

**What we have left with**

**Lecture 5**

**Constraint satisfaction (part 2)**

In the last lecture we see that CSP can be solved by standard DFS, but we can improve it by modifying the algorithm to backtrack early.

This type of problems cannot be solved using the approach of heuristic function we have learnt in the last lecture, and we will see other ways to deal with them.

**Reference:**

- Section 4.2 (pp. 104–105), 4.4.

AI(0270)

- At many times, we can use **backtracking search** and **forward checking** to know early that a whole subtree won't have a goal state.
- We can use **constraint propagation** to assign variables to some nodes early, to further reduce the need of back-tracking.
- The ordering of variable assignments is **not important in the solution**.
- The ordering of variable assignments is **important in reducing search cost**: a good ordering significantly reduce the amount of backtracking.
- So what ordering is good? The principles:
  - If we have to **make a harsh decision** sooner or later, make it **early**. So that if it fails, we fail early.
  - When we decide, **first try the choice less harsh to others**. Giving more room to other decisions, hoping that we can find a solution early and can stop the search altogether.

AI(0270)-5.1

**The two orders to make**

**Which variable** to try now? Two choices:

- **Most constrained variable**: Choose the variable that has fewest value to choose from.
- **Most constraining variable**: Choose the variable that is involved in the most constraints of unassigned variables. In practice, they work similarly, making it possible to solve 100-queen problems.

Further improvement: **Which value** to try first?  
Consider this after we have chosen a variable

- **Least constraining value**: Choose a value that rules out the least number of choices for other variables. Make 1000-queen feasible!

Why completely opposite strategies?  
Think of the basic principle on the last page!

AI(0270)-5.2

**Example: coloring again**

Let's color maps that are a bit more difficult. Suppose we have 4 colors.

We will use everything we have learnt (**back-tracking, forward-checking, constraint propagation**), except that:

- The first time we use **most-constrained-variable** to choose variables. This actually makes it unnecessary to use back-tracking and constraint propagation: they are implied by most-constrained variable.
- Then we use another map with **most-constraining-variable** together with **least-constraining-value** instead.

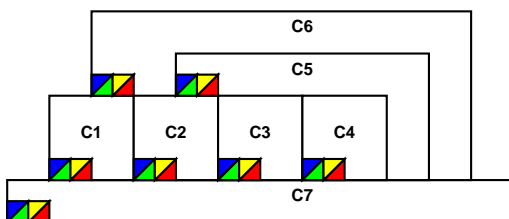
Keep in mind that although we need no backtrack in these examples, **we might need to backtrack in practice**. And in some case, **the heuristics can lead us away from the solution**.

But this is expected: NP-completeness means that all known algorithms needs to spend exponential time in the worst case.

How good? E.g., Can solve thousand-queen problems.

AI(0270)-5.3

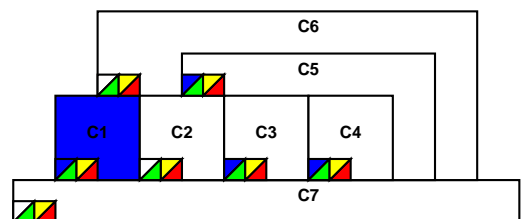
**Example: most-constrained-variable (1)**



Initially, all colors (values) are available for all regions (variables), so all regions are "most constrained". Let's break ties by choosing the region with smallest number, C1.

AI(0270)-5.4

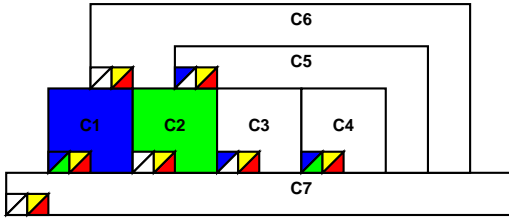
**Example: most-constrained-variable (2)**



Now C2, C6, C7 have 3 choices while all others have 4. So choose one of them, say C2.

AI(0270)-5.5

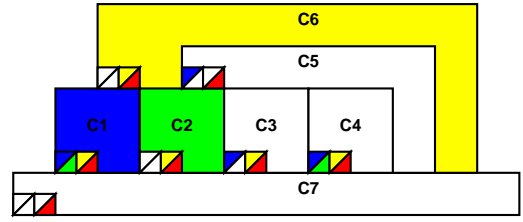
Example: most-constrained-variable (3)



Both C6 and C7 now only have two choices, so we choose one of them, say C6. Note that just using constraint propagation will miss the chance to do constraint propagation and get a solution immediately (because it would choose to color C3).

