CS 171 F24 Final Exam Reference Sheet

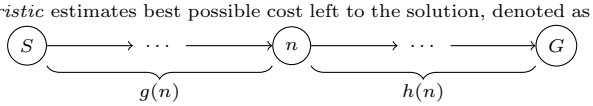
Created by Steven He, 2024

Uninformed Search

A **state** represents particular configuration of environment. **Nodes** in search tree can encode details like state, parent, action, path cost, depth, etc.
Tree Search doesn't remember visited nodes: slow search, low memory
Graph Search remembers visited nodes: fast search, high memory
Basic search scheme maintains explored and frontier nodes. Process the frontier in some order based on some policy/search strategy, marking as visited if graph search, then expanding node.
Strategies evaluted by:
· *completeness* - does it guarantee a solution if one exists
· *optimality* - does it guarantee min-cost solution
· *time/space complexity* - # of nodes generated / max # of nodes in memory. In terms of max branching factor b, depth of min-cost solution d, maximum depth of state space m.
Uninformed search strategies:
· *Breadth-first Search* - Frontier is FIFO queue. Goal test node before push. Always complete; only optimal if cost is non-decreasing function of depth.
· *Uniform-cost Search* - Frontier is priority queue, ordered by total path cost $g(n)$. Goal test after popping from queue. Complete if b is finite; optimal if all step costs $\geq \epsilon > 0$ (non-zero).
· *Depth-first Search* - Frontier is LIFO stack. Defined as Depth-limited search with limit $l = \infty$. Goal test after pop from stack. Incomplete if depth of states is infinite; not optimal.
· *Iterative Deepening Search* - Perform Depth-limited search, while iteratively increasing depth limit. Memory advantage of DFS, completeness & optimality conditions of BFS.

Heuristic Search

A *heuristic* estimates best possible cost left to the solution, denoted as $h(n)$.



$g(n)$: known path cost so far to state n

$h(n)$: estimate of optimal cost to goal from n

$f(n) = g(n) + h(n)$: estimate of total cost to goal through n

A heuristic is **admissible** iff $\forall n, h(n) \leq h^*(n)$, where $h^*(n)$ is true optimal cost to goal. Admissible heuristics never overestimate cost to the goal.
A heuristic is **consistent** iff $\forall n, f(n') \geq f(n)$ where n' is any successor of n. $f(n)$ is non-decreasing along any path.

$$\textbf{consistent} \implies \textbf{admissible}$$
$$\neg\textbf{admissible} \implies \neg\textbf{consistent}$$
$$\textbf{admissible} \not\implies \textbf{consistent}$$

Heuristic search strategies:
· *Greedy Best-first Search* - Frontier is priority queue, ordered by heuristic $h(n)$. Only graph version complete in finite spaces; not optimal even with perfect heuristic $h = h^*$.
· *A^* Search* - Frontier is priority queue, ordered by heuristic + path cost $f(n)$. Complete unless infinite nodes with $f < f(G)$. Optimal with tree search if h is admissible; graph search if h is consistent.
For two heuristics, if $\forall n, h_2(n) \geq h_1(n)$ then h_2 **dominates** h_1. If h_2 dominates h_1 and both are admissible, then h_2 is almost always better, and never expands more nodes than h_1.
A problem with fewer restrictions is called **relaxed** (add more available actions). The solution of a relaxed problem is at least as good as the solution for the original problem. Intuition: If original problem hard to solve, relax the problem state space that is easy to solve. Solve it exactly and use the solution as heuristic for original problem.
Thm. Heuristics that are generated from relaxed models are consistent

Local Search

In some problems, path is irrelevant so we might avoid systematic search. The goal state itself is the solution.
Random restart wrapper - Repeatedly perform local search, each time starting from a new random start state
Tabu search wrapper - Within each random restart, add visited states to a "tabu" list of states that are temporarily excluded from being revisited.
Local search algorithms:
· *Hill Climbing* - At some current state, always compare to neighboring state with maximum value. If current state has better value, return, otherwise set current to the best neighbor. Neither complete nor optimal since we always approach local and not global optima relative to current position.
· *WalkSAT* - Starting from random assignment, instead of always picking neighbor with most satisfied clauses, with some probability p, flip a random symbol (pick random neighbor). Randomness can potentially help escape local optima.
· *Local Beam Search* - Create k random initial states, generate their children, then select k best children. Repeat until goal found.
· *Simulated Annealing* - Intelligently combine random and greedy moves. Some temperature T gradually decreases, representing state volatility. Temperature gives probability of forcing a random move.
· *Genetic ALgorithms* - Mimic biology by mutating some set of offspring by crossing over traits, and selecting individuals based on fitness (heuristic) function.

