Problem Set 1: Air Travel Planning

6.034 Fall 2004

`http://www.ai.mit.edu/courses/6.034f/`

Due: Tuesday, September 28th, 2003

(9/21: Problems through 4)

Ben Bitdiddle, having recently graduated from MIT, went looking for a job. After a long search, he was finally hired by Ur-Bits, a startup company planning to use their proprietary “Primordial Soup” software technology to break into the air travel planning business and steal market share from the ITA Software.

When Ben arrived, he was surprised to discover that his new boss was none other than his old MIT chum Louis Reasoner, who seemed to have an aptitude for management far outstripping his record as a computer programmer.

Sitting down the first day with Ben, Louis brought him up to date. “We’ve just gotten our first big client, Black Helicopter Ltd. They’re a new company with a radical business plan that I can’t tell you about, but we’re very excited to be working with them.

“What I can tell you is that we’re responsible for modelling the types of customers who will be using Black Helicopter flights. Despite other differences, Black Helicopter, like most airlines, is concerned with maximizing the revenue it can extract from each customer — it’s a practice called ‘price discrimination’.

“Basically, price discrimination means they want to sell really pricey tickets to all their customers — but there aren’t enough customers who can afford the really pricey tickets, so they sell moderate tickets to customers with moderate amounts of money. Even a really cheap ticket is better than an empty seat, once you’ve decided to fly the plane at all. The problem is figuring out which customers are which, and arranging things so that the rich customers won’t buy the cheap tickets.”

Ben nodded, and said, “That sounds like a hard problem. So what’s the part the we have to do?”

“Black Helicopter expects to get four types of clients: eccentric millionaires, corporate executives, politicians, and government spooks. We’re using a rule-based reasoning system to catergorize customers into classes according to how much they can afford. The corporate executives are the rich plums, as are the politicians if they’re travelling with an executive who will pay for their ticket. The eccentrics might do anything, so we charge them moderate prices, unless we think they’re an internet millionaire, in which case we charge them just like an executive. Finally, while the spooks are the best repeat customers, they can’t pay much because they’re required to use the lowest bidder.

“We had an intern putting this together, only she’s just gone back to school and didn’t have time to write the rules for politicians and millionaires. She left these notes though...” Louis handed Ben a scribbled piece of paper.

Ben took one look and, recalling 6.034, said, “No problem. We just need five rules to solve this.” Then he told Louis how to do it.

Problem 1: Writing Rules

The intern’s notes said, “Politicians wear suits, unless they are trying to act like ordinary people, in which case they wear jeans. Politicians nover wear jeans when they are travelling with an executive. Politicans always have an outgoing attitude. Eccentric millionaires wear suits, but generally there is something wierd about their clothes, and they always behave wierdly. Internet millionaires, on the other hand, wear jeans, carry a laptop, and are nervous around people, which also means they speak in monosyllables.”

The intern also noted that she had put assertions in the database to reason from. There are six types of assertions: `(person has laptop)`, `(person wearing clothing)`, where clothing is “suit”, “jeans”, “wierd clothes” or “shades”, `(person attitude is demeanor)`, where demeanor is “outgoing”, “unreadable”, “nervous”, “wierd”. `(person and person are together)`, `(person speaks in monosyllables)`, and `(person has silver hair)`.

You need to write rules which use these assertions to deduce the person type (`(person is politician)`, `(person is sponsored politician)`, `(person millionaire)`, and/or `(person internet millionaire)`) as specified by the intern’s notes. Note that `(alice and bob are together)` is not the same as `(bob and alice are together)`, so one of your rules needs to make sure that both exist.

Note that it is possible for a person to belong to more than more than one category, or to not be categorizable according to this system.

Write down five rules Ben could use to implement the intern's notes. The rest of the intern's rules are contained in `passenger.k`

Louis was quite impressed with Ben's quick work, and wanted to take it and show it off immediately. "Wait a minute!" Ben said, "We should try it out and see that it really works first."

