Artificial Intelligence: 6.034
Massachusetts Institute of Technology
Spring 2012

March 23, 2012
Recitation 7

Motion Planning and CSPs

Motion Planning

- Sometimes, the state space over which we need to search is not discrete, so we can't represent goals and actions in the symbolic way we used before. But, for planning the motion of a robot, we can use representations specific to that domain.
- A **configuration** is a complete specification of the robot's body position and location in the world.
- The world can contain **obstacles**, through which the robot can't move. A configuration is in **collision** if the robot is hitting an obstacle. When computing a path, we want all of the configurations to be collision-free.
- Two configurations are **visible** to each other if they can be connected by a straight line (in configuration space) that does not pass through any obstacles. We can discretize the space by only considering configurations at the corners of obstacles, and drawing edges between pairs of configurations that are visible to each other.
- Another way to discretize the configuration space is to build a **roadmap**, by randomly sampling configurations from the space (discarding any that are in collision) and drawing edges from each configuration to its k nearest visible neighbors. This method does not depend on what the initial and goal configurations are, so the roadmap can be reused for multiple searches. However, this is inefficient.
- The **Rapidly-exploring random tree (RRT)** algorithm finds a path by constructing a tree rooted at the initial configuration. A configuration is randomly sampled from the space, its nearest neighbor in the tree is found, and the new point is added to the tree with an edge from the neighbor (or, if that edge would intersect an obstacle, a new point in that direction is added). Periodically, instead of sampling a random configuration, the goal configuration is used. This process is repeated until the goal configuration is added to the tree. The resulting path can then be **smoothed**, by selecting random pairs of points in the path and trying to connect them (creating a "shortcut"). A faster version of this algorithm is **bidirectional**, building a tree from the goal as well as one from the initial configuration.

Constraint Satisfaction Problems (CSPs)

- Constraint satisfaction problems can be used when we need to find some assignment of values to variables, where some assignments are valid and some aren't but otherwise none are "better" than others. A **constraint** is some restriction on some set of variables (e.g., "if x is 1 then y can't be 2 unless z is 3").

- We can draw a **constraint graph**, where each node is a variable and each edge (**constraint arc**) corresponds to a constraint and connects the variables involved in that constraint. In **binary CSP**, each constraint only applies to two variables.

- A constraint arc between variables V_i and V_j is **consistent** if, for each x in V_i's domain, there is some y in V_j's domain such that the edge's constraint is satisfied. In **constraint propagation** (aka **arc consistency**), a value is eliminated from a domain if there are no possible assignments of its neighbors that would satisfy the constraints. This can be repeated until all constraint arcs are consistent. One algorithm used for checking arc-consistency is called **AC-3**.

- However, the fact that all constraint arcs are consistent does not necessarily mean that a solution has been found; there could be no solutions, or there could be multiple solutions. So, we can perform depth-first search with backtracking, propagating constraints along the way. Instead of performing the entire constraint-propagation process at every step, we can just do **forward-checking**: each time a value is assigned to a variable, we can eliminate any inconsistent values from the domains of its neighbors in the constraint graph.

- Because we're just looking for an assignment of values to variables, we can assign them in any order. It is generally faster to assign the most-constrained variable next (the one with the smallest domain) (this is called the **minimum-remaining-values (MRV)** heuristic), and to assign it the least-constraining value (the one that least reduces the sizes of its neighbors' domains).

- An alternative algorithm is to use a greedy heuristic to find a solution "near" correct, then use the **min-conflict heuristic**, assigning new values that minimize unsatisfied constraints, until a solution is found.

Exercises

1. (AIMA 6.4) Give precise formulations for each of the following as constraint satisfaction problems:

   (a) Rectilinear floor-planning: find non-overlapping places in a large rectangle for a number of smaller rectangles.

   (b) Class scheduling: There is a fixed number of professors and classrooms, a list of classes to be offered, and a list of possible time slots for classes. Each professor has a set of classes that he or she can teach.

   (c) Hamiltonian tour: given a network of cities connected by roads, choose an order to visit all cities in a country without repeating any.

