CSE 415 Final Prep

zjmcnulty

March 2019

Contents

1 State Space Search **3**
- 1.1 State Space . . . 3
- 1.2 Operators . . . 3
- 1.3 Problem Space Graph . . . 3
- 1.4 Heuristics . . . 3
  - 1.4.1 Admissibility . . . 3
  - 1.4.2 Consistency . . . 4
- 1.5 Problem Formulation . . . 4

2 Adversarial Search - 2 player Zero sum games **5**
- 2.1 Static Evaluation Functions . . . 5
- 2.2 Minimax Search . . . 5
- 2.3 Alpha Beta Pruning . . . 5
- 2.4 Zobrist Hashing . . . 6
- 2.5 Expectimax Search . . . 6

3 Probabilistic Reasoning **7**
- 3.1 Conditional Probability . . . 7
- 3.2 Bayes Rule . . . 7
- 3.3 Joint Distributions . . . 7
- 3.4 Independence of Random Variables . . . 7

4 Bayes Nets **8**
- 4.1 Determining number of nonredundant probabilities required . . . 8

5 Markov Decision Processes **9**
- 5.1 Value Iteration . . . 9
- 5.2 Q-Learning . . . 9
- 5.3 Sample-Based Q-value iteration . . . 10

6 Markov Models/Chains **11**

7 Hidden Markov Models **12**
- 7.1 Forward Algorithm . . . 12

8 Natural Language Processing (NLP) **13**
8.1 Parse Trees 13
8.2 Cosine Similarity and Stemming : comparing documents 13

9 Perceptrons **14**
9.1 Perceptron Training Algorithm 14

10 Future of AI **15**
10.1 Asimow's Three Laws 15
10.2 Kurzweil's Singularity 15

11 Misc **16**

1 State Space Search

1.1 State Space

Set, Σ,of all possible states for some problem.

1.2 Operators

Operators are **partial functions**, not defined on entire domain/state space, from the state space to itself, $\phi : \Sigma \to \Sigma$. Each operator has a **precondition** that must be true about the current state before the operator can be applied. Operators define how we move through state space. Together with the set of states Σ the set of operators Φ define a **problem space** (Σ, Φ).

1.3 Problem Space Graph

Draw a graph with a vertex for each state and an edge whenever there is a valid move from state σ_i to state σ_j: i.e. $\phi_k(\sigma_i) = \sigma_j$. Label the edge with the operator used ϕ_k.

1.4 Heuristics

A heuristic is a function used in best-first searches (i.e. A^* search) that approximates the distance to the goal state from a given state/node. This allows the search algorithm to explore nodes possibly closer to the goal state first, reducing the amount of steps that is required to reach the goal. This call an **informed search** and can improve on the performance of blind searches like BFS, DFS, and Uniform Cost Search.

1.4.1 Admissibility

A heuristic is admissible if it never overestimates the distance from the current state to the goal. i.e. if $d(s, \gamma)$ is the distance from the current state to the goal state γ then the heuristic h is admissible iff:

$$h(s) \leq d(s, \gamma) \qquad \forall s \in \Sigma, \gamma \in \Gamma$$

i.e. if there are multiple goal states, the heuristic is less or equal to than the true distance to the nearest goal state.

IMPORTANCE: If a given heuristic is admissible, an A^* search using it will find the goal state along an optimal path the first time the goal state is discovered (i.e. we can stop algorithm once we find the goal and we will have an optimal path).

1.4.2 Consistency

A heuristic h is consistent if it never over approximates the distance between two states/nodes of the state space tree. Specifically, for each edge (s_i, s_j) in state space where $d(s_i, s_j)$ is the length/cost of that edge:

$$h(s_i) - h(s_j) \leq d(s_i, s_j)$$

IMPORTANCE: If a heuristic is consistent then A^* never has to re-expand a node. ALSO, note that consistency implies admissibility.

1.5 Problem Formulation

Define what a state is and what information it stores, the total set of states Σ, the set of operators (and their precondition) Φ, the initial state, and the goal state(s).