Louis grudgingly agreed, and they tried it on with some people and flights. "Hmm..." said Ben, "It's booking people with tickets they can afford, but when I put in two tickets at once, it doesn't choose the most expensive one. In fact, it's selling them both!"

Louis was crestfallen. "Does that mean it won't work after all?"

"No, no," said Ben, "I think I know how to fix it... I'll make it so that it tries to sell the expensive tickets first, then the normal tickets, and the cheap tickets last." He quickly wrote down three new rules, and modified the ticket-selling rule. Then he scratched his head and said, "There's just two other things I have to do... write a rule to prevent it from selling more than one ticket to a customer, and order the rules to make it work correctly."

Problem 2: Execution Order

Your starting place is the rules from `passenger2.k`, plus the rules you wrote for Problem 1.

You need to add one rule that will allow the system to stop when it has sold a ticket. Then you need to change the order of the rules to make sure that the most expensive ticket is the one which is sold first, and that only one ticket is sold.

Ben wasn't very satisfied with his kludge, though: it could only book one flight per data set, which was a major limitation. Finally, feeling desperate, he called up his old friend Alyssa P. Hacker, who he hoped would be able to help him out. After he explained the problem and his attempted solution, Alyssa laughed and said, "You forgot about production rule systems, didn't you? If you're using a deduction system with just IF-THEN rules, then it's hard to keep track state because the system is monotonic. Just DELETE things that aren't true any more when the state changes and it'll work well. But watch out for infinite loops."

Ben thanked her profusely, hung up, and changed his system. After a while, he had a system he thought would work, but it kept getting hung up with infinite loops. Examining the output, he discovered several errors and eventually tracked down his infinite loop problems.

Problem 3: Infinite Loops

Ben went a bit overboard in converting a production rule system, and has built a complex iterative system which can encounter an infinite loop in **four** different ways. You will need to make minor changes to a number of rules in order to fix his system.

Ben's new rules are in `prodpassenger.k`. *Write replacements for some of Ben's rules to fix his infinite loop problem.*

For purposes of this problem, categorization of passengers is assumed to have already been done, so you do not need to worry about the rules from Problem 1.

Satisfied at last, Ben tried to hook up his rules to the main Ur-Bits software engine, only to run into yet another bug. Louis had neglected to tell Ben that the Ur-Bits software used backward chaining to run queries, while Ben had been testing his system with forward chaining.

So Ben had to scrap his production rule system, since the backward chainer couldn't deal with deletions. But Ben wasn't too dismayed, because he remembered from 6.034 that his deduction rules would run just as well backward as forward.

Imagine his surprise when that turned out not to be the case! Running his passenger rules backwards produced a legal booking, but not necessarily the most expensive one. Figuring this was a really tough problem, he called up a third MIT friend, Eva Lu Ator, to get some help. "I don't understand, Eva," complained Ben, “The rules should run the same backwards and forwards. Isn’t that the whole point of separating the knowledge representation from the reasoning algorithm?”

Eva thought carefully, and replied: “Yes, of course. And all of the knowledge you’ve represented explicitly is still producing correct results. But there is knowledge represented implicitly in your system: the fact that a person should be sold the most expensive ticket is encoded implicitly by means of the order of your rules. While explicitly represented knowledge should produce the same final results no matter what reasoning algorithm you use, implicitly represented knowledge has no such guarantees.”

Unfortunately, Ben only understood part of what Eva was saying, and so rather than changing his representation, he just decided to change his rules slightly for backward chaining.

Problem 4: Backward Chaining

Fill in the blank in `backtweak` *with a modification to make to Ben’s* `passenger2.k` *rules that will allow backwards chaining to sell the most expensive tickets to customers.*

For purposes of this problem, categorization of passengers is assumed to have already been done, so you do not need to worry about the rules from Problem 1.

