Unit 2
Constraint Satisfaction Problems

Context:

A variety of interesting problems, which are also search problems, are related to finding a solution to problems that require us to ***find suitable values for a set of variables*** of interest, in such a way that a ***set of pre-defined constraints is respected*** by our chosen assignment.

A classic example of this type of problem is ***scheduling***. The Registrar's office must solve this problem each term: Assign courses to classrooms and time-slots in such a way that:

- Courses taught by the same instructor don't happen at the same time
- Courses commonly taken together by many students don't happen at the same time
- Courses are assigned to a classroom of appropriate capacity

As you know from experience, it's likely impossible to find a perfect solution to this problem in the setting of UTSC courses! But we must try and find a mapping of courses to time-slots and classrooms that is as close as possible to what we want.

In this unit, we will look at how CSPs are specified, and at the process used for solving them.

Components of a CSP

- A ***set of variables*** whose values are to be set
- The ***domain*** for each of the variables (for example, in the scheduling case, the different time slots that courses can be assigned to)
- A set of ***constraints*** that apply to individual variables, or that relate subsets of variables and determine which value assignments are ***valid*** and which are not.
- Constraints can be ***unary*** (applying to a single variable, e.g. CSCD84 ***can not be scheduled*** at 12 (noon) because the instructor has lunch at that time); ***binary*** (relating 2 variables, e.g. CSCD84 and CSCA48 can not be scheduled at the same time because they have the same instructor); or ***higher order*** (relating more than 2 variables, e.g. CSCA08, CSCA48, and CSCA67 can not be scheduled at the same time because they all require the same auditorium room).

A common way of representing a CSP is as ***a graph, with nodes corresponding to variables and edges corresponding to constraints***.

The type of CSP we will be studying in this Unit is characterized by:

* Variables are categorical (i.e. the values we can assign to them come from a distinct, finite subset of possible values). It is possible for the actual values assigned to be real numbers, but they have to come from a finite subset of possibilities
* Under these conditions, solving a CSP becomes a combinatorial optimization problem, and as such, a solution can be found through search of the finite solution space.

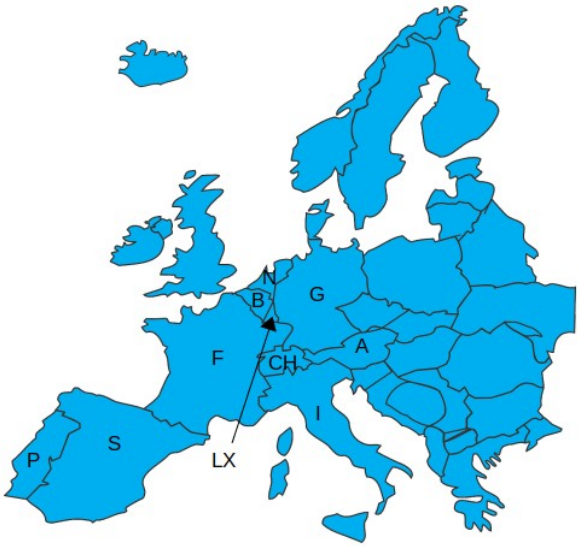
(c) 2020 – F. Estrada

Example

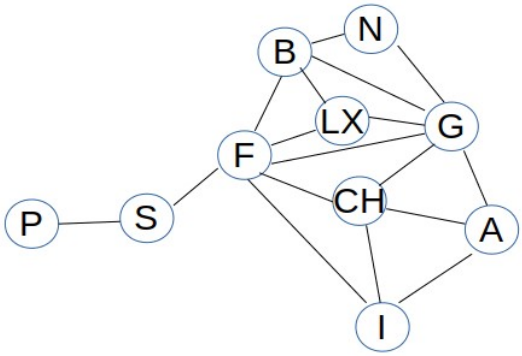
To understand the CSP formulation and the solution process, we will take a look at a traditional constraint satisfaction problem: Graph colouring – given a graph, and a subset of colours, we want an assignment of colours to nodes such that no two adjacent nodes (nodes directly connected by an edge) have the same colour.