AI(0270)-5.6

Example: most-constrained-variable (4)



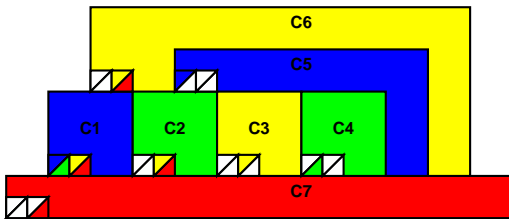
Now C7 has only one color remaining, and is the most constrained variable. Note how the **most constrained variable strategy is a generalization of constraint propagation**.

Much of the code will be the same as constraint propagation, making it particularly easy to implement.

From now on Constraint propagation will take us to the final solution.

AI(0270)-5.7

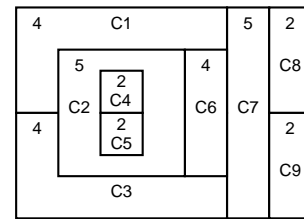
Example: most-constrained-variable (8)



Final coloring. Note that no backtracking is needed at all.

AI(0270)-5.8

Example: most-constraining-variable (1)



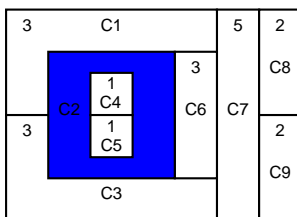
C2 and C7 are most constraining: they constrain the choices of 5 other regions. Let's again choose the one with the smaller number.

Any color (value) is equally constraining.

In this problem, it is very easy to update the conflict counts: **when coloring a region, the count of all adjacent regions are decremented**.

AI(0270)-5.9

Example: most-constraining-variable (2)



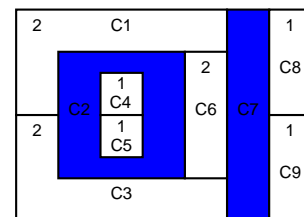
Now C7 is most constraining, still constraining the choices of 5 other regions.

There are three possible colors. Choosing blue does not further restrict C1, C6 and C7, so it is the least constraining choice.

Indeed, it is the only choice that will lead to a good coloring. Any other color will make it impossible to color one of C1, C3 and C6.

AI(0270)-5.10

Example: most-constraining-variable (3)

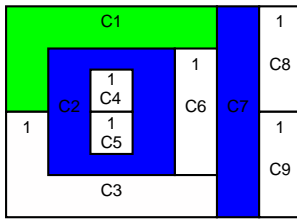


C1, C3 and C6 are equally constraining. We choose C1.

The three remaining colors are equally constraining. Let's choose green.

AI(0270)-5.11

#### Example: most-constraining-variable (4)

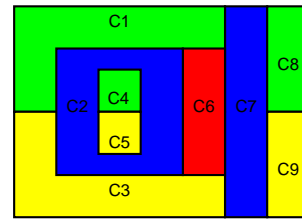


Once we get here, all the remainder are trivial. The program should continue to color each cell, breaking ties.

This illustrates one property of most-constraining variable: it tries to reduce future branching factor as quickly as possible.

AI(0270)-5.12

#### Example: most-constraining-variable (5)



Let's skip the boring tie breaking and just see the final coloring.

AI(0270)-5.13

#### A completely different direction...

There is a completely different approach to solve the problem: **to start with a wrong answer and patch it until it becomes correct.**

Let's see an example: the 8-queen problem

- **States:** 8 numbers from 1 to 8 specifying the location of the queen in each column.  
Not allowing unassigned queen location!
- **Operators:** to modify one of the 8 numbers to a new value.
- **Initial state:** arbitrary. In general we will create a random initial state.
- **Goal state:** a state in which all queens do not attack any other queen.
- **Path cost:** inverse of number of pairs of queens that conflict.