2 Adversarial Search - 2 player Zero sum games

A 2-player zero sum game is a game in which there are two opponents who often take turns making moves. Furthermore, there are no 'win-win' or 'lose-lose' outcomes in the game: every game has features/goal states that are good for exactly one of the players. Adversarial search is the process of searching form the state spaces of these games, often in order to devise some sort of optimal play strategy.

2.1 Static Evaluation Functions

Often, in 2PZS the state space is too large to inspect every possible state resulting from an action. Consider chess for example. Rather, the search will stop after some number of moves and evaluate the game as is. In order to measure how "good" a given position is for one player or for the other, we need some way of evaluating a state. A static evaluation function does just that: given a state it calculates some properties of the state in order to determine how good it is for each player. Often, we have a high score for a state represent a good score for player 1 (the maximizing player) and a bad score for player 2 (the minimizing player).

2.2 Minimax Search

Assuming there are two perfectly rational players scoring each state using the same static evaluation function, we can infer some of our opponents moves. The maximizing player will always choose the move that forces the highest possible end state and the minimizing player will always choose the move that forces the lowest possible end state.

1. Perform static evaluations on the leaf nodes of the tree.
2. Move up one level. If the parent is a maximizing player/node, choose the maximum value in its children to be thats nodes value. Else if its a minimizing player choose the min value.
3. Move up another layer and repeat.

2.3 Alpha Beta Pruning

Goal is to reduce the amount of state space we have to explore. The basic idea is that since both players are behaving rationally then neither will want to make a move that allows them to force a bad position for them. As such, once we find one such bad position (worse than the positions we have found the current player can force) there is no reason to keep exploring the tree. For example, if the maximizing player already knows they can force a score of 4 but making a specific move there is a move their opponent can make to force a score of 0, why bother exploring that move further? Let:

α = highest value for maximizing player from current node to root.
β = lowest value for minimizing player from current node to root.

If $\alpha \geq \beta$, prune and no need to explore that subtree further so traverse back to parent. This requires you to perform the static evaluations of each child at the time you reach them and updating the parents value accordingly (keeping in mind whether parent is a minimizer or maximizer; always choose lower/higher respectively value).

2.4 Zobrist Hashing

Goal is to reduce the number of static evaluations we perform as these are often expensive. The idea is to store the static evaluations in a hash table that has as few collisions as possible and where it is fast to calculate the hash of a given state. For board based games (where there are set squares and pieces) zobrist hashing is the tool for you!

1. If there are n possible squares that pieces can sit on and m possible states for just that one square (i.e. what pieces can be on that square; don't forget to include the empty state) generate an $n \times m$ array of random numbers.
2. set the initial hash value to zero.
3. For each square on the board, find what state it is in (what piece is on it or if its empty) and extract the element of our random array corresponding to that square/piece combo and XOR it with the current hash.
4. Repeat!

Hash in this way generates remarkably few collisions, but it also has a cool property: you can easy calculate the new hash after making a move due to the reversible nature of XOR (i.e. if $c = a^b$ then we can easily find what a was using $c^b = a$). So for every piece that moved simply XOR out its old position and XOR in its new one (if only few pieces moved).

2.5 Expectimax Search

Now, assuming our opponent has some randomness in their moves, we give each state a value based on the expected value of our opponents moves. We still maximize this expected value.

3 Probabilistic Reasoning

3.1 Conditional Probability

The probability of one event given you know another event to be true. This added information may or may not change the probability this first event occurs.

$$P(X \mid Y) = \frac{P(X \cap Y)}{P(Y)}$$

3.2 Bayes Rule

Bayes rule is a way of calculating the **posterior probability**, the probability a hypothesis is true (i.e. I am in this state) given evidence, from the **prior probability**, the probability that we get certain evidence given some state. Mathematically:

$$P(H \mid E) = \frac{P(E \mid H)P(H)}{P(E)}$$

3.3 Joint Distributions

3.4 Independence of Random Variables

Two events (random variables) A, B are independent if knowing one occurred does not effect the probability that the other occurred. For example, rolling a die and flipping a coin are independent because knowing I got a 6 on the die does not effect the probability I flipped a heads on the coin. Mathematically, A, B are independent iff:

$$A \perp\!\!\!\perp B \leftrightarrow P(A \cap B) = P(A)P(B) \leftrightarrow P(A \mid B) = P(A)$$