This problem is interesting not just as a model for CSPs, it turns out to be a very close analogy to the scheduling problem we discussed above: Given a graph whose nodes are courses, and courses that can not be scheduled at the same time are connected by edges, the problem of assigning time-slots to courses is equivalent to the problem of assigning colours to the nodes in the graph.

To study the graph colouring problem, let's first build a graph. Consider the figure below:



Map courtesy of pixbay



We are going to use graph colouring to colour the map of Europe shown above (or at least a part of it) so that countries that share a border don't get the same colour. The corresponding graph has one node per each of the countries we labeled in the map, and an edge between any pair of countries that has a shared border.

The ***domain*** of our variables will be the set of colours {R,G,B,Y} (red, green, blue, and yellow).

Given the graph and the domain for our variables, solving the CSP is straightforward (in the simplest form, which does not say anything about whether the solution can be found in an acceptable amount of time!).

(c) 2020 – F. Estrada

Backtracking Search

The search process is a variation of DFS, and keeps track of the assignments of values to variables, the goal of the process is to generate a complete assignment of values to variables in such a way that no constraints are broken (we will discuss later what to do if no such assignment exists).

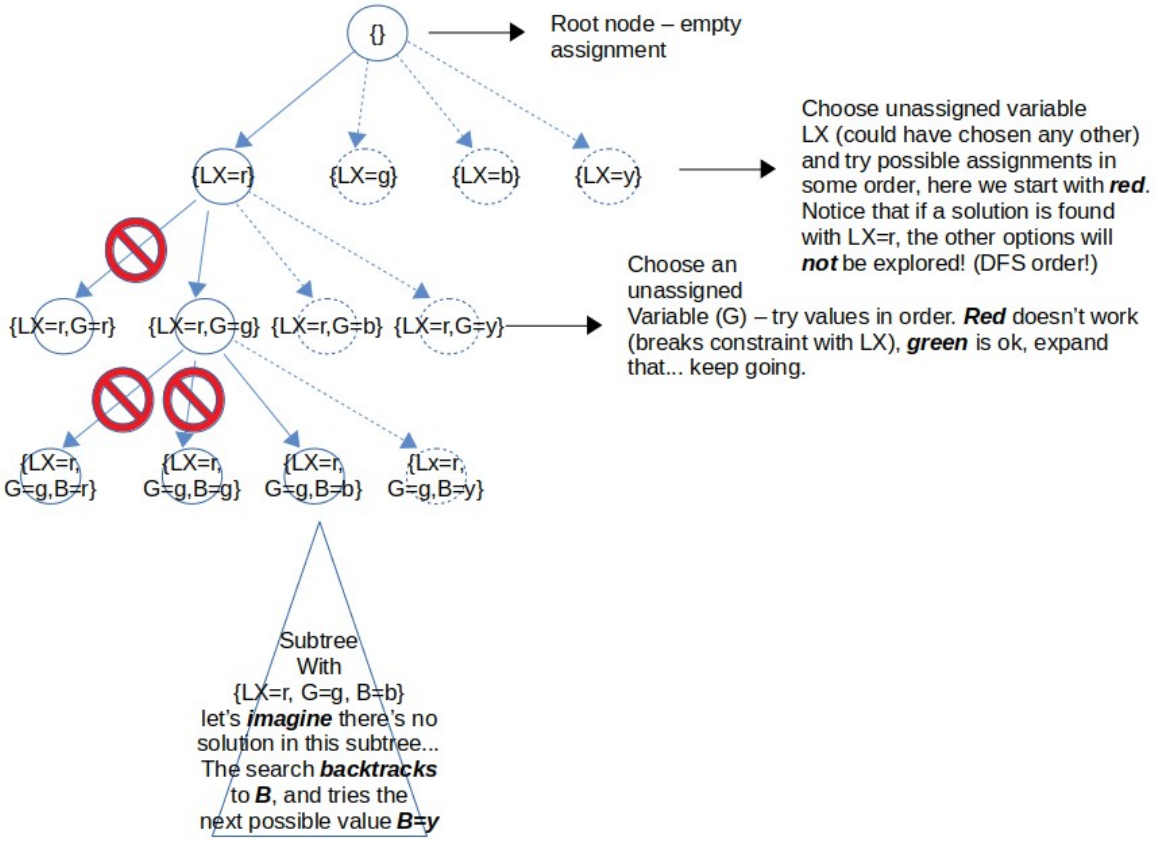
```
Input: Set of variables csp_variables
       Set of constraints csp_constraints
       Initial assignment as a set with tuples <variable:value>.
           // Initially empty, or with any initial conditions given to us (e.g. we
           // may have decided that in our graph colouring problem above, France
           // should be blue - the initial state will contain this tuple)

recursive_backtracking(assignment, csp_variables, csp_constraints)

    if is_solution(assignment)
    {
        return success
    }
    else
    {
        // Choose a variable to assign
        var <-- select_unassigned(csp_variables)

        // Try each possible value for this variable, one by one
        for each value in var.domain
        {
            add <var:value> to assignment

            if  consistent(assignment,csp_constraints)
            {
                result=recursive_backtracking(assignment,csp_variables,csp_constraints)

                if result==success
                    return success
            }
             remove <var:value> from assignment
        }

        return failure
    }
```