This approach is also applicable in other problems in which **exact path does not matter.**

AI(0270)-5.14

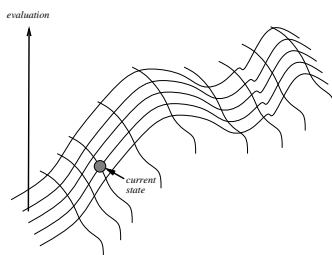
#### Properties

- Note that the SS becomes a general graph rather than a tree, so it is not immediately clear why the formulation is useful.
- But note that the **search landscape** is now completely different: any state is a solution, but **we want the "best" solution.**  
Just like a teacher who, rather than just telling their students whether their answers are right or wrong, always say the answer is right—to a certain degree. What we now want is the best answer.
- A solution with **less conflict** is considered "**closer**" to the solution, so our target is to move from the random starting position to an optimal solution step-by-step.
- The **path-cost** (or "goodness") of the current state **guide us to move towards the goal.**  
Just like any heuristics, at times they fool us away from the goal as well.
- Such algorithms are called **Iterative Improvement algorithms.**  
Applied on CSP, called heuristic repair.

AI(0270)-5.15

#### Or, a graphical view

You can **visualize iterative improvement** by a picture like this:



In other words, we **look around the landscape** and find a promising way to **reach a place with highest evaluation function.**

What is the x and y axis? In reality, there is no x or y axis. It is just an illustration trying to visually give you idea about what we are doing.

AI(0270)-5.16

#### Hill climbing

- Repeatedly apply the operator that **increases the evaluation function** by the largest amount.  
"Finding the top of Mount Everest in a thick fog while suffering amnesia"?
- E.g., in 8-Queen, consider all one-step modifications of the queens and select the one which reduce the number of conflicts the most.

Two major problems:

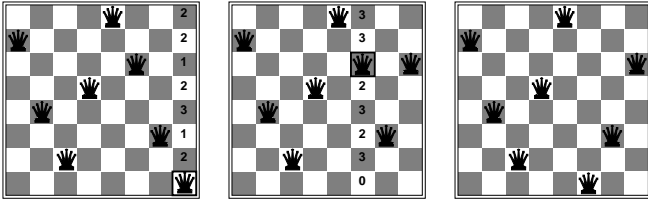
- **Local maxima:** the search may get stuck at a place where the evaluation function is **largest in the short range**, but is not the global optimal.
- **Plateau:** if we have a large area where the **evaluation function is essentially the same**, the search becomes a *random walk*.

Hill climbing is more effective if the landscape is smooth and leading towards a peak with not too many local maximum.

Most realistic difficult problem does have a lot of local maximum, though.

AI(0270)-5.17

### Example: 8-Queen



Note: This example use one variable for each column, not row.

In reality, we will have to consider all the possible moves of all queens that are in conflict.

AI(0270)-5.18

### Random restart

- So the algorithm repeat the steps **until the evaluating function is no longer increasing**, and stop there.
- What to do then? **We might be in a local maxima, plateau or ridge...**
- One possibility: **restart from another random initial position!**
- We will **remember the best solution so far**. After a certain number of tries, we **report the best solution we recorded**.
- This is called **Hill climbing with random restart**.
- Very effective on certain CSPs. E.g., solving million-queen problem is feasible.
- Of course, if enough tries are made, we will eventually be "lucky enough" to hit somewhere that leads to the optimal solution.

AI(0270)-5.20

### Hill climbing in continuous environments

Hill climbing is about the only method for **continuous environment**.

- Repeatedly choose **the steepest increasing direction** of the evaluation function, by some predefined **step length**.

We will need a step length, i.e., **how far we go each time**, because the environment is continuous and we don't have the "smallest step".

But that gives us another problem:

- **Ridges**: very steep on both sides, so we always overshoot rather than getting to the top of the ridge.

AI(0270)-5.19

### Simulated annealing

- For other problems, **forgetting everything done so far is too expensive** to deal with small local maxima.
- An alternative: **just choose a random next-state**, and then **move to it** with a **probability depending on how much it worsen the solution**.
- What is the probability? We will want to make it **larger at the beginning of the search** and **slowly reduce the probability to 0** later.
- So two parameters:  $\Delta E$ : how large is the downhill move, and  $T$ , the "temperature". **High temperature means easy downhill move**.
- The probability is usually  $e^{\Delta E/T}$ .
- We want **high temperature** at the beginning, and **0 temperature** at the end: **simulated annealing**.  
Anneal: To subject, e.g., glass, metal, etc., to great heat, and then cool slowly, for the purpose of rendering it less brittle, to temper, to toughen.

AI(0270)-5.21