1 Search

BFS, DFS, UCS, A*. For A* to work, the heuristic gotta be admissible (heuristic is always less than or equal to true cost to the end) and consistent (cost from a to b is c, $h(a) - h(b) \leq c$). TODO: If available space, add tree/graph search

2 CSP

Represent problems as constraint graphs; nodes are variables, edges are constraints. Can solve with backtracking search (DFS, try assigning variables until you run into a contradiction, backtrack, can be implemented recursively), forward checking (when a variable is assigned, check consistency of all arcs from unassigned variables to newly assigned variable), arc constistency (if a variable X loses a value in its domain of possible values when checking arc consistency, check all of its neighbors). An arc from unassigned X to assigned Y (i.e. $X \rightarrow Y$) is consistent iff for every $x \in X$, there is some $y \in Y$ which could be assigned without violating a constraint; when checking consistency, delete invalid assignments from X.

3 Adversarial Search

4 Utilities

Utility of a lottery: expectation value of lottery

5 MDPs

Markov property: action outcomes depend only on the current state.
An MDP is defined by: a set of states $s \in S$, a set of actions $a \in A$, a transition function $T(s, a, s')$, a reward function $R(s, a, s')$, a start state s_0, and maybe a terminal state. For model-based methods, we want an optimal policy $\pi^* : S \rightarrow A$.
Instead of measuring overall return R for a trajectory, prefer sooner rewards, so use discount factor γ. This also ensures that discounted returns are finite even for infinite states.
We have 2 relevant values for a policy π:

$$V(s) = \text{expected utility starting in } s \text{ and acting according to } \pi$$

$$Q(s, a) = \text{exp. util. at } s, \text{ taking action } a, \text{ and acting according to } \pi$$

For optimal policy π^*, these are V^* and Q^*.
Because V^* and Q^* represent values for the optimal policy π^*, we can derive recursive formulas from them known as the Bellman equations:

$$V^*(s) = \max_a Q^*(s, a)$$

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \Big(R(s, a, s') + \gamma V^*(s') \Big)$$

$$\implies V^*(s) = \max_a \sum_{s'} T(s, a, s') \Big(R(s, a, s') + \gamma V^*(s') \Big)$$

$$\implies Q^*(s, a) = \sum_{s'} T(s, a, s') \Big(R(s, a, s') + \gamma \max_a Q^*(s, a) \Big)$$

Policy stuff
Value iteration: Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero. Given a vector of $V_k(s)$ values, do one step of expectimax from each state:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \Big(R(s, a, s') + \gamma V_k(s') \Big)$$

Repeat until convergence.
Policy evaluation: Do policy iteration, except instead of doing argmax over all a from a given state, do $\pi(a)$ based on policy π.
Policy extraction: Assume you have optimal values $V^*(s)$. Then, to compute actions from these values, just do a one-step expectimax over possible actions from given state.
Policy iteration: repeat policy evaluation and policy extraction until convergence.

6 RL

7 Bayes Net

8 HMMs

9 Particle Filters

10 Decision Networks

11 Naive Bayes

12 Perceptrons

13 Clustering

14 Deep Learning