Administration

- Next lecture will be on Monday, 10/15
- If next week we will have a midterm review, the midterm will be on the week of October 22nd, otherwise the midterm will be on the week after. It will be open for a week, PDF for download; do not spend more than 3 hours on it.
- Question 4 in assignment 1: we can make some assumptions and state them in our solution.

Warm-up exercise

- $g(n)$: the path cost function
- $h(n)$: the heuristic for the rest of the path
- $f(n)$: the sum of the two; the lowest f-value is chosen to be expanded.

If the heuristic is admissible, we are guaranteed that $f(n) \leq g(n)$, since $h(n)$ never overestimates the true cost. Therefore when a goal node is added to the fringe, it still needs to be expanded so we can calculate the REAL path cost.

The triangle inequality of consistent heuristics means that we can't have negative weights.

Consistent heuristic guarantees optimal solution.

Today's lecture

- Local search
- Constraint reasoning

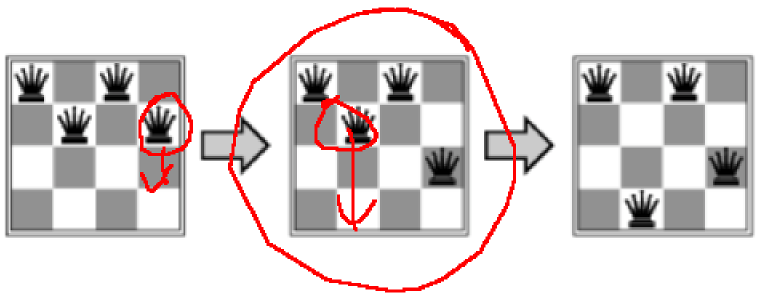
Local Search

Everything we talked about so far was around path, or sequence of actions. In those types of problems we know the start and goal state and want to discover what's in between.

In many problems we know the start state but we don't know the goal state, only criteria for it/them. For instance: **map coloring** – a map of the US, where we want to assign a color per state such that no 2 adjacent states have the same color. Here we don't care about the path, order of coloring, we just want the goal state. The state space is the set of all possible coloring for the map. Another problem: **n-queens problem** – find a configuration that satisfies some constraints.

<u>n-queens problem</u>:

Given an NxN chess board we want to place N queens on the board such that no queen threatens another, i.e. no queen is on the same row / column / diagonal with another queen.



Hill-Climbing Search:

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

When we get to a state in which we can't make a step that gets us to a higher solution, we stop. But, then we can get stuck in a local maximum, meaning we may miss the optimal solution.

In some cases that's ok, for instance if we know the start state is on a gradient up to the true maximum. But, in general that's hard / impossible.

Applied to the 8-queen problem:

Let h be a heuristic to measure how good a state is; we can use: how many pairs of queens are attacking each other.

Simulated annealing search:

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling the probability of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability $e^{\Delta E/T}$
```

This algorithm is used in order to escape local maxima. It gives a way of "jumping" from maximum to another hill.

Schedule is a mapping from time to temperature. If the temperature is 0, nothing changes, otherwise we randomly select the successor and get the delta between the current value and heuristic value. If that delta is positive, we advance to that successor state. Otherwise we take it in a probability that is some function of the temperature.

Properties:

- If T decreases slowly enough, the simulated annealing search will find a global optimum with probability approaching 1.
- This approach is used in many real-world applications like visual circuits design: in designing a CPU, determining transistor layout with constraints of connections.

As the temperature cools down, it becomes similar to hill-climbing search.

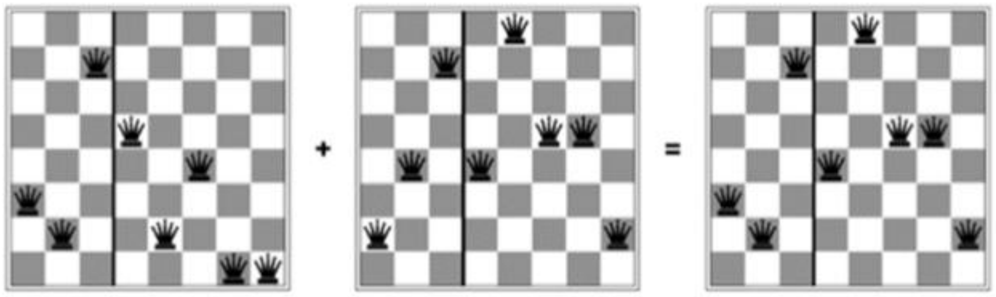
Local beam search:

- Keep track on k states
- Start with k randomly generated states.
- Each iteration, all the successors of all k states are generated
- Stops if anyone is a goal state.

Genetic Algorithms:

- A successor state is generated by combining two parent states.
- We start with k randomly generated states.

For instance, in map-coloring we can start with a string representation of some coloring of the states. For the next iteration, (generation) we use a fitness function to evaluate the colors. In N queens:



The left side of the left board and the right side of the right board are combined.

Genetic algorithms allow us almost instantaneously jump from one hill to another. At any single point in time we have k points we look at. We have the breadth of local beam search and randomness that allows jumping between local maxima.

Constraint Reasoning

Constraint satisfaction problem (CSP):

A way of modeling certain types of problems, such that if they are represented in this type of format, there are generic algorithms we can apply on them.

- Variables $\{v_1, v_2, \dots\}$ and domains $\{d_1, d_2, \dots\}$ - the domains are possible values that can be assigned to the corresponding variables.
- Constraints: set of allowed value pairs

<u>Example: 3SAT</u>:

- $\{x_i\}$ are variables
- $\{true, false\}$ are the domain
- The constraints are expressed in CNF: $(x_1 \vee x_2 \vee x_3)$ …
- We will need to know how to model a problem as a CSP – what are the variables, constraints and domains

<u>Exercise: 4-queens</u>:

There are multiple solutions. One of them:

- Variables: the board cells $\{x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, \dots, x_{44}\}$
- Domain: $\{Q, N\}$ – queen or no queen
- Constraints: $x_{11} \wedge \neg x_{12}$ … - location constraints, and $\sum x_{ij} = 4$

This gives us n^2 binary variables and a lot of constraints.

Another way:

- Variables: queens $\{q_1, q_2, q_3, q_4\}$
- Values: board coordinates $\{(1,1), \dots, (n,n) = (4,4)\}$
- Constraints:
  - No two queens are on the same space on the same time
  - No-threats constraints (on the locations of the queens)

<u>No solution / multiple solutions</u>:

- On solution - minimize number of broken constraints
- Multiple solutions – maybe some solution is preferable over another

$\Rightarrow$

<u>Constraint Optimization</u>:

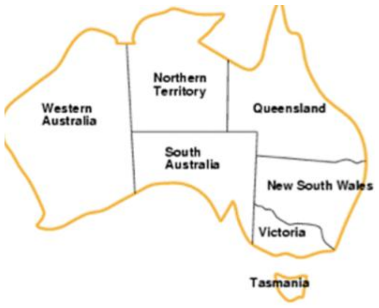
- Constraints have degree of violation
- We want to minimize violation

<u>Constraint optimization problem (COP)</u>**:**

For each pair of variables that constrain each other x_i, x_j we have a mapping $f_{ij}: D_i \times D_j \to \mathbb{N}$

And we want to minimize this constraint function.

Example: Australia map coloring:



- Variables: the states
- Domains: $D_i = \{red, green, blue\}$
- Constraints: adjacent regions must be colored differently

Optimization: if $WA = NT$ the cost is 1, otherwise it's 0

Then we simply want to minimize the sum of the coloring cost

Constraint graph:

That's a graph in which the variables are nodes and the edges are between two variables that constrain each other.

There could be complex constraints that cannot be represented as a simple constraint graph.

CSP as search:

- Initial state: empty assignment {}
- Successor function: assign vale to an unassigned variable
- Goal state: complete legal assignment
- Path cost: constant for each step

Backtracking search – basically DFS:

We start with an empty state, like empty coloring

Successors: state x is red, state x is blue, state x is green

We continue down some path until a leaf. If it's illegal, we continue to another branch like DFS.

Improving efficiency:

There are different heuristics for how to choose the next successor to expand, e.g. what state to color next.

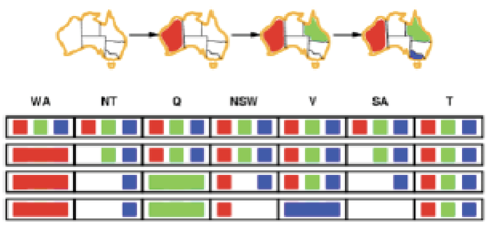
M**ost constrained variable** heuristic: we chose the most constrained variable. For instance, after an initial coloring, immediately one of the two adjacent states are the most constrained, and one of them will be chosen. This is also known as the **minimum remaining value (MRV)** heuristic.

Most constraining variable: look for the variable with the highest degree in the graph – the one that puts the maximum number of constraints out of all the variables.

Least constraining variable: given a variable, choose the value that constrains the least. For instance, choosing red for one state may leave 1 option for an adjacent state, and blue – will leave 0 options. So we will choose the first.

Forward checking:

We keep track of remaining legal values for the unassigned variables, and we terminate the search when any variable has no legal values. We propagate the decisions forward. For instance, a representation of the remaining possibilities for coloring:



In the example above the empty value for SA means there is no legal domain.

Constraint propagation: we propagate forward more than one step at-a-time.

Arc-consistency

A simple form of propagation makes each arc consistent: X and Y are consistent if for **every** value of X, there is some allowed value of Y. When X loses a value, the neighbors of X are rechecked.

Local search for CSPs:

In local search we usually we model it such that all variables are assigned, and the successors would be switching assignment.

Example: 4 queens:

- States: 4 queens in 4 columns, 256 states
- Actions: move queen in column (one queen at a time)
- Goal test: no attacks
- Evaluation: $h(n)$ is the number of attacks.

Modeling as a CSP: skipped…

Distributed Optimization Problem

That's basically a CSP where each agent has control on some set of the variables.

Each agent knows only about its own variables. Problem: for instance, if two non-adjacent states belong to the same agent, the agent may color them such that the global solution is not legal.

There's a need for negotiation between the agents to achieve a global solution.