The search process above will find a solution, if one exists, by starting from whatever initial assignment it was given, and assigning values to variables one by one, growing the potential solution as long as the values assigned are consistent. In the worst case, this process will try every possible combination of values to variables only to find no possible assignment exists that respects all constraints given!

The process is called ***backtracking search*** because the search function returns ***failure*** when it is unable to grow the solution given the input assignment, and so the search process goes back to the ***previous*** variable that was assigned and tries to grow a solution with a different value for this variable, if that is not possible for any values of this variable then it ***backtracks*** one more assignment and tries again, and so on.

Example of Backtracking Search

In the graph above, backtracking search would produce a ***search tree*** that looks as below:



Keep in mind:

- The search tree could be huge. Imagine for a moment (though it's probably not the case in this particular problem!) that there's no solution under the branch for {LX=r}, it may take a lot of searching to determine this is the case, and to backtrack to the next assignment for LX.

- This means that our naive search procedure, while able to discover the solution if one exists in the tree, may take a very long time to find it.

- There must be something smart we can do to help the search along...

Bookkeeping

In order to be able to speed up the search process, we want to do a bit of book keeping.

- We will keep track of the set of possible values that can be assigned to variables during search.

(c) 2020 – F. Estrada

Example:

In the map colouring problem above, ***initially*** all countries can be assigned colours {r,g,b,y}. We keep a set like this for ***each country***.

Now, like in the example, assume our first assignment to try is {LX=r}. This immediately constrains the valid choices for neighbours of LX: B, F, and G. For each of these countries, we ***remove*** 'r' from the set of remaining valid assignments, and we get {g,b,y}.

Continuing with the example, our next assignment is {LX=r, G=g}. This constrains the neighbours of G: B, N, CH, F, A, and LX.

B's set now contains {b,y} only
N's set contains {r,b,y}
CH's set contains {r,b,y}
F's set contains {b,y}
A's set contains {r,b,y}
and LX ***has already been assigned*** to LX=r

Notice that at this point, different countries have different sized sets of possible valid values. This information is extremely useful in helping us order our search.

Speeding up Backtracking Search

There are heuristics that can be applied to speed up search by being careful about the ***order*** in which we expand variables in our problem, and given a choice for a variable to expand, the order in which we try values for this variable.

These heuristics ***which are applied in order*** are:

1) Choose the variable with ***least remaining values*** first. This means we look at the sets of valid values remaining for each variable in our problem, and choose the one with the smallest set to expand next. This also means that ***whenever any of these sets becomes empty, we backtrack immediately!*** Our naive search could have done a lot of work only to find out that there was no solution inside a large subtree which already contained variables that could not be assigned to any value without breaking constraints.

- The goal of rule 1) is to reduce the search space by creating a tree that is less wide at every step.