Louis was highly pleased with Ben’s performance and promised that he would add a complimentary memo to his personnel file.

The next day, Louis was back at Ben’s desk again in a panic. “We’ve got a real crisis in my route planning software, and I need your help to straighten it out.”

“What’s wrong?” asked Ben, “Isn’t the route planning just a simple search problem?”

“Simple! If only!” moaned Louis, “It looks like it’s just too big to handle. There are thousands of airports in the world and millions of possible flights between them. The problem is so huge we can’t even find a path from Boston to San Francisco: it’s been running for hours now, and hasn’t finished yet.”

Ben thought this sounded fishy, but agreed to take a look at the problem. Immediately on sitting down with the program, he saw a problem in the search data. “Louis, the program is considering a connection through Moscow. In fact, it looks like it’s considered nothing *but* connections through Moscow for the last thousand tries. What kind of search are you useing here?”

“Well...” said Louis, “It’s a depth first search, because that’s the simplest.”

“It’s not a very smart search, though,” said Ben, “And you’re not using an enqueued list either, so it’s trying lots of different routes through the same set of cities.”

“Oh...” said Louis, “I guess that would make it take a long time?”

“Exponential in the number of nodes,” said Ben, “With an enqueued list it will only take time linear in the number of nodes, but it still won’t get a good result. Here, I’ll show you...”

Ben pulled over a whiteboard, put up the Black Helicopter domestic route map (Figure **??**), and ran the search by hand.

Problem 5: Depth First Search

For the rest of the problem set, you will be using the various types of search described in lecture. The problems can be solved either by hand or by writing and running an appropriate search function. The file `search.scm` *contains search infrastructure, as well as some example search functions.*

You are to perform a depth-first search, with enqueued list, on the six pairs of cities listed. For your answer, record a list of two elements. The first element should be a list of the nodes expanded, in the order in which they are expanded. The second element is the final path. See the public test cases for examples.

“See how ridiculous this is? Looking for a flight from San Francisco to Portland, it goes searching the whole rest of the map! Now, most people like single-connection flights, so if we use a breadth-first search instead, it will prefer the short flights.”

“Here, let me show you what it will look like...”

Problem 6: Breadth-First Search

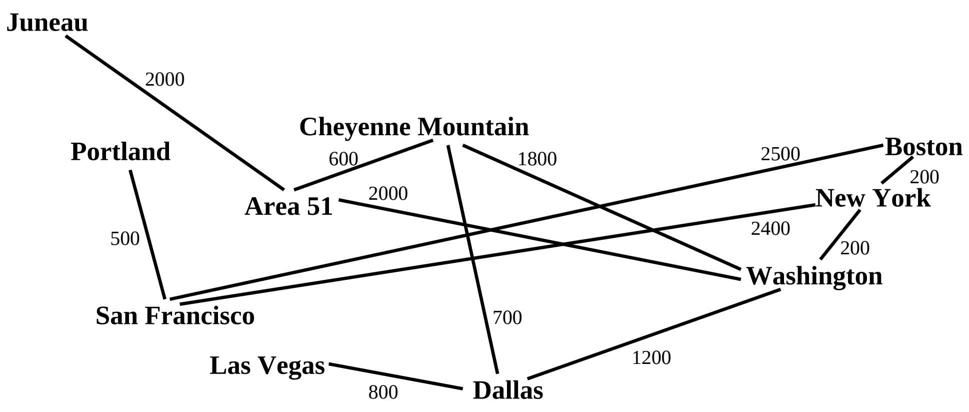

Figure 1: Most popular domestic flights for Black Helicopter Ltd. (used in problems 5-12).

Perform a breadth-first search, with enqueued list, on the six pairs of cities. Record your answers as for problem 5.

Louis wasn't having any of that, though. "But what about finding cheaper multi-stop flights? The breadth-first search doesn't do that. And besides, won't it be horrible for our memory costs?"

