

1 Administrivia

We've had a number of people drop the class since the initial reading groups were assigned (obviously). I will reallocate the reading groups and post new group assignments shortly.

2 Constraint Satisfaction

- So we've looked at a number of different ways to search complex spaces, including using domain-specific heuristics. The trouble is that it can be hard to figure out useful heuristics (as you may have discovered in the homework).
- But what about *general* purpose heuristics? Methods that can be applied to a wide number of different problems and be helpful to all of them?
- This desire led to the class of **constraint satisfaction** methods, developed starting in the early-1980's.
- The idea is roughly that if you can represent a problem in terms of what constraints the solution (goal) needs to obey, then you can operate directly on the constraints and devise heuristics that work in terms of constraints.
- We've seen this already in LP; today we'll discuss some methods that are intended largely for integer problems.
- (Recall that integer programming is known to be NP-hard; the methods we'll discuss today fall into the same class, so you can't expect a *perfect* solution or *ideal* heuristics, but you can hope for substantial improvements in the general case.)
- The key to CSP (constraint satisfaction problem) methods is representing the problem in a uniform way.
- Example: Map 3-coloring: The country of Australia has seven major administrative divisions (states and territories): Western Australia (WA), South Australia (SA), the Northern Territory (NT), Queensland (Q), New South Wales (NSW), Victoria (V), and Tasmania (T). (I'm ignoring the Capital Territory, which is the equivalent of our Washington DC.) Our problem is to color a map of the continent with only 3 colors (red, green, blue) such that no two adjacent divisions are the same color.
- In general, this is the *graph 3-coloring problem*, and it has practical applications in cryptography, compilers, and bioinformatics. It's also NP-hard.
- One way to formulate this is as a set of variables, each of which can take on one of the values {red, green, blue}.

- In addition, we need the *constraints* that specify that adjacent things can't be given the same value:

$$\begin{aligned}
 WA &\neq NT \\
 WA &\neq SA \\
 NT &\neq SA \\
 &\vdots \\
 &\text{etc.}
 \end{aligned}$$

- This should look somewhat familiar. Like LP, we're defining a problem in terms of constraints, though now they're binary integer (in)equality constraints rather than the real-valued linear inequalities.
- We *aren't* giving an optimization criterion in this problem, we're just looking for *any* solution in the feasible region. (There are CSP formulations that include cost, but we're not going to consider them now.)
- Note that we can write down a graph that describes the constraints. (This is why this is called the *graph 3-coloring* problem.)
- Q: how many binary constraints are possible for a system with n variables?
- Now we can formulate the problem as a search problem. Start state: the empty set. Successor function: assign a value to some variable that doesn't violate any of the constraints.
- We could now apply *any* of the search methods we've learned to attack this problem. So far, we haven't done much that's different from those.
- But there are a couple of important points to note about this problem:
 1. The paths from the goal are always a fixed length. We don't have a "retract variable" action in our list, so we can only ever add variables to the solution. Since there're a fixed number of variables (7), we can only ever make 7 moves from the start. (Q: how many paths are possible?)
 2. We don't care how we *get to* the final solution – we only care that we find it. Path is irrelevant.
- These conditions should remind you of something you've seen recently.
- How would you frame the 8-queens problem as a CSP?
- Note that the kinds of constraints you come up with here are *not* binary – they're "all different" constraints.
- Any "all different" constraint can be reduced to a set of (n^2) binary inequality constraints, so we'll just talk about binary constraints for the moment.

- The important point is that we now have a uniform way to write down superficially very different problems. Specifically, a CSP is defined in terms of:
 1. A set of variables, $v_1 \dots v_n$
 2. A set of possible values for each variable, $r_{11} \dots r_{nd}$ (i.e., each variable can take on one of d values). This is called the *domain* of each variable.
 3. A set of (binary) constraints of the form $v_i \neq v_j$ or $v_i = v_j$ (in general, some set of constraints in a fixed **constraint language**)
- So let's look at a simple *search* algorithm and examine the effects of different *heuristics*
- Simplest search: backtracking search: pick and assign vars in some order until either you find a goal or you reach a point at which you can't assign any more vars. This point is called a **conflict** point. If you hit a conflict, backtrack and change an earlier variable assignment.
- (I.e., just depth-first search.)
- The kinds of heuristics we're using today are slightly different than the kinds we've used previously – rather than simply assigning a value to a state, they're going to influence the search more directly, by adjusting the variables that can be assigned, etc. On the other hand, they *won't* depend on a specific problem – they're generic to *all* CSPs.
- Simplest thing you can do: for each variable, keep track of which values are valid for it at each step. When picking the next variable to be updated, pick the one with the fewest possible values. This is called the **minimum remaining values** (RMV) heuristic.
- E.g., if some variable has zero values available, this will be immediately detected and cause a backtrack. Also, if a variable has only a single choice, it makes sense to pick it first b/c you'll have to make that decision eventually *anyway* and that will help constrain *future* variables.
- If all choices are equal (e.g., the first time that you pick a node in Australia), then pick the one that connected to the most other variables. This is the **degree heuristic** – it attempts to find the variable that has the most *influence* on its neighbors.
- Finally, once you've picked a variable to go next, you have to pick which *value* for that variable to try first. In this case, you could use the **least constraining value** – the value that influences the neighbors the least (i.e., rules out the fewest values among neighbors).
- The general property we're going for here is that we should first pick things that reduce our branching factor the most (i.e., do things that have to happen as soon as possible), but leave the maximum solution flexibility otherwise.
- A more sophisticated thing you can do is ensure that you detect the possibility of constraint conflict as early as possible.
- This process is called **constraint propagation**.

- This is more sophisticated than just making sure that each assignment rules out blocked values for its neighbors – you also need to make sure that all *consequences* of your choice are propagated through the system.
- This is done with an **arc consistency** algorithm.
- First, rewrite every arc in the constraint graph as a bidirectional arc. Basically, each of these says “that node has to be consistent with me”.
- Formally, we’ll say that an arc from node x to node y is **consistent** iff for every value of x there is *some* value of y that satisfies the arc.
- When you find an arc that *isn’t* consistent, you delete elements from its domain until it is consistent or it’s empty.
- Example: if SA has the domain {blue} and NSW has the domain {red, blue}, then the arc from SA to NSW is consistent (because for SA=blue, you can choose NSW=red), but the arc from NSW to SA is *not* consistent (because if you take NSW=blue, there’s no choice available for SA). So you delete blue from NSW and then both arcs are consistent.
- Of course, it’s not enough to simply do arc consistency *once* between all nodes in the graph. Changing the domain of one variable may render all of its other neighbors inconsistent. It turns out that you have to update arcs *repeatedly* until no more inconsistencies are detected.
- It’s not obvious (and I won’t prove it for you here), but this can be done in polynomial time ($O(n^2d^3)$ absolute worst case, though you can usually do better than that even). There’s an algorithm called AC-3 (and its successor AC-4, and probably some more modern ones) from mid-1980’s that do this.
- After doing this constraint propagation via arc consistency checking, you will have wiped out a lot of the immediate problems and detected a lot of the potential failure points early. In combination with the other heuristics, you can *dramatically* speed up the final search.
- Of course, the general CSP is *still* NP-hard, so even this won’t solve all your problems.