Two events (random variables) can also be **conditionally independent**. i.e In the case that C is known to already occur then A is conditionally independent of B iff knowing B also occurs does not change the probability that A occurs. Formally:

$$A \perp\!\!\!\perp B \mid C \leftrightarrow P(A \mid B \cap C) = P(A \mid C) \leftrightarrow P(A \cap B \mid C) = P(A \mid C)P(B \mid C)$$

4 Bayes Nets

A Bayes net is a compact way of writing a joint distribution that takes into account the independence of various random variables and the redundancy in defining some probabilities:

i.e why define P(X = T) = 0.4 and P(X = F) = 0.6 when you know they must sum to one? Simply defining P(X = T) = 0.4 is enough. Then, it follows that

These are Directed Acyclic Graphs where each edge (X, Y) represents that (X, Y) are not conditionally independent. Thus, only nodes for which there exists no PATH (not necessarily no edge) between them are conditionally independent.

$$P(X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = \prod_{i=1}^{n} P(X_i = x_i \mid parents(X_i) = x) \quad (1)$$

4.1 Determining number of nonredundant probabilities required

If random variable X_i has $|X_i|$ values in its domain (i.e. it can take on that many values), then the number of nonredundant probabilities required to characterize each node of the Bayes Net is:

$$(|X_i| - 1) * \prod_{p \in parent(X_i)} |p|$$

Simply add all these up for each random variable to get the total number of probabilities required.

5 Markov Decision Processes

A Markov decision process $M(S, A, T, R)$ describes a system where the agent has a choice of moves it can make to take it through the state space, but there is a randomness associated to each move. The state the agent ends up is not fixed, but with some given probability distribution based on the initial state and the action chosen they will end up in a different choice. This process consists of four components:

- The state-space S
- The set of operators/actions A which define how we can move through state space
- The transition probabilities $T(s, a, s')$ for all $s, s' \in S$ and $a \in A$ that gives the probability of reaching state s' from state s given we took action s.
- The rewards $R(s, a, s')$ for all $s, s' \in S$ and $a \in A$ that gives the reward for making the given transition from state s to state s' by taking action a.

5.1 Value Iteration

Given a discount factor γ, transition probabilities $T(s, a, s')$, and rewards $R(s, a, s')$:

$$V_{k+1}(s) = max(\sum_{s' \in States} T(s, a, s') \left(R(s, a, s') + \gamma V_k(s')\right) : a \in Actions) \quad (2)$$

Drawbacks

- Cannot compute optimal policy directly from the state values; need Q-state values.

5.2 Q-Learning

Given a discount factor γ, transition probabilities $T(s, a, s')$, and rewards $R(s, a, s')$:

$$Q_{k+1}(s, a) = \sum_{s' \in States} T(s, a, s') \left(R(s, a, s') + \gamma * max_{a'}(Q_k(s', a'))\right) \quad (3)$$

The idea here is that for each Q-value, we update its value by finding the expected value of the transition that is about to occur: we have decided on an action, a, and now probability decides our move. For each possible move $Q(s, a)$

we want to know what rewards we could possibly get out of it, $R(s, a, s')$ but also what kind of moves we have available in the state we are left in, $Q(s', a')$

Pros

- Can directly compute the optimal policy from all the Q-values.

Cons

- Larger number of updates must occur than with value iteration: there are more $Q(s, a)$ — one for each state/action pair — than there are $V(s)$ — one for each state.

