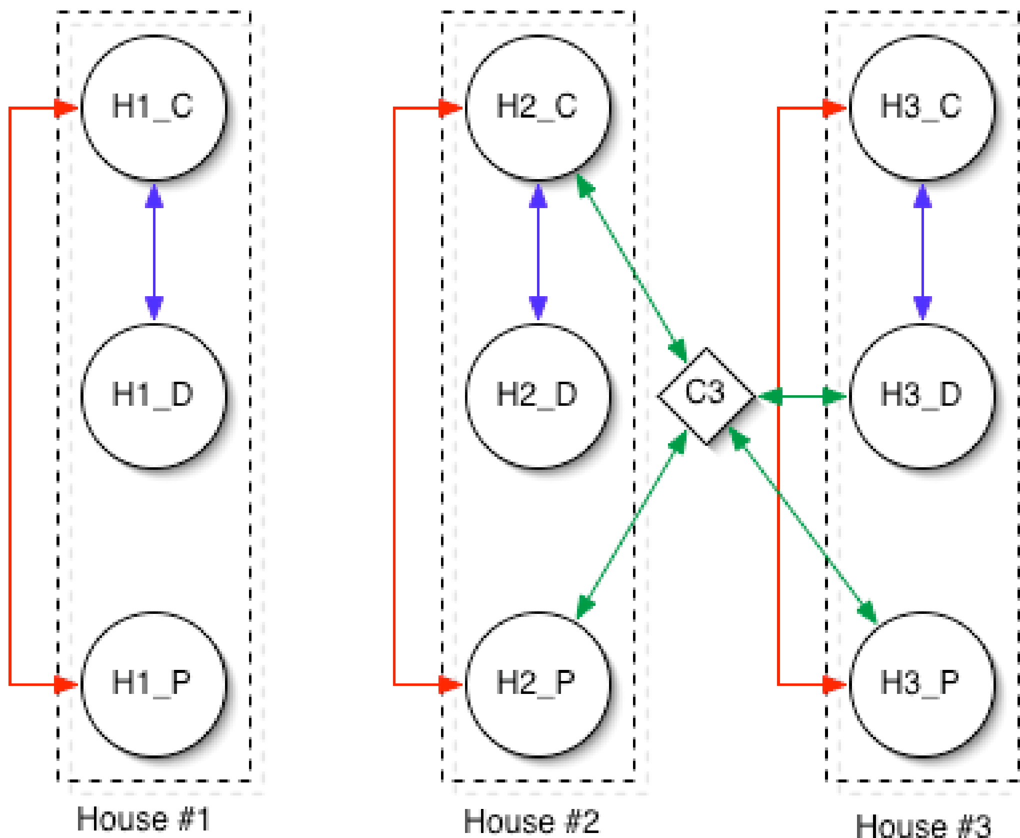
Another representation makes the position of each house a variable as well. So in addition to the variables of Representation #1, we have an additional Hn_P variable for each house.

VARIABLES = { H1_C, H1_D, H2_C, H2_D, H3_C, H3_D, H1_P, H2_P, H3_P }

with the following values:

H#_C = { red, yellow, blue } = {R,Y,B}
H#_D = { Krispy Kreme, Top Pot, Winchell's } = { KK,TP,W }
H#_P = { left, middle, right } = { L, M, R }

We can draw a constraint graph of this representation as well.



The constraint arc for constraint #1 has now expanded to each house and now connects house color and position. Constraint #2 looks largely the same. Constraint #3 looks a lot different, as it now connects four variables. The graph representation is simplified, since it only shows one of the arcs groups for constraint #3. There are actually similar constraint arc groups for each combination of Hn_C and Hn_D with the requisite links to house position. Needless to say, representation #2 makes constraint modeling slightly more complicated.

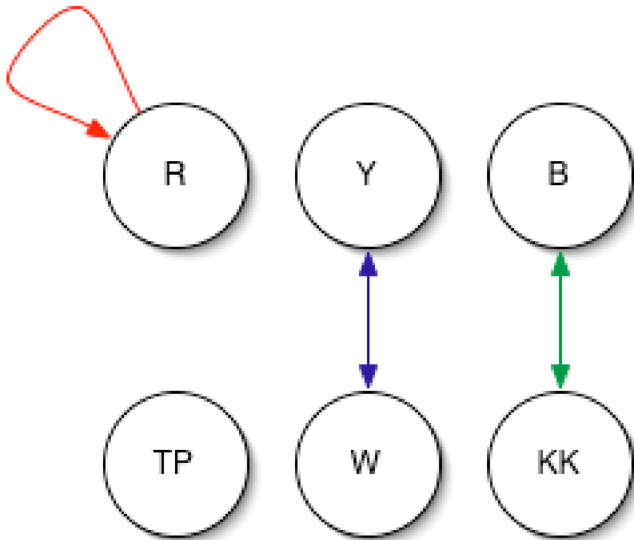
Representation #3

A final representation takes advantage of the fact that a value can only be assigned once to each variable. That is to say, there is only one yellow house and one house where Top Pot donuts are eaten. In AI search problems, we often talk about transforming our search space into a different domain in order to make the search easier. We can apply this advice in our current problem by realizing that we can treat the variable values as variables themselves. Consider our original use of the H1_C variable to represent the color of house #1 with possible values of red, yellow, and blue. Instead now consider three variables YELLOW, RED, and BLUE, each of which can have an integer value representing the number of the house. This representation seems backwards to our initial conception, but it makes constraint much easier, as we

shall soon see.