2) ***If more than one variable can be chosen by 1),*** choose first the ***variable involved in the largest number of constraints.*** This means that if several variables have the same number of possible assignments left, we should give preference to variables that are involved in a larger number of constraints.

(c) 2020 – F. Estrada

- The goal of rule 2) is to reduce the set of possible values left for as many variables as possible, as quickly as possible.

Note: If there is a tie by rules 1) and 2), we can choose any of the variables that are in a tie.

3) Given a choice of variable to expand***, choose first the least constraining value***. This means that instead of trying values to assign is some fixed, arbitrary order, we will try first values that ***remove as few possible assignments from neighbour nodes*** as possible.

Example:

In the map colouring problem above, suppose we have assigned:

{LX=r, G=g}

And suppose we are going to assign a value to CH next.

The neighbours of CH have the following valid sets at this point:

F:{b,y}
A:{r,b,y}
I:{r,g,b,y}
G: already assigned ***green***

We can't choose ***green*** for CH because its neighbour G is already green.

- If we choose ***red***, the remaining value sets for its neighbours become
F:{b,y}
A:{b,y}
I:{g,b,y}
G: already assigned ***green***
(2 values were removed from neighbour's sets)

- If we choose ***blue,*** the remaining value sets for neighbours become
F:{y}
A:{r,y}
I:{r,g,y}
G: already assigned ***green***
(3 values were removed from neighbour's sets)

- If we choose ***yellow***, the remaining value sets for neighbours become
F:{b}
A:{r,b}
I:{r,g,b}

(c) 2020 – F. Estrada

G: already assigned ***green***
(3 values were removed from neighbour's sets)

So we choose ***red*** first – this removes the fewest valid options from neighbours ***thereby increasing the chance we can find a solution in the subtree with*** {LX=r, G=g, CH=r}.

- Rule 3) has the goal of maximizing the chance that a solution exists down the subtrees we expand first. Once again, if there is a tie between values for rule 3), we can choose any of the tied ones.

Applied correctly and in order, rules 1), 2), and 3) can very significantly speed the search process. Notice that they ***do not compromise the completeness of the search***. We are not ***pruning*** any subtrees. We are only changing the order in which variables are expanded and values are assigned. Any solution that exists in the search tree for the naive backtracking method is also in the search tree for the enhanced search process using the three heuristic rules above.

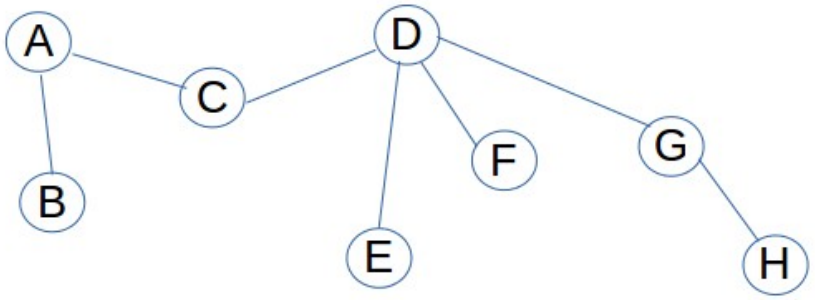
Complexity headaches

For a CSP with ***n*** variables, each of which has a domain of size ***d***, backtracking search has a complexity of $O(d^n)$. This should not be too surprising since in the worst case we would have to try every possible combination of values to be assigned to vadiables before discovering there is no solution to our problem.

Even without approaching this worst case, the search may become unmanageable for a large CSP. For problems which have a particular structure it may be possible to significantly reduce search complexity.