Game Search

Games have an adversary, where solution is a strategy.
MiniMax - Generate full game tree. Assume that opponent plays optimally and tries to minimize your utility. Max nodes compute max of its children, Min nodes compute min of its children. Complete if tree is finite, optimal against an optimal opponent. **Static heuristic evaluation** estimates utility of a non-terminal state so heuristic MiniMax can stop before terminal nodes.
Alpha-Beta Pruning
Initially $\alpha = -\infty, \beta = \infty$ representing highest value found at MAX node and lowest found at MIN. Pass values down to child in recursive call, and update values after backtracking (α at MAX nodes, β at MIN). If we hit pruning condition $\alpha \geq \beta$, stop searching rest of subtree.
Iterative deepening reordering can be applied to improve pruning. E.g. reorder children based on heuristic at each layer to be non-increasing on max layer and non-decreasing on min.
Expectimax - Variant of Minimax for nondeterministic games, where utility factors in probability. $\alpha\beta$-pruning still fundamentally works. **Monte Carlo Tree Search** - Sampling outcomes to determine moves when branching factor is too high. Sample = self-play rollout, simulate entire game until terminal node.
Exploitation is making the best decision given current info vs. *exploration* is gathering more info.
Select action maximizing Upper Confidence Bound (UCB), since we choose to be optimistic even with uncertainty.

$$a_t = \arg\max_{a \in \mathcal{A}} \hat{Q}_t(a) + \hat{U}_t(a)$$

Where $\hat{Q}_t(a)$ is estimate of action a, and $\hat{U}_t(a)$ is the uncertainty of a. Optimistically the upper bound is the sum.

$$\text{Hoeffding's Inequality: } \mathbb{P}[Q(a) > \hat{Q}_t(a) + \hat{U}_t(a)] \leq e^{-2N_t(a)U_t(a)^2}$$

i.e. The probability UCB under estimated true value of a (we made a mistake), based on samples of action a, $N_t(a)$ and uncertainty $U_t(a)^2$.

MCTS Move Selection

$$\text{UCB} = \frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

Where $\frac{w}{s}$ denotes $\frac{\text{parent wins}}{\text{samples}}$, having selected node i with parent p, and c is some exploration parameter. To pick a move for p, select child with highest computed UCT. Once a leaf node is reached, simulate rest of the game and propagate results back up the tree.

Constraint Satisfaction

CSP formally defined as finite set of variables $x_1, \ldots, x_n$, with nonempty domain of possible values $d_1, \ldots, d_n$, a set of constraints $c_1, \ldots, c_m$ defining allowed combinations of values.
A **assignment** is *complete* if every variable has a value, *partial* if some variables have no values, *consistent* if no constraints violated. A **solution** is a complete and consistent assignment. Any CSP can be expressed as a binary CSP.
When ordering variables for backtracking, prioritize variable selection by **MRV** (minimum remaining values), then break ties with **degree heuristic** (variable involved in constraints with the most unassigned variables).
When ordering values, **LCV** (Least Constraining Value) heuristic, pick value in remaining domain that rules out fewest values in remaining variables.

Inference for CSPs

Constructing/deducing new explicit relationships/constraints that were originally implicit.
Forward checking (FC) - Update effective domain of unassigned variables based on local constraints. Then undo/restore updates on backtrack. If pruned domain of neighbor becomes empty, immediately backtrack.
Constraint propagation - Similar to FC, but also goes beyond immediate neighbors and *propagates* changes through the graph.
Def. Arc(X_i, X_j) is arc-consistent if for every value of X_i, there exist a matching (and consistent) value X_j.
AC-3 Algorithm - $O(n^2k^3)$, for n variables and domain size k:

```
Q ← queue of all arcs (X_i, X_j) ▷ Undirected arcs pushed as (X_i, X_j) and (X_j, X_i)
while Q not empty do
    (X, Y) ← pop from Q
    Remove inconsistent values from X
    if Any change in Domain(X) then
        Push arcs (Z, X), ∀Z into Q
    end if
end while
```

CSP's are NP-Complete, however special easier cases exist:
Thm. If a CSP constraint graph is a tree, it can be solved in $O(nk^2)$ time.
Cutset conditioning and **tree decomposition** can exploit this property.
ADD STUFF FOR IMPROVING BACKTRACKING
ADD STUFF FOR LOCAL SEARCH FOR CSPS

Logic

todo

Probability & Uncertainty, Bayesian Networks

todo

Intro to ML, Linear Regression, kNN

todo

Decision Trees and Neural Networks

todo

Reinforcement Learning

Markov Property - Future is independent of past given present:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t] \text{ where } S_t \text{ is state at time } t$$

We use matrix $\mathcal{P}$ to define transition property from state s to s', denoted as probability in row s, column s'.

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \ldots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \ldots & \mathcal{P}_{nn} \end{bmatrix}, \mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$$

Markov Process/Chain - Sequence of states $S_1, S_2, \ldots$ satisfying Markov property. Formally defined as tuple $\langle \mathcal{S}, \mathcal{P} \rangle$ i.e. (set of states, prob matrix)
Episode - Some sequence of traveresed states in a MP
Markov Reward Process - Give states in MP some reward. We "gain" reward R_{t+1} when transitioning from states $S_t \to S_{t+1}$
placeholder intermediary stuff goes here

	Evaluate Policy, π	**Find Best Policy, π^***
MDP Known (Planning probs)	Policy Evaluation	Policy/Value Iteration
MDP Unknown	MC and TD Learning	Q-Learning