

**Why A\* search doesn't solve all search problems?**

The heuristic function in the last lecture has an **important assumption**:

We want to look for an optimal solution, in the sense that the optimal solution means the solution with the **minimum path cost**.

For many state-space, this leaves no room for informed search:

- 8-queen problem...
- Crypto-arithmetic...
- map coloring problem... (Given a geographic map and a number of colors, make sure that two adjacent regions won't share the same color.)

For these algorithm, **we cannot express our domain knowledge as a heuristic function**, since "adding up path costs" makes no sense.

For all the above problems, the path cost is always 0, so the poor 0-function is the only reasonable heuristic.

AI(0270)

AI(0270)-4.1

**Lecture 4**

**Constraint satisfaction**

In this chapter we examine another type of state-space search problems, namely constraint satisfaction.

This type of problems cannot be solved using the approach (heuristic function) we have learnt in the last lecture.

In this lecture and the next, we will see other ways to deal with them.

**Reference:**

- Section 3.7, 4.2 (pp. 104–105), 4.4.

**Constraint Satisfaction Problem (CSP)**

Many such problems have the following form, and are called CSP:

- There are a number of **variables**
- Each variable has **some values** that it can take.
- Our target is to find some way to set all the variables so that **a set of constraints** about the variables are all satisfied.

In our standard state space form:

- A **state** is any **partial assignments** to the variables.
- **Operator**: **adding an assignment** to one of the variables.
- **Initial state**: **no assignment** is made. **Goal state**: any state with **all constraints satisfied**.
- All goal states are **equally good**. In other words, path cost is the same for all goal states.

AI(0270)-4.2

**Example**

All the problems we have just seen are CSP

- **8-queen problem**: we can have **one variable per row**, each may take a values from 1 to 8 (the column number of the queen in that row).

Constraint: all variables are different; no two queens are on the same diagonal.

- **Crypto-arithmetic**: each letter is a variable, with values from 0 to 9.

Constraint: all variables are different; the resulting formula is correct.

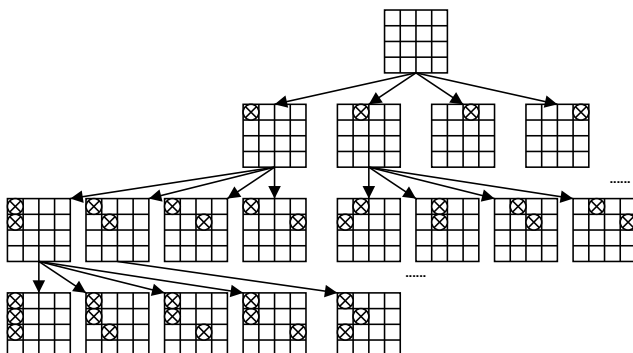
- **Map coloring**: each region is a variable, the value being one of the available colors.

Constraint: variables corresponding to adjacent regions are given different values.

AI(0270)-4.3

**State space**

Let's see the state space of one typical (but small) problem: 4-queen.



AI(0270)-4.4

**Observations**

First, some very easy things:

- **In each layer, we just need to try one variable.**

The order doesn't matter, so we can always use the ordering we like. We won't miss a solution just because we try variables in the wrong order.

- **Doing it depth-first is better than doing it breadth-first.**

The **depth of all leaf nodes are the same**, so the incompleteness of DFS is not a matter at all.

All **solutions are equally good**, so the sub-optimality of DFS doesn't matter, either.

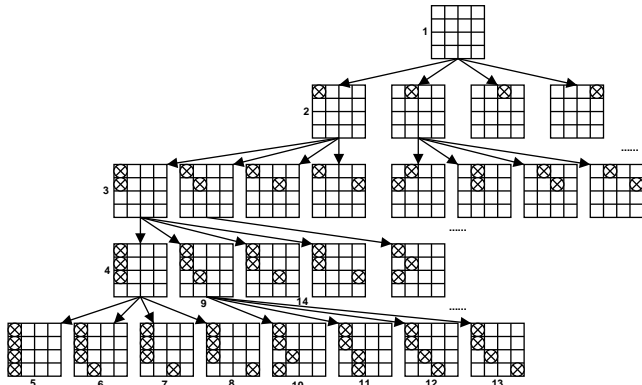
But Using DFS would save a lot of memory!

No need to do IDFS: why?

- **The search space is a tree**, no cycles (so no need to deal with them).

AI(0270)-4.5

### Doing DFS: example



The numbers are the order in which DFS try to expand the state.

AI(0270)-4.6

### Doing better: State-checking

- But wait... it seems that **we are guaranteed not to find a solution for a long time!**
- Why? We are guaranteed not to find any solution in the **whole subtree** of the node marked "3"...  
And also the node just beside it.
- Why? Because we **already have some constraints failed there.**
- Note that sometimes, we can know that **the whole sub-tree of a search tree cannot give rise to any goal state.**
- If we can effectively **detect such states and not to search in the whole subtree**, then we can search much more efficiently.  
This is speed trade-off: it will takes several times longer to examine a node. But if we can cut down half the subtrees, then effectively the branching factor is halved.
- But what kind of failures of constraints can we find?