VARIABLES = { YELLOW, RED, BLUE, TP, KK, W }
VALUES = { 1, 2 ,3 }

Let's take a look at the constraint graph.



Now, I hope you agree that this graph is much simpler than our initial representations. We see the biggest benefit in modeling "next to" relationships. The multiple constraints of representation #1 and #2 for the third constraint are reduced to just one constraint. How is this possible? To check that Krispy Kreme donuts are eaten in the house next to the blue house, we now only have to check that "abs(B - KK) == 1". That is to say, the house number value for B must be one away from the house number value for KK. Good job to Ita who pointed this representation out to me in office hours and kudos to anyone else who thought of this representation as well.

Constraint Graph versus Assignment Graph

Once we've settled on a representation, we can begin the task of encoding the representation. One of the questions raised in class concerned representing the CSP as a graph: what the graph looked like and how it was traversed. I think some of the confusion stems from the fact that there are really two graphs in play. I will talk about each in the following section:

Graph #1: The Constraint Graph

The first graph is the one we've seen in the previous section: the constraint graph. This graph models the constraint connections between variables in the problem representation. We've seen that this graph is highly dependent on the representation chosen for a given problem. In all cases, however, variables are represented as

nodes and constraints are represented as arcs between variables.

A major point of confusion about the constraint graph is that, unlike almost every graph and tree we've worked with in class and homework, this graph did not need to be searched (traversed) for the homework implementation. Constraint graphs are most useful when full constraint propagation by way of arc consistency (see page #145) is implemented. Constraint propagation is a heavier-duty version of forward checking, which we will discuss in moment. However, full arc consistency graph traversal is not needed to implement forward checking.

Graph #2: The BACKTRACKING-SEARCH Algorithm Assignment Graph

The second graph is actually a tree and it is definitely one that is traversed when using backtracking search. Let take a look at the BACKTRACKING-SEARCH algorithm (see page 142).

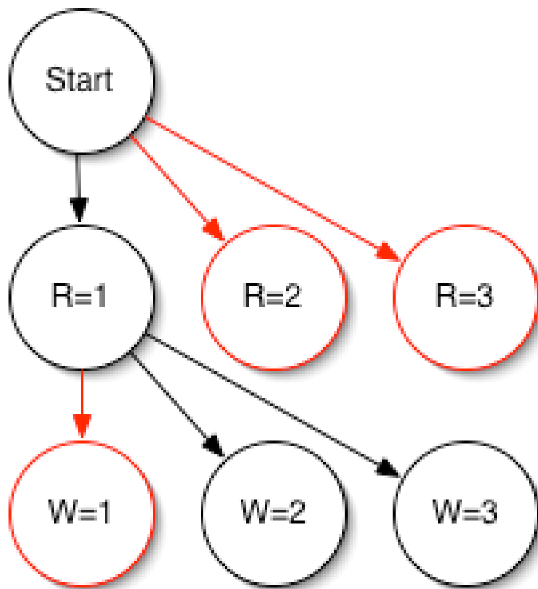
```
function RECURSIVE-BACKTRACKING( assignment, csp ) returns a
solution or failure
     if assignment is complete then return assignment
     var = SELECT-UNASSIGNED-VARIABLE( assignment, csp )
     foreach value in POSSIBLE-VALUES( var, assignment, csp ) do
          if value is consistent with assignment,csp then
               add { var = value } to assignment
               result = RECURSIVE-BACKTRACKING( assignment, csp )
               if result != failure then return result
               remove { var = value } from assignment
          endif
     endfor
     return failure
```

Let's talk over what exactly is going on here. The RECUSRIVE-BRANCHING (RB) function is initially with no variables assignments and a representation of the CSP. We select an unassigned variable and figure out what values it could possibly be (based on potential values that haven't already been assigned to something else). For each of these potential values, we check if its assignment is consistent with the CSP, and, if so, recursively call RB.

We know that any quality search tree is going to have depth and branching and the RB function is no exception. Depth comes from recursive calls to the RB function and branching comes from the "for each value" iteration of potential variable values. So what is the branching factor and depth of our CSP assignment tree? The depth is equal to the number of variables in our representation (d=6 for representation #3) and branching factor is equals to the number of possible values for variable (b=3 for representation #3, though that value does go down as assignments are made).