Tree structured CSPs

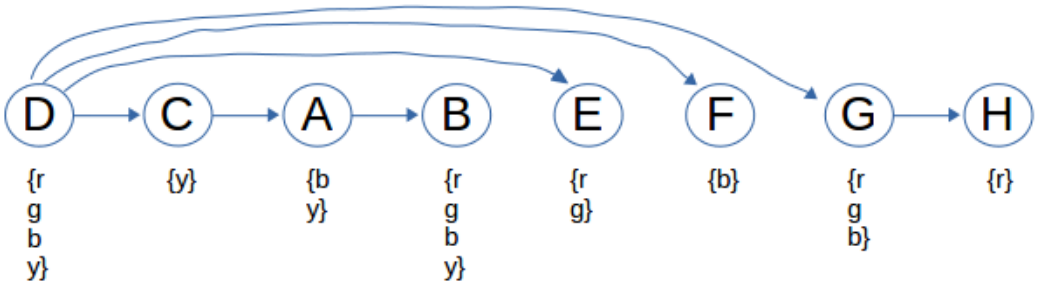
A CSP whose constraint graph has ***no loops*** is a tree-structured CSP. An example of such a CSP is shown below



(c) 2020 – F. Estrada

Tree structured CSPs can be solved efficiently, with complexity $O(nd^2)$ by following a 2-pass process.

1) Choose a ***root variable*** and perform a ***topological sort*** so the children of each node appear after their parent. Initialize the set of valid values for each variable in the CSP.

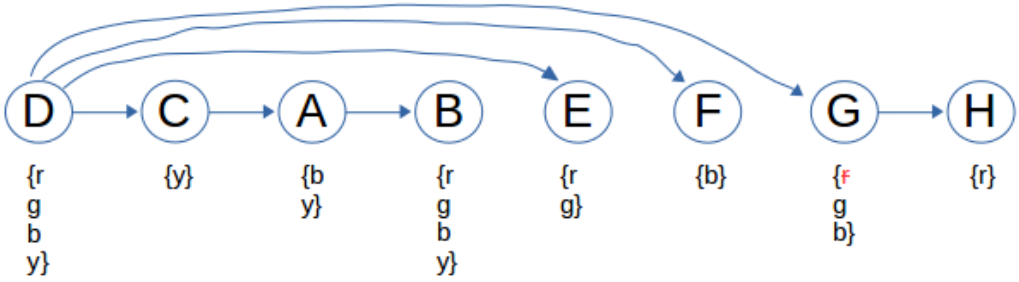


In the example above, we chose ***D*** as the root, but we could have chosen a different variable. Also, the initial sets of valid colours were selected ***as an example*** so that you can see how the process works and to make it clear that the initial sets depend on your problem and any initial constraints given.

Note that the graph we formed is ***directed*** making it explicit what are the parent-child relationships between nodes.

2) ***Backward pass*** – from right to left (H then G, F, E, B, A, C, and finally D). Check ***the parent of each node*** and ***remove any values that would make the constraints in the problem not valid***.

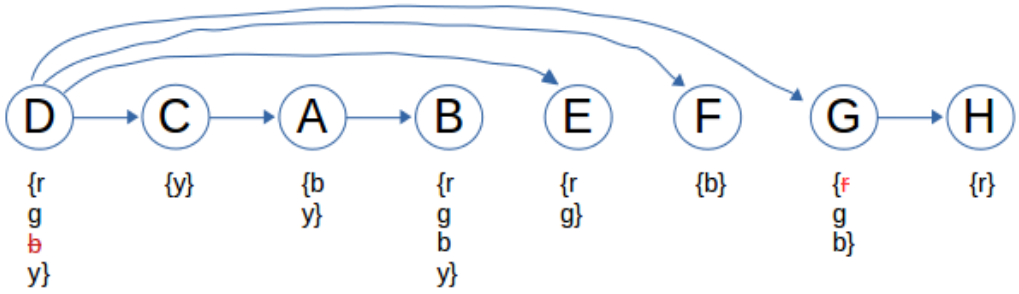
In the example above, we start with node ***H***. Note that ***H*** can only be assigned to ***red***. This means that its parent ***can not be assigned red***. We remove 'r' from ***H***'s parent ***G***.



Next we look at node ***G***, this node can be assigned to ***red or blue*** (we just removed red!). Its parent is node ***D***, which can take any colour. ***Do we need to remove any colours from D?*** Turns out that it doesn't matter which colour we choose for ***D***, there is always an option for ***G*** that doesn't break constraints, so no colours need to be removed from ***D*** at this point.