5.3 Sample-Based Q-value iteration

Suppose we do not know the transition function T or the reward function R. All we can do is move around and see what happens, recording what actions we took, where they brought us, and what reward we got. This requires a new form of approximation of the Q-values. Given a learning rate α (how much do we value new samples over old ones; higher α favors new) and discounting rate γ (same as before). Let s be the state we began in, a the action we chose. Our sample generates a reward r and a new state s' where we ended up. Thus:

$$Q_{t+1}(s, a) = (1 - \alpha)Q_t(s, a) + \alpha(r + \gamma * max_{a'}Q(s', a'))$$

Specifically, we update our Q-value based on the reward we received and how good of a move we have available from the state we ended up in (maximizing over the available actions a' at that new state s').

Amazingly, even if you choose suboptimal actions this system will converge to the true Q-values: **offpolicy learning**. This requires you to adjust the learning rate to decrease over time and whatnot (see slide 13-Q learning slide 3).

An ideal learning strategy minimizes **regret**: learn to be optimal optimally! How many mistakes did you make along the way to learning the optimal policy?

6 Markov Models/Chains

A sequence of random variables evolving based on some transition matrix. In these processes, the only thing that effects a transition is what state you are currently in. i.e. X_{t+1} is conditionally independent of $X_1, ..., X_{t-1}$ given X_t: the former provide no additional information on the transition that might occur at time $t+1$. Thus:

$$P(X_{t+1} \mid X_1, ..., X_t) = P(X_{t+1}|X_t)$$

Thus, do to this conditional independence we can rewrite the probability of a sequence of states $X_1, X_2, ..., X_t$ occuring as :

$$P(X_1 \cap X_2 \cap ... \cap X_T) = P(X_1) \prod_{t=2}^{T} P(X_t \mid X_{t-1})$$

Eventually, this system may come to a steady-state. Here, we can find the **stationary distribution**: where the probability distributions converge to over time:

$$P_\infty(X = s) = \sum_{x \in States} P(X_{t+1} = s|X_t = x) P_\infty(X = x)$$

The above simply says that the stationary distribution, long term probability of ending in state s, is simply the sum of (the probability of transitioning from another state x to s times the long-term probability of being in state x). Now, $P_\infty(s)$ and all $P_\infty(x)$ are unknown, so this generates a system of equations. However, using the fact that $P_\infty(s) + P_\infty(x_1) + ... + P_\infty(x_n) = 1$, we can solve the desired system for the stationary probabilities.

7 Hidden Markov Models

In this case, we do not know what states we are currently in. Rather, we have a series of observations about the world at each given point of time and we will use this information to try to make some guesses about our current state.

7.1 Forward Algorithm

Goal: try to use a series of observations about previous states to predict the current state. If X_t is our current state and E_i the observation received at state i, than mathematically the goal is to determine:

$$P(X_t \mid E_1, E_2, ..., E_t)$$

We start with some initial belief about the probability distribution of the states of our system, $P_0(X)$, and from there use evidence to adjust our beliefs about the system. At $t = 1$ since we have no previous states to transition from, we simply weight our initial beliefs using the given evidence:

$$B_1(X_1 = x) = P(E_1 \mid x)P_0(x) \quad \forall x \in X$$

where $x \in X$ is the set of all possible values X can take on. As we see here, the evidence is weighting our given beliefs about the initial probability distribution: if the probability of observing E_1 given $X_0 = x$ is low, we would expect $P(X = x)$ to be low. Normalize B_1 to regain the probability distribution P_1. If P_0 does not reflect this, it is scaled down in P_1. For further steps, we also begin to incorporate the probability of transitioning from a previous state. To do so, we have to consider how likely it was we were in that initial state, how likely the transition is, and how much the evidence received at the new state supports that value of that state. Thus:

$$B_{t+1}(X_{t+1} = x) = P(E_{t+1} \mid x) \sum_{s \in X} P(X_{t+1} = x \mid X_t = s)P_t(X_t = s)$$

Again, normalize B_{t+1} to regain the probability distribution.