2. (Based on AIMA 6.11) The following figure shows a map of the states and territories of Australia:

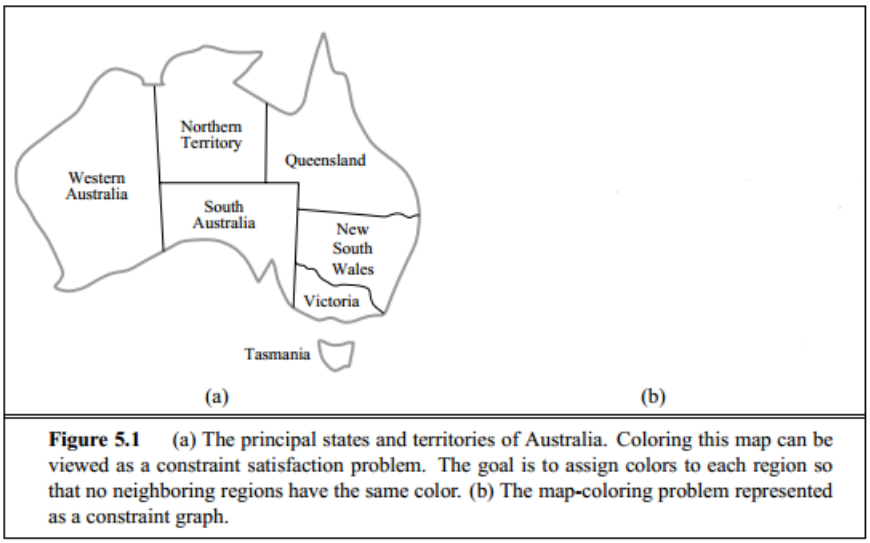

Figure 5.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

   We are given the task of coloring each region either red, green or blue in such a way that no neighboring regions have the same color.

   (a) Choose a CSP formulation. What are the variables, what are the possible values of each variable, and how are the variables constrained?

   (b) Draw the constraint graph for this CSP formulation.

   (c) Use the AC-3 algorithm to show that arc consistency can detect the inconsistency of the partial assignment $\{WA = green, V = red\}$.

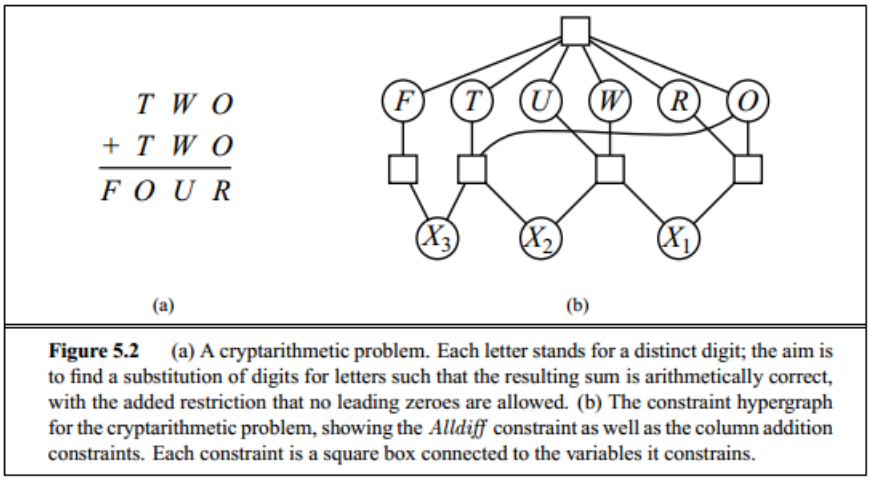

Figure 5.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint as well as the column addition constraints. Each constraint is a square box connected to the variables it constrains.

3. (AIMA 6.5) Figure 5.2 shows a cryptarithmetic puzzle. Each letter in a cryptarithmetic puzzle represents a different digit: this would be represented as the global constraint $Alldiff(F, T, U, W, R, O)$. The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:

$$\begin{aligned} O + O &= R + 10 \cdot C_{10} \\ C_{10} + W + W &= U + 10 \cdot C_{100} \\ C_{100} + T + T &= O + 10 \cdot C_{1000} \\ C_{1000} &= F \end{aligned}$$

where C_{10}, C_{100}, and C_{1000} are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. These constraints can be represented in a **constraint hypergraph**, which consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints.

Solve the cryptarithmetic problem in Figure 5.2 by hand, using the strategy of backtracking with forward checking and the MRV and least-constraining-value heuristics.