Next up is ***F***, this node can only be assigned to ***blue***, its parent is node ***D***, this means node ***D can not be blue***, and we remove 'b' from node ***D***'s list.

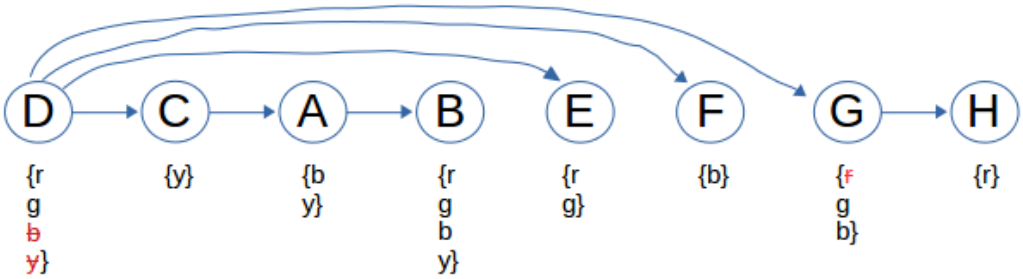
(c) 2020 – F. Estrada



Next is node ***E***, this node can be either ***blue*** or ***green***, any remaining choices for its parent ***D*** are fine since they leave at least one valid choice for node ***E***, so no need to remove any colours from ***D*** at this point.

Next up is node ***B*** which can take any colour, so no need to remove anything from its parent, node ***A***'s list. Node ***A*** can be ***blue or yellow***, its parent, node ***C*** is ok since choosing ***yellow*** for ***C*** leaves one valid choice for ***A***, no need to remove anything from ***C***'s list.

Finally, we look at node ***C*** which can only be ***yellow***, this means its parent, node ***D can not be yellow***, so we remove 'y' as a valid choice from ***D's*** list.



Note: After this backward pass, ***all remaining values for variables in the CSP are known to leave at least one valid choice for children nodes.***

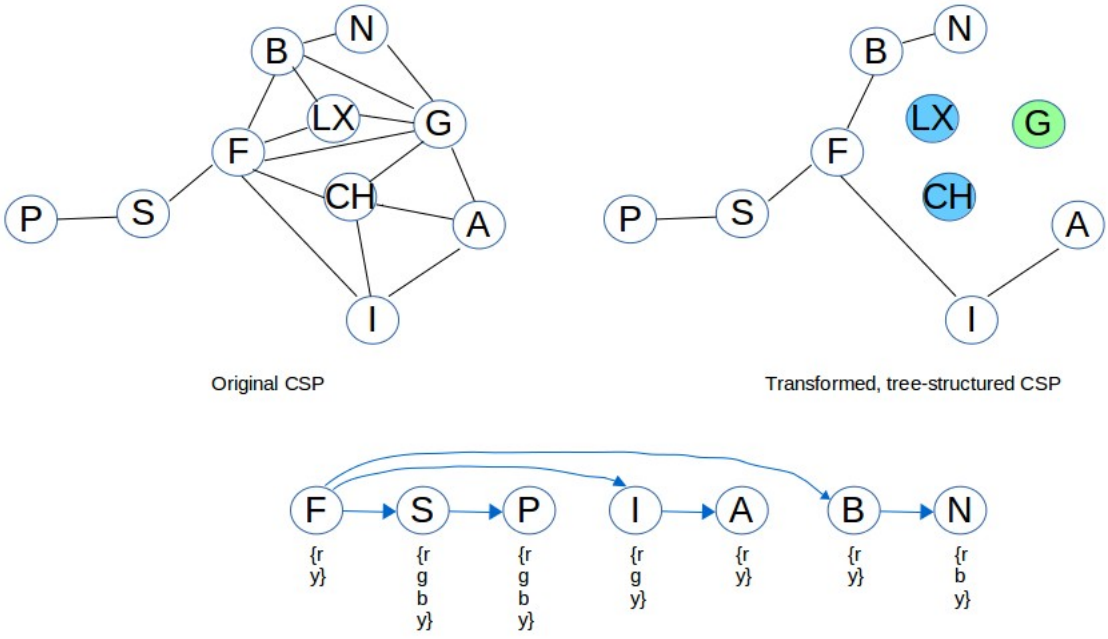
3) Backtracking search – starting at the root node (in the case above, ***D***) and proceeding to assign values in order from left to right (D, then C, A, B, E, F, G, and finally H). The search ***will not backtrack!*** This is because any value chosen for a parent node is consistent with constraints on their children.

