ECE 448 Midterm1 possible questions

Shakey and Blocks world： (1966-72) the world's first mobile "automaton." A* search was developed around 1968 as part of the Shakey project at Stanford.

Towards the end of the project, it was using a PDP-10 with about 800K bytes of RAM. The programs occupied 1.35M of disk space. This kind of memory starvation was typical of the time. For example, the guidance computer for Apollo 11 (first moon landing) in 1969 had only 70K bytes of storage.

Unfortunately, strictly hierarchical planning doesn't always work A famous example is the Sussman anomaly (1975) This comes from a variant of the blocks world. Blocks can be put on an (infinite) table or stacked on top of one another. The robot can move only one block at a time. Suppose we have the following starting state and goal:

```
                                                  A
   Start state            C                goal   B
                    B     A                       C
```

So strictly hierarchical planning doesn't work. We need a more flexible approach to planning. For example, expand what's required to meet **subgoals**, then try to order all the little tasks. So you can "interleave" sub-tasks from more than one main goal.

Waltz line labelling Constraint propagation was developed by David Waltz in the early 1970's for the Shakey project. The original task was line labelling (from Lana Lazebnik based on David Waltz's 1972 thesis).

STRIPS planning： STRIPS planning, or the Stanford Research Institute Problem Solver, was developed in 1971. It is a simple version of classical planning, which is used to manage the high level goals and constraints of a complex planning problem, like waypoints for a robot doing assembly/disassembly tasks.

Boston Dynamics:

We often imagine intelligent agents that have a sophisticated physical presence, like a human or animal.

robot falling down 2017

Google self-driving bike fake

McCulloch and Pitts: early neural nets, 1940s

Another classical example for A^* is the 15-puzzle. A smaller variant of this is the 8-puzzle..

15 2 1 12
8 5 6 11
4 9 10 7
3 14 13

Two simple heuristics for the 15-puzzle are

- Number of misplaced tiles
- Sum of the Manhattan distances between each tile's current location and goal location.

Why Admissibility?

Admissibility guarantees that the output solution is optimal. Here's a sketch of the proof:

> Search stops when goal state becomes top option in frontier (not when goal is first discovered). Suppose we have found goal with path P and cost C. At that point, anything left in the frontier has estimated cost $\geq C$. Since our heuristic is admissible, its true cost must also be $\geq C$.

If the heuristic isn't admissible, A^* will still run but it may return a path that isn't optimal. A^* decays gracefully if this condition is relaxed. If the heuristic only slightly overestimates the distance, then the return path should be close to optimal.

Consistency

The problem with shorter paths to old states is simplified if our heuristic is "consistent." That is, if we can get from n to n' with cost c(n,n'), we must have

$$h(s) \leq c(s, s') + h(s')$$

So as we follow a path away from start state, our estimated cost f never decreases. Useful heuristics are often consistent. For example, the straight-line distance is consistent because of the triangle inequality.

When the heuristic is consistent, we never have to update the path length for a node that is in the done/closed set. (Though we may still have to update path lengths for nodes in the frontier.)

We can view UCS as A* with a heuristic function f that always returns zero. This heuristic is consistent, so updates to costs in UCS involve only states in the frontier.

Other hacks and variants

There are many variants of A*, mostly hacks that may help for some problems and not others.

- "Beam search" keeps frontier small by removing states with high f values. Sometimes necessary when frontier gets very big (e.g. speech understanding) but sabotages guarantees of optimality.
- "Suboptimal" heuristics that occasionally overestimate. Weighted A* is one of these.
- Iterative deepening A*. Iterative deepening, but the cutoffs are based on the A* estimated cost f(s) = g(s) + h(s).
- Pattern databases (cost to solve a partial problem is an under-estimate of cost to solve full problem containing that partial pattern)

CSP problem consists of

- Variables
- Set of allowed values (for each variable)
- Constraints

Notice that graph coloring is NP complete. We don't know for sure if NP problems require polynomial or exponential time, but we suspect they require exponential time. However, many practical applications can exploit domain-specific heuristics (e.g. linear scan for register allocation) or loopholes (e.g. ok to have small conflicts in final exams) to produce fast approximate algorithms.

> However, CSP search has some special properties which will affect our choice of search method. If our problem has n variables, then all solutions are n steps away. So DFS is a good choice for our search algorithm.
>
> - We don't have to worry about finding a shortest path. (They are all length n.)
> - It's not possible to loop. Each step adds a variable value and search cuts off at depth n.
>
> We can use a very stripped down DFS implementation, without explicit loop detection. This can be made to use only a very small amount of memory. In this type of AI context, DFS is often called "backtracking search" because it spends much of its time getting blocked and retreating back up the search tree to try other options.

When to stop and backtrack?

Smart method (**forward checking**): During search, each variable keeps a list of its possible values. At each search step, remove values from these lists if they violate constraints, given the values we've already assigned to other variables. Back up if any variable has no possible values left.

Heuristics for variable assignments

Which variable should we pick?

- (a) Variable with fewest remaining values
- (b) If there are ties, which variable constrains the most other variables (aka degree of node in constraint graph)

Choosing a value

Once we've chosen a variable, what is a good value to choose?