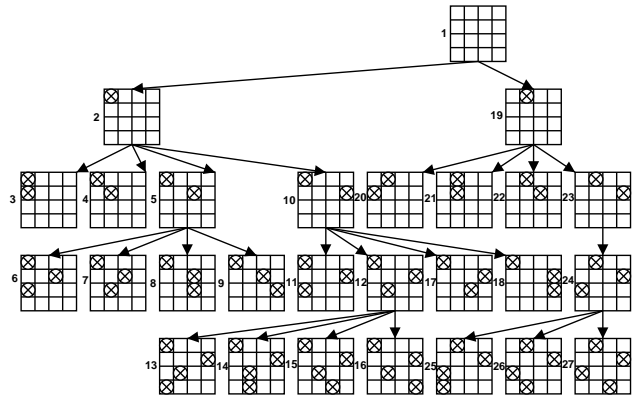
AI(0270)-4.7

### Checking for failed constraints: backtracking search

- One of the constraints of 8-queen is that **no piece should be in the same column or diagonal.**
  - The constraint fails, because the two already placed pieces are in the same column.
  - Such failures **cannot be corrected by giving values to more variables**, so it is really point-less to continue the search.
  - We call this **backtracking search**. The general form:  
We check whether the constraints are satisfied for those variables that are already given values.
- The name is really strange: we should back-track if we know a state can't have goal state in the subtree for any reason. Perhaps it is to contrast with "constraint propagation" that we will see soon.
- Implementation? Just add checking before we put a state into a queue.
  - How good? Feasible to solve n-queen for  $n \approx 15$  now.

AI(0270)-4.8

### Example



And this is about the best we can do in this case.

AI(0270)-4.9

### Trivial impossibility to assign variables: Forward checking

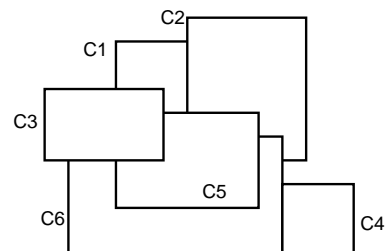
- If we look at the states expanded in step 5, we will understand that we are really lucky.
- We expand states 6, 7, 8 and 9 from state 5, all conflicting with something we already assigned a value.
  - In other words, **there is no way to assign a value to the third row variable** once we make the move on step 5.
  - We are lucky because **this happens to be the next variable we try**. If this is not, we would have to try something that is already impossible.
  - So comes the idea for another way to improve our algorithm: we **check whether any variable has run out of choices** that satisfy constraints.
  - This is called **forward checking**.
  - How good? Now feasible to solve n-queen for  $n \approx 20$ .

AI(0270)-4.10

### Example

Now that the 4-queen problem is already solved in the best way, we use another problem: **coloring**.

Suppose we want to **color this map using Red, Green and Blue**:



By the way, 3-coloring a planar graph is NP-hard.

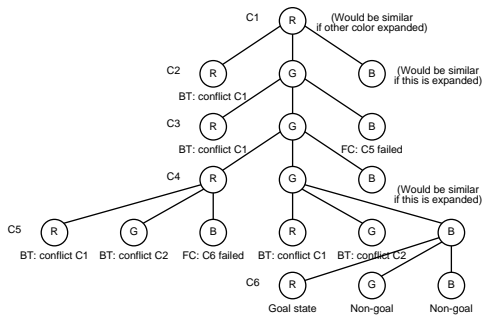
AI(0270)-4.11

### Let's do DFS on it with Backtracking and Forward-Checking

This is what we get.

"BT" denotes nodes removed by back-tracking.

"FC" denotes nodes removed by forward checking.



Note that we make some errors, but **they are found real early**.

But... **do we really need to make those errors?**

AI(0270)-4.12

### Constraint propagation: reduce backtracking by early assignment

- In Forward-Checking, we **stop** if the number of available choices for a variable becomes 0.
- But in fact, **if the number of choices for a variable becomes 1, the assignment for that variable is fixed.**
- If we really make that assignment early, **other variables might have only one choice remaining as well**, and can be given a value.
- This is called **Constraint propagation**.  
A constraint propagate and causes other variables to become constrained too.
- E.g.: after we assign C1 to Red and C2 to Green, C5 is forced to be Blue. Then C3 is forced to be Green, C6 is forced to be Red. Either choice of C4 gives a correct coloring.
- So for this particular case: **no backtracking is needed at all!**
- In general: need some backtracking, but reduced a lot.

AI(0270)-4.13

### Generalizing constraint propagation

- Constraint propagation basically says to **shuffle the order to assign variables**, if a new order is clearly better.
- This works because order is not important: **we can choose variables in any order we like**, and **different subtree may take different orderings**.
- But also notice: the efficiency of constraint propagation **depends on the initial choice of variables**.  
E.g., if initially we choose the variables C1 and C4, then quite a few more backtracking must be done.
- So we have another way of making the search terminates faster: **choose a promising ordering!**
- We will see a couple of things we can do in this line in the next lecture, and then a completely different way to solve CSP.

AI(0270)-4.14