Here's a question for you to think about: What does it mean if at some point while removing possible values from a parent node's list, that list becomes empty?

To make sure you got this – follow the same process as above, but starting with node ***A*** as root.

Cutset Conditioning

Of course, the process above is only useful to us if our CSP is tree-structured. So it may not seem that useful. However, consider the original map colouring problem we have been studying:



We have taken our original problem, and turned it into a tree-structured CSP by ***assigning values to a subset of variables*** so that we can ***disconnect them*** from the rest of the CSP. Notice that the resulting tree-structured CSP has variables whose initial sets of valid values have missing colours, corresponding to the constraints we removed on nodes whose colours we assigned (e.g. the Netherlands, ***N*** can not be ***green***, because we assigned this colour to its former neighbour ***G***).

We can take advantage now of the low complexity of solving the resulting tree-structured CSP. However, we assigned values arbitrarily to some variables and ***there's no guarantee*** that the initial assignments to these variables still allow for a solution to the CSP to be found!

In practice, we ***may have to try all possible combinations*** of these variables to discover the one that leads to a solution for the full CSP, therefore, the complexity of solving a CSP for which ***we fix the value of k variables to obtain a tree-structured CSP*** is:

$O(d^k \ (n-k)d^2)$

The d^k term accounts for all possible combinations of ***k*** variables whose values were initially set (in the example above, ***k=3***), the rest of the expression accounts for the complexity of solving a tree-structured CSP with ***(n-k)*** variables.

This will likely still result in a significant computational saving compared to the complexity of solving the original CSP with ***n*** variables. The procedure we outlined above is called ***cut-set conditioning***.

(c) 2020 – F. Estrada

Iterative Methods and Approximate Solutions

Sometimes, solving a CSP using backtracking search is either too costly, or we are interested in obtaining only a ***sufficiently good*** solution in a much faster timespan. Iterative methods can be used to find approximate solutions to CSPs (i.e. solutions in which some of the constraints are broken).

The simplest and most general of these approaches is called ***Local Search***.

The Local Search process is defined as follows:

```
Randomly generate a full assignment of values to variables in the CSP
its=0

while its<MAX_ITERATIONS

    randomly pick a variable that has conflicts
    randomly select a new value for this variable

    if the updated assignment is a solution: STOP

    if the updated assignment breaks fewer constraints than the older one
        keep updated assignment
    else
        keep old assignment instead

    its++
end
```

As you can see, the procedure itself is incredibly simple, and very easy to implement. It is ***local search*** because at each step only one variable changes value, so each successive guess at a solution is ***close in solution space*** to the previous one. We are looking around the ***neighbourhood of possible solutions*** from wherever we currently are for something that looks better.

The question is ***why should we expect this to do anything good?***

The key is the statement that keeps an updated assignment ***only when it breaks fewer constraints***. The randomness of the process means many of the random changes we make won't improve things, but some will, and those are kept. Over time the solution should get better than whatever we started with.

This kind of process is called ***hill climbing***. We start somewhere in solution space, and run a process that consistently makes our solution better. The process is so simple that we can run millions and millions of rounds of local search in a very short time. So improving on an initial guess is fast.

Of course, if solving CSPs was that easy we wouldn't need backtracking search. The problem with local search is that it frequently gets stuck: The current solution can not be improved by changing the value of a single variable to any value within its domain. The process has arrived at a ***local maximum*** within the solution space for the problem, and there's nowhere to go.

This will happen, it's very unlikely we will get so lucky with our initial guess that the local search process will arrive at the global optimal solution. The more complex the CSP (more variables, larger domains), the less likely it is that we'll find even a very good solution. So local search has to be used with the understanding that we can find an assignment, improve on it, but we can't expect it to be a great solution most of the time.