8 Natural Language Processing (NLP)

8.1 Parse Trees

Given a set of grammatical rules, these rules often occur at varying frequencies in the action language. While we can construct many possible parse trees, due to the varying frequencies of these grammatical components there are often some trees which are more likely than others.

$$score = log_2\left(\frac{1}{P(rule))}\right)$$

To find the total score of a parse/grammar tree, add up the scores: the lower the better. To extract a probability from this note that

$$log_2\left(\frac{1}{P(rule_1))}\right) + log_2\left(\frac{1}{P(rule_2))}\right) = log_2\left(\frac{1}{P(rule_1))} * \frac{1}{P(rule_2))}\right)$$

So just add up the scores and the total probability will be $\frac{1}{2^{total\_score}}$

8.2 Cosine Similarity and Stemming : comparing documents

With this, we use the bags of words representation where we not only make a collection of important words, but we store duplicates so we can get a count of the number of each word we find. Once we form these reference words we can turn them into a vector where each entry stores the word count. Then, we can do document comparisons with a dot product!

1. Get rid of stopwords (i.e. transition words like the, and, a, etc... and other words that do not reveal much information about topic of document).
2. Stemming: get rid of various suffixes and prefixes; put all words in the same tense. Makes easier for comparison
3. What remains are **reference words**. These words confer information about the contents of the document.
4. For each reference word, form a vector where index i stores a count of the number of reference word w_i found in the document. Construct such a vector for each document to compare.
5. Take the dot product to find the "angle" between these two vectors

$$\cos(\theta) = \frac{\langle v_1, v_2\rangle}{||v_1|| * ||v_2||}$$

6. large angle (small cosine) tells us the documents are very different while a small angle (large cosine) tells us the documents are very similar.

9 Perceptrons

9.1 Perceptron Training Algorithm

1. Let $X = P \cup N$ be the set of training inputs to the perceptron where P is the set of positive inputs that ought to trigger the perceptron and N the set of negative inputs that ought not to trigger the perceptron. Let $S = \{X_1, X_2, ..., X_n\}$ be a training sequence where the inputs are presented in the order given by S. Let w_k be the set of weights at step k

2. Choose an inital weight vector w_0 and an arbitrary threshold θ (i.e. $\theta = 0.5$.

3. Choose a learning rate/sequence c_k of length n where the ith entry in c_k will be used as a weight during the training with the ith training input. Often, a good choice is $c_k = c_0/k$ for some positive c_0

4. For each step $k = \{1, 2, ..., n\}$

   (a) Classify X_k using the current set of weights w_k and the threshold θ.

      i. If X_k is in N but is classified as positive, set $w_{k+1} = w_k - c_k X_k$

      ii. If X_k is in P but is misclassifed as negative, set $w_{k+1} = w_k + c_k X_k$

      iii. If X_k is correctly classified, set $w_{k+1} = w_k$

NOTE: Perceptron training (only one perceptron) can only converge if the desired function is **linearly separable**. See below and perceptron slides. If the function is linearly separable, this training always converges.

We can use perceptrons to generate logical gates. A single perceptron can only perform a task/logical function if that function is **linearly separable**: i.e. plotting the inputs on a set of axes, we can draw a line/plane/hyperplane to separate the inputs that would fire the perceptron from those that would not. A common example of a task that is NOT linearly separable is the XOR gate.

10 Future of AI

10.1 Asimow's Three Laws

1. A robot may neither injure a human being nor through inaction allow a human being to come to harm.

2. A robot must obey the commands of human beings, provided doing so would not violate the first law

3. A robot must protect its own existence unless it violates either law 1 or law 2.

10.2 Kurzweil's Singularity

Advances in artificial intelligence, computer technology, biology, and brain science, have been progressing at exponential rates. As these developments continue, and as they come to support one another, we can expect a confluence of scientific and technical progress that permits artificially intelligent systems to improve themselves more and more rapidly until a seeming explosion of intelligence occurs in artificial systems. It is this explosion of power of AI that is the predicted phenomenon known as the singularity

11 Misc

Laplace Smoothing: Adding one to the numerator of every probability and N to the denominator, where N is the number of possible states. This helps prevent zero probabilities due to "unlucky" sampling of the probability distribution.