So we start at the root of our tree with no assignments and work our way to the end of a

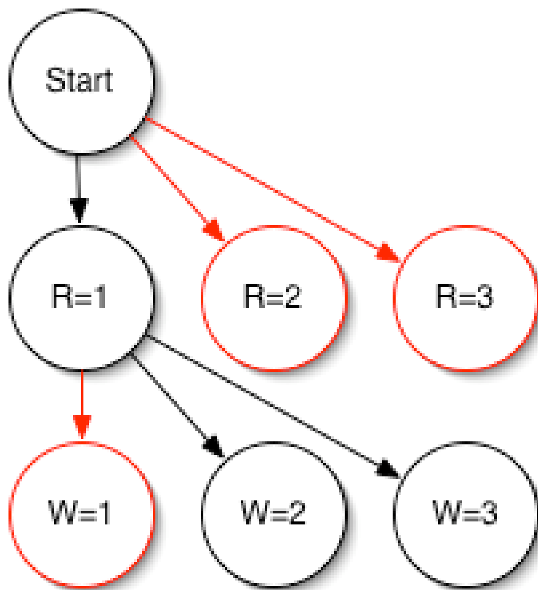
branch, at which point we've assigned some value to every variable in the problem. Each leaf of the tree represents a distinct set of assignments for the set of variables so a complete traversal of the assignment tree will enumerate ALL possible assignments. Obviously, that's a lot of work, so we'd like to avoid searching all those branches if possible. That is where the "if value is consistent with assignment according to csp" comes into play. There is no point exploring the remainder of a branch if we've found a variable in the branch that puts the CSP in an inconsistent state. Consider the following example:



We start with no assignments and then select the red variable (R) for consideration. Our tree branches in three ways for each of the possible values of RED (House #1, #2, and #3). However, only assignment R=1 meets constraint #1 that the left house is the red house. Thus, the branches R=2 and R=3 go no further. The R=1 keeps going and recursively calls the RB function. We next pick the Winchell's donuts variable (W) for consideration. We again consider three values (the three houses) and branch for each value. Here, we see that the W=1 assignment is in conflict with constraint #2 (though we'd only catch this with forward checking... see below) so that branch goes no further. The remaining branches keep going. Eventually, if the CSP has a solution, one of the branches will make it all the way to the bottom of the tree and find a complete and consistent assignment. At this point, we've completed our graph traversal and found a solution.

Forward Checking

As I mentioned in the previous section, some constraint inconsistencies will only be caught by forward checking. Let's go back to the previous example:



Without forward checking, the assignment of W=1 would not actually cause an inconsistency. Why not? We know by constraint #2 that only the Yellow house can enjoy Winchell's donuts, but our assignment graph states that house #1 is red, not yellow as constrained. Our representation of constraints using representation #3 states only that the values of the Yellow variable and the Winchell's variable must be equal. However, we have made no assignment to the Yellow variable in our assignment graph yet. Thus, the constraint checks out. We may not make an assignment to the yellow variable until much deeper in the tree, which means will be doing lots of unnecessary tree traversal before eventually finding the inconsistency.

Forward checking alleviates this problem by considering not what values a variable can be but instead what values a variable cannot be. Consider our assignment of R=1 in the first branch. Now that house #1 is red, it cannot be another color as well. Thus, the yellow variable and blue variable can now only be 2 or 3. Next, when we assign W=1, forward checking constraint #2 determines that the yellow variable can only have a value of 1, which means it cannot be 2 or 3. At this point, the yellow variable's potential value set is empty and the current assignment is inconsistent.

While forward checking is more powerful about catching future inconsistencies, it doesn't catch everything. As we mentioned previously, there are cases where only full constraint propagation through arc consistency will find all future inconsistencies. Consider the following partial assignment: R=1, KK=2, W=3

Consider the possible values that can be assigned to each variable at the beginning of the CSP search:

R=123 Y=123 B=123 KK=123 W=123 TP=123

Assigning R=1 gives us (satisfying constraint #1):

R=1__ Y=_23 B=_23 KK=123 W=123 TP=123

Assigning KK=2 gives us (by constraint #3):

R=1__ Y=_23 B=__3 KK=_2_ W=1_3 TP=1_3

Assigning W=3 gives us (by constraint #2):

R=1__ Y=__3 B=__3 KK=_2_ W=__3 TP=1__

With forward checking, we have not caught any problems, yet a big problem is immediately obvious. House #3 can't be yellow and blue at the same time, yet that is the only assignment left. Implementation of full arc checking to catch problems like this was not required for the homework assignment and is beyond the scope of this document (see page 145 for more details). However, do remember that forward checking can prune inconsistent branches from the backtracking assignment tree that simple constraint checking would not catch, and also remember that there are more powerful methods than forward checking when speed really counts.