- The one that leaves the most options open for other variables.

AC-3

Fullly working contraint propagation algorithm AC-3, by David Waltz and Alan Mackworth (1977).

The constraint relationship between two variables ("some constraint relates the values of X and Y") is symmetric. For this algorithm, however, we view constraints between variables ("arcs") as ORDERED pairs. So the pair (X,Y) will represent contraint flowing from Y to X.

Revise(X,Y) prunes values of D(X) that are inconsistent with what's currently in D(Y).

Maintain queue of pairs ("arcs") that still need attention.

Initialize queue. Two options

- all constraint arcs in problem [for starting up a new CSP search]
- all arcs (X,Y) where Y's value was just set [for use during CSP search]

Loop:

- Remove (X,Y) from queue. Call Revise(X,Y).
- If D(X) has become empty, halt returning a value that forces main algorithm to backtrack.
- If D(X) changed, push all arcs (C,X) onto the queue, but don't push (Y,X).

Stop when queue is empty

Answer to last week's exercise

In lecture 9, I left the following question unanswered:

> Near the end of AC-3, we push all arcs (C,X) onto the queue, but we don't push (Y,X). Why don't we need to push (Y,X)?

There's two cases. First (Y,X) might already be in the queue.

- This is our first pass through all the arcs, when we start working on the constraint problem.
- Some other part of constraint propagation has modified D(X) since the last time we checked (Y,X).

If neither of these situations holds, then we've already checked (Y,X) at some point in the past. That check ensured that every value in D(Y) had a matching value in D(X). And, moreover, nothing has happened to D(X) since then. So every value in D(Y) still has a matching value in D(X).

What we've just done in AC-3 is that D(Y) has become smaller. This means that some values in D(X) no longer have matching values in D(Y). But everything in D(Y) still has a matching value in D(X). This doesn't change when we remove unmatched values from D(X).

Hill climbing

Another approach to CSP problems involves randomly picking a full set of variable assignments, then trying to tweak it into a legal solution. At each step, we change the value of one variable to try to reduce the number of conflicts.

Hill-climbing n-queens (from Lana Lazebnik, Fall 2017).

This seems to be how people solve certain complicated graph-coloring problems such as avoiding/reducing conflicts in final exam scheduling.

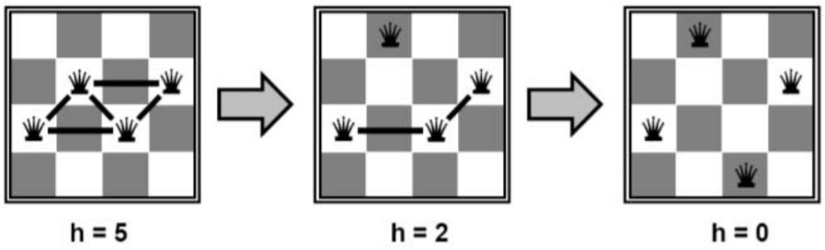
Hill climbing can be very effective, but it can get stuck in local optima. So algorithms often add some randomization:

- Randomly pick a new starting configuration when apparently stuck.
- Add a random component to the movement algorithm, so it sometimes moves to a higher-cost configuration ("simulated annealing").

Hill climbing can be very effective, but it can get stuck in local optima.

Local search for CSPs

- Start with "complete" states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints



h = number of conflicts

Two approaches to planning

There are two basic ways to model planning as a graph search problem:

Situation space planning

- Each state is a complete model of the world.
- Each edge is an action that changes the state of the world.

Plan space planning

- Each state is a partial plan, missing some actions or ordering links.
- Each edge adds detail to the partial plan.

These alternatives are similar to the two approaches we saw for the edit distance problem.

Issues with situation space planning

- very large branching factor
- unnecessarily definite about ordering of unrelated actions

The preconditions and effects are invisible in this diagram. For our tea example, they might be

- effects of start: cold(water), not(in(teabag,teapot)), not(tasty(water), not(in(water,teapot))
- pre-conditions of finish: tasty(water), not(in(teabag,teapot))

What could be wrong with a partial plan?

- "open precondition": an action (including finish) has a precondition with no guarantee that it will be true
- "threat": P is true at some point, but an (unordered) action might cause not(P)

- plan modification operations and how to fix the planning defects

  In general, we have the following set of plan modification operations

  - To fix an open precondition
    - add an ordering link to an existing action
    - add a new action
  - To fix X threatening to "clobber" Y's preconditions
    - order X after Y
    - order Y after X (leaving Y with an explicitly open precondition)

The frame problem and friends

Notice how complicated our representations are already getting. We might wish to make some simplifying assumptions.

The high-level vision of action representations is that the describe how the action changes the world. However, a complete logical representation also requires "frame axioms" that specify all the predicates that an action does not change. For example

> heat(substance,pan) has in(substance,pan) as both a precondition and effect

We need a lot of frame axioms, so it's easy for some to be missing. This nuisance is called the "frame problem." It's usually solved by incorporating a default assumption that properties don't change unless actions explicitly change them. (We've been tacitly assuming that.)