Example of local search

To make the above ideas concrete, let's look at a simple CSP that is often used to illustrate how local search works.

The N-Queens problem is a thought problem in Chess, it consists of placing N queens in an NxN chess board in such a way that none of them can attack each other. In terms of constraints, this means:

- No two queens can be on the same row
- No two queens can be on the same column
- No two queens can be on the same diagonal

We can solve this problem as a typical CSP and use backtracking search, but here let's see how the solution process would work:

```
Initialize an NxN board with N queens each in a different column
its=0
while its<MAX_ITERATIONS

    choose one queen at random
    move the queen to a different location in its own column

    if the new assignment is a solution: STOP

    if the new assignment has fewer conflicts
        keep new assignment
    else
        keep old assignment

    its++
end
```

Notice that here we are being a bit clever, and we resolve one source of conflict by assigning each queen to its own column, and never moving them anywhere other than along that specific column. This means conflicts can only come from queens being in the same row, or the same diagonal. Each step of the search moves one of the queens along its column, and keeps the new location if it reduces the total number of conflicts for the whole board.

The above process is enough to solve small boards with some luck (try it out, see how many times you find a solution for a given N, with N=4, 8, 16, or 32). Even for a very small N, you'll soon see that the process gets stuck.

Faster Hill Climbing

Let's be a bit more clever about how we choose an updated assignment:

```
Initialize an NxN board with N queens each in a different column
its=0
while its<MAX_ITERATIONS

    choose one queen at random
    move the queen to the location in its column that minimizes conflicts

    if the new assignment is a solution: STOP

    if the new assignment has fewer conflicts
        keep new assignment
    else
        keep old assignment

    its++
end
```

Notice that we changed the step that randomly moves a queen to a new location so that it now looks at all possible location in a column and chooses the one that minimizes conflicts. This will produce a better assignment much more quickly at the expense of having to do more computation.

Let's do try one more thing:

```
Initialize an NxN board with N queens each in a different column
its=0
while its<MAX_ITERATIONS

    choose the queen with the most conflicts
    move the queen to the location in its column that minimizes conflicts

    if the new assignment is a solution: STOP

    if the new assignment has fewer conflicts
        keep new assignment
    else
        keep old assignment

    its++
end
```

Now we changed the way in which we choose which queen to move so that it's not random, instead, we're going after whatever queen currently is involved in the largest number of conflicts, and move that one.

The point to make here is ***not*** that this is ***the right way*** to solve N-Queens using local search! (there is no such thing). The point is that ***given a specific problem you are solving***, you can and should be a bit clever about how you choose variables to update, and values to try, and different choices will cause your local search to climb faster or slower to a local maximum (or even change which local maximum is going to be reached! - this is a very hilly, multi-dimensional space we're in).

(c) 2020 – F. Estrada

So keep in mind: Local search is a simple and efficient way to improve on an initial guess for a CSP, you can be smart about how it works, and it will produce a solution better than what you started with. Sometimes that is enough.

Improvements and extensions to local search

There are many extensions and improvements that have been proposed to improve the convergence of local search, and to improve the chance it will arrive at a better solution. We won't cover these in detail but you can dig into any of them if you are interested. It's worth knowing what some of them are so you can look into them if you need to use local search in the future:

a) You can run the local-search process K times, and keep the best solution

b) Deterministic Annealing, it is a variation of local search that allows the process to keep a solution even if it is worse than the current one ***with some probability***. This probability decreases over time. The goal is to allow the search to become ***unstuck*** by allowing a move to a worse solution that may eventually lead to a better local maximum.

c) Beam search/Taboo search, keep track of multiple guesses found in the search process and attempt to maximize the chance of finding a good solution.

d) Genetic algorithms / Ant colony optimization – which are variations on local search methods

Conclusion:

CSPs are a wide class of fairly important problems in AI, and you now have tools to solve a large class of them using backtracking search, or to find approximate solutions using iterative methods. Remember that these algorithms are still within the domain of ***search problems*** that we started with.

(c) 2020 – F. Estrada