"Well," said Ben, "We could do a beam search instead. If we use straight-line distance as a heuristic, that should guide it to the answer pretty quickly. Then it'll be fast because it won't back up, and it will usually get an answer since it's searching more than one option at a time. The only problem is, occasionally it might not find any answer at all."

"Show me," said Louis.

Problem 7: Beam Search

Perform a beam search, with no enqueued list and beam width k=2*, on the six pairs of cities. Record your answers as for problem 5.*

"It always has to produce an answer," said Louis, "Beam search is nifty, but if the search fails, we get in big trouble. And this failed on half the examples!"

"We could do a hill-climbing search, then," said Ben, "It uses the distance heuristic too, but it can back up when it hits a dead end."

"I want to see it," said Louis, growing impatient, "and don't use an enqueued list this time, it's making me confused."

Problem 8: Hill Climbing

Perform a hill climbing search, with no enqueued list, on the six pairs of cities. Record your answers as for problem 5.

"That's ridiculous!" fumed Louis, "It's flying across the country three times to get from Boston to Los Vegas!"

"How about a best-first search, then?" asked Ben, "It's like a breadth-first search, but it chooses the node with the best heuristic value to expand, rather than just the one with the least links in its path. It's sort of like a combination of hill climbing and breath-first search. Here, I'll show you, but I'm going to use the enqueued list again, because otherwise it will be ridiculous..."

OPTIONAL Problem 9: Best-First Search

Perform a best-first search, with an enqueued list, on the six pairs of cities. Record your answers as for *problem 5.*

"I suppose that's better..." said Louis, "but what I'd really like is to know for sure that we're getting the best path."

"Well," said Ben, "what criteria do you have for figuring out which is the best path? It could be number of flights, or it could be length of flights, or any number of things..."

"Money!" said Louis, "It's always about money in the end. But I suppose that distance travelled is a good approximation for money, because the distance determines a lot of the expenses for Black Helicopter Ltd. So we'll use air miles as our distance measure."

"We'll use an optimal search algorithm." said Ben, "Branch and bound with dynamic programming — oh, and we'll add an estimate of distance to the goal and turn it into A*. That should work pretty well — I'll just take these distance figures and add those to the map. Hmm... looks like this needs a little rescaling to work well with your coordinate system, but that's not a big problem..."

OPTIONAL Problem 10: Branch & Bound Search

Perform a branch and bound search, with an expanded list, on the six pairs of cities. Record your answers as for problem 5.

Problem 11: A* Search

Perform an A search on the six pairs of cities. Record your answers as for problem 5.*

"Just one more thing," said Louis, "and it'll be complete. There are special programs that make some of Black Helicopter's flights cheaper to run. Organizations pay a monthly subscription to hold a set of first-class seats available. If they don't use the seats, then Black Helicopter can resell them to standbys. Because of this, two of the flights are significantly cheaper than they appear — the cost of Washington/Cheyenne is effectively 1300 air miles rather than 1800 and Vegas/Dallas is 500 rather than 790."

"Wait a minute!" said Ben, "That's actually a big problem. Now the heuristic isn't admissible any more."

"Does that actually matter?" Louis shot back, "Most of the time that doesn't have any effect, does it?"

"In this case it does. If you plan a path from here to there" said Ben, pointing at two cities on the map, "then A* won't find the optimal path."

Problem 12: Inadmissible Heuristics

Find a two distinct pairs of cities (i.e. four distinct cities) such that each pair causes a problem when using distance estimates and Louis' new values.

Louis sheepishly allowed Ben to correct the heuristic so that it wouldn't overestimate costs, and went trotting off happily with the fix for the route planner, promising Ben yet another complimentary memo. "I'll be back soon with more!" he called over his shoulder as he left.

Problem 13: Survey

- How many hours did you work on this problem set?
- If you were Ben, what grade would you give Louis in your 360-degree performance review?