We might want to specify relationships among predicates. For example, how are cold and hot related? Also might want a "Truth Maintenance System" to spell out the consequences of each proposition. For example, automatically account for the fact that the teabag cannot be in two places at once. Such knowledge and inference would allow more succinct (and likely more correct) state and action definitions.

The frame problem is saying that we need lots of frame axioms so that we can properly account for all of the things that are changed and not changed by actions. It is easy to make the mistake of not accounting all frame axioms. And so, instead, we should assume by default that properties don't change unless the actions explicitly change them.

> The formal definition:of conditional probability is
>
> P(A | C) = P(A,C)/P(C)
>
> Let's write this in the equivalent form P(A,C) = P(C) * P(A | C).
>
> Flipping the roles of A and B (seems like middle school but just keep reading):
>
> equivalently P(C,A) = P(A) * P(C | A)
>
> P(A,C) and P(C,A) are the same quantity. (AND is commutative.) So we have
>
> P(A) * P(B | A) = P(A,B) = P(B) * P(A | B)
>
> So
>
> P(A) * P(B | A) = P(B) * P(A | B)
>
> So we have Bayes rule
>
> P(B | A) = P(A | B) * P(B) / P(A)
>
> Here's how to think about the pieces of this formula
>
> ```
> P(cause | evidence) =  P(evidence | cause) *  P(cause) / P(evidence)
>   posterior            likelihood             prior     normalization
> ```

P(evidence | cause) tends to be stable, because it is due to some underlying mechanism.

P(cause | evidence) is less stable, because it depends on the set of possible causes and how common they are right now.

> "Maximum a posteriori" (MAP) estimate
>
> pick the type X such that P(X | evidence) is highest
>
> Or, we should pick X from a set of types T such that
>
> $$X = \text{argmax}_{x \in T} P(x|\text{evidence})$$

> P(cause | evidence) = P(evidence | cause) * P(cause) / P(evidence)
>
> P(evidence) is the same for all the causes we are considering. Said another way, P(evidence) is the probability that we would see this evidence if we did a lot of observations. But our current situation is that we've actually seen this particular evidence and we don't really care if we're analyzing a common or unusual situation.
>
> So Bayesian estimation often works with the equation
>
> P(cause | evidence) $\propto$ P(evidence | cause) * P(cause)

If we know that all causes are equally likely, we can set all P(cause) to the same value for all causes. In that case

P(cause | evidence) $\propto$ P(evidence | cause)

So we can pick the cause that maximizes P(evidence | cause). This is called the "Maximum Likelihood Estimate" (MLE).

The MLE estimate can be very inaccurate if the prior probabilities of different causes are very different. On the other hand, it can be a sensible choice if we have poor information about the prior probabilities.

Jargon:

- word type: a dictionary entry such as "cat"
- word token: a word in a specific position in the text

So the sentence "The big cat saw the small cat" contains seven tokens drawn from five types. There are more tokens than types because, For example, W_2 is "cat" and W_7 is also "cat".

Suppose we have n word types, then creating our bag of words model requires estimating O(n) probabilities. This is much smaller than the full conditional probability table which has 2^n entries. The big problem with too many parameters is having enough training data to estimate them reliably. (I said this last lecture but it deserves a repeat.)

The process of testing

Training data: estimate the probability values we need for Naive Bayes

Development data: tuning algorithm details

Test data: final evalution (not always available to developer) or the data seen only when system is deployed

Training data only contains a sampling of possible inputs, hopefully typically but nowhere near exhaustive. So development and test data will contain items that weren't in the training data. E.g. the training data might focus on apples where the development data has more discussion of oranges. Development data is used to make sure the system isn't too sensitive to these differences. The developer typically tests the algorithm both on the training data (where it should do well) and the development data. The final test is on the test data.

Results of classification experiments are often summarized into a few key numbers.

	Algorithm Claims	
	Spam	**Not Spam**
Spam	True Positive (TP)	False Negative (FN)
Not Spam	False Positive (FP)	True Negative (TN)

We can summarize performance using the rates at which errors occur:

- False positive rate = FP/(FP+TN) [how many wrong things are in the negative outputs]
- False negative rate = FN/(TP+FN) [how many wrong things are in the positive outputs]
- Accuracy = (TP+TN)/(TP+TN+FP+FN)
- Error rate = 1-accuracy

> Or we can ask how well our output set contains all, and only, the desired items:
>
> - precision (p) = TP/(TP+FP)
> - recall (r) = TP/(TP+FN)
> - F1 = 2pr/(p+r)
>
> F1 is the harmonic mean of precision and recall. Both recall and precision need to be good to get a high F1 value.

- How does the size of a Naive Bayes model compare to a full joint distribution?

  The numerator of Bayes rule, i.e P(evidence | cause) * P(cause) is equivalent to the joint probability model. But For n variables, a naive Bayes model has only O(n) parameters to estimate, whereas the full joint distribution has 2^n parameters

- Why does it matter that Naive Bayes reduces the number of parameters we need to estimate?

  Because it decreases drastically the amount of parameters (I.E the test data) needed for Bayes algorithm to work. Bayse algorithm only needs O(n) training data.