CS 188
Spring 2020

Section Handout 2

CSPs

CSPs are defined by three factors:

1. *Variables* - CSPs possess a set of N variables $X_1, ..., X_N$ that can each take on a single value from some defined set of values.
2. *Domain* - A set $\{x_1, ..., x_d\}$ representing all possible values that a CSP variable can take on.
3. *Constraints* - Constraints define restrictions on the values of variables, potentially with regard to other variables.

CSPs are often represented as constraint graphs, where nodes represent variables and edges represent constraints between them.

- *Unary Constraints* - Unary constraints involve a single variable in the CSP. They are not represented in constraint graphs, instead simply being used to prune the domain of the variable they constrain when necessary.
- *Binary Constraints* - Binary constraints involve two variables. They're represented in constraint graphs as traditional graph edges.
- *Higher-order Constraints* - Constraints involving three or more variables can also be represented with edges in a CSP graph.

In **forward checking**, whenever a value is assigned to a variable X_i, forward checking prunes the domains of unassigned variables that share a constraint with X_i that would violate the constraint if assigned. The idea of forward checking can be generalized into the principle of **arc consistency**. For arc consistency, we interpret each undirected edge of the constraint graph for a CSP as two directed edges pointing in opposite directions. Each of these directed edges is called an **arc**. The arc consistency algorithm works as follows:

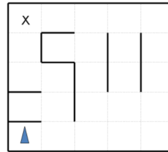
- Begin by storing all arcs in the constraint graph for the CSP in a queue Q.
- Iteratively remove arcs from Q and enforce the condition that in each removed arc $X_i \longrightarrow X_j$, for every remaining value v for the tail variable X_i, there is at least one remaining value w for the head variable X_j such that $X_i = v, X_j = w$ does not violate any constraints. If some value v for X_i would not work with any of the remaining values for X_j, we remove v from the set of possible values for X_i.
- If at least one value is removed for X_i when enforcing arc consistency for an arc $X_i \longrightarrow X_j$, add arcs of the form $X_k \longrightarrow X_i$ to Q, for all unassigned variables X_k. If an arc $X_k \longrightarrow X_i$ is already in Q during this step, it doesn't need to be added again.
- Continue until Q is empty, or the domain of some variable is empty and triggers a backtrack.

We've delineated that when solving a CSP, we fix some ordering for both the variables and values involved. In practice, it's often much more effective to compute the next variable and corresponding value "on the fly" with two broad principles, **minimum remaining values** and **least constraining value**:

- *Minimum Remaining Values (MRV)* - When selecting which variable to assign next, using an MRV policy chooses whichever unassigned variable has the fewest valid remaining values (the *most constrained variable*).
- *Least Constraining Value (LCV)* - Similarly, when selecting which value to assign next, a good policy to implement is to select the value that prunes the fewest values from the domains of the remaining unassigned values.

1 Search and Heuristics

Imagine a car-like agent wishes to exit a maze like the one shown below:



The agent is directional and at all times faces some direction $d \in (N, S, E, W)$. With a single action, the agent can *either* move forward at an adjustable velocity v *or* turn. The turning actions are *left* and *right*, which change the agent's direction by 90 degrees. Turning is only permitted when the velocity is zero (and leaves it at zero). The moving actions are *fast* and *slow*. *Fast* increments the velocity by 1 and *slow* decrements the velocity by 1; in both cases the agent then moves a number of squares equal to its NEW adjusted velocity (see example below). A consequence of this formulation is that it is impossible for the agent to move in the same nonzero velocity for two consecutive timesteps. Any action that would result in a collision with a wall crashes the agent and is illegal. Any action that would reduce v below 0 or above a maximum speed $V_{\max}$ is also illegal. The agent's goal is to find a plan which parks it (stationary) on the exit square using as few actions (time steps) as possible.

As an example: if at timestep t the agent's current velocity is 2, by taking the *fast* action, the agent first increases the velocity to 3 and move 3 squares forward, such that at timestep $t + 1$ the agent's current velocity will be 3 and will be 3 squares away from where it was at timestep t. If instead the agent takes the *slow* action, it first decreases its velocity to 1 and then moves 1 square forward, such that at timestep $t + 1$ the agent's current velocity will be 1 and will be 1 squares away from where it was at timestep t. If, with an instantaneous velocity of 1 at timestep $t + 1$, it takes the slow action again, the agent's current velocity will become 0, and it will not move at timestep $t + 1$. Then at timestep $t + 2$, it will be free to turn if it wishes. Note that the agent could not have turned at timestep $t + 1$ when it had a current velocity of 1, because it has to be stationary to turn.

(a) If the grid is M by N, what is the size of the state space? Justify your answer. You should assume that all configurations are reachable from the start state.

(b) Is the Manhattan distance from the agent's location to the exit's location admissible? Why or why not?

(c) State and justify a non-trivial admissible heuristic for this problem which is not the Manhattan distance to the exit.

(d) If we used an inadmissible heuristic in A* graph search, would the search be complete? Would it be optimal?

(e) If we used an *admissible* heuristic in A* graph search, is it guaranteed to return an optimal solution? What if the heuristic was consistent? What if we were using A* tree search instead of A* graph search?

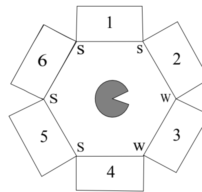
(f) Give a general advantage that an inadmissible heuristic might have over an admissible one.

Q2. CSPs: Trapped Pacman

Pacman is trapped! He is surrounded by mysterious corridors, each of which leads to either a pit (P), a ghost (G), or an exit (E). In order to escape, he needs to figure out which corridors, if any, lead to an exit and freedom, rather than the certain doom of a pit or a ghost.

The one sign of what lies behind the corridors is the wind: a pit produces a strong breeze (S) and an exit produces a weak breeze (W), while a ghost doesn't produce any breeze at all. Unfortunately, Pacman cannot measure the strength of the breeze at a specific corridor. Instead, he can stand *between* two adjacent corridors and feel the max of the two breezes. For example, if he stands between a pit and an exit he will sense a strong (S) breeze, while if he stands between an exit and a ghost, he will sense a weak (W) breeze. The measurements for all intersections are shown in the figure below.

Also, while the total number of exits might be zero, one, or more, Pacman knows that two neighboring squares will *not* both be exits.



Pacman models this problem using variables X_i for each corridor i and domains P, G, and E.

(a) State the binary and/or unary constraints for this CSP (either implicitly or explicitly).

(b) Suppose we assign X_1 to E. Perform forward checking after this assignment. Also, enforce unary constraints.

X_1			E
X_2	P	G	E
X_3	P	G	E
X_4	P	G	E
X_5	P	G	E
X_6	P	G	E

(c) Suppose forward checking returns the following set of possible assignments:

X_1	P		
X_2		G	E
X_3		G	E
X_4		G	E
X_5	P		
X_6	P	G	E

According to MRV, which variable or variables could the solver assign first?

(d) Assume that Pacman knows that $X_6 = G$. List all the solutions of this CSP or write *none* if no solutions exist.