Chapter 1

1. Every year the Loebner prize is awarded to the program that comes closest to passing a version of the Turing test. Research and report (1/2 page) on the latest winner of the Loebner prize. What techniques does it use? How does it advance the state of the art in AI?

Information on all the recent winners is available here: http://www.loebner.net/Prizef/loebner-prize.html, and related links. The following information on the 2003 winner is excerpted from these web pages:

2003 Winner: Jabberwock, created by Juergen Pirner of Hamburg, Germany. The human judges scored it between “definitely a machine” and “probably a machine”, so it didn’t fool many people. The Jabberwock program is written in RealBasic, which is a version of Visual Basic for Apple Macintosh. The system runs on a Web Server with 4 MB RAM, and 2 MB on disk including a “complete brain file” with 1,200,000 responses. The editing tool for the brain file is a simple text editor. The developers tried to focus on Jabberwock’s skills to be involved in “dialogs” and to be able to follow the topic of a conversation. They allowed up to 5 wild cards in each sentence, and included very large numbers of synonyms in the recognition system. They also developed an “enhanced topic-system” which can handle more than one topic at the same time. They also focused on achieving a varied answer system that works recursively and with a large number of variables and synonyms. They attempted to have Jabberwock never give the same answer twice in a normal conversation. If Jabberwock is not able to follow the conversation or to understand what the user is talking about, it generates random sentences from scratch. It is programmed with 320 jokes, a random story generator, and a basic spell-checking system. (Not many more technical details seem to be available). It seems that the AI advances made by Jabberwock are to be entertaining and funny in a way that seems “off-the-wall”, and at times unexpected. The creators have accomplished this through the incorporation of humorous stories, jokes, and unusual topics, which could be brought up at any time.

Chapter 2

2. Consider the vacuum cleaner agent function described in figure 2.3.

- Show that this function is indeed rational under the assumptions listed on page 36.
  
  It is sufficient to show that for all possible actual environments (i.e., all dirt distributions and initial locations), this agent cleans the squares at least as fast as any other agent. This is trivially true when there is no dirt. When there is dirt in the initial location and none in the other location, the world is clean after one step; no agent can do better. When there is no dirt in the initial location but dirt in the other, the world is clean after two steps; no agent can do better. When there is dirt in both locations, the world is clean after three steps; no agent can do better. (Note: in general, the condition stated in the first sentence of this answer is much stricter than necessary for an agent to be rational.)

- Describe a rational agent function for the modified performance measure that deducts one point for each movement. Does the corresponding agent program require internal state?
  
  The agent in (a) keeps moving backwards and forwards even after the world is clean. It is better to do NoOp once the world is clean (the chapter says this). Now, since the agent’s percept doesn’t say whether the other square is clean, it would seem that the agent must have some memory to say whether the other square has already been cleaned. To make this argument rigorous is more difficult – for example, could the agent arrange things so that it would only be in a clean left square when the right square was already clean? As a general strategy, an agent can use the environment itself as a form of external memory – a common technique for humans who use things like appointment calendars. In this particular case, however, that is not possible. Consider the reflex actions for \([A, Clean]\) and \([B, Clean]\). If either of these is NoOp, then the
agent will fail in the case where that is the initial percept but the other square is dirty; hence, neither can be $NoOp$ and therefore the simple reflex agent is doomed to keep moving. In general, the problem with reflex agents is that they have to do the same thing in situations that look the same, even when the situations are actually quite different. In the vacuum world this is a big liability, because every interior square (except home) looks either like a square with dirt or a square without dirt.

- Discuss possible agent designs for the cases in which clean squares can become dirty and the geography of the environment is unknown. Does it make sense for the agent to learn from its experience in these cases? If so, what should it learn?
  
  If we consider asymptotically long lifetimes, then it is clear that learning a map (in some form) provides an advantage because it means that the agent can avoid bumping into walls. It can also learn where dirt is most likely to accumulate and can devise an optimal inspection strategy. Additional details of the exploration method needed to construct a complete map are given in Section 4.5 of the text. Methods for deriving an optimal inspection/cleanup strategy will be covered in Chapter 21.

3. Consider a modified version of the vacuum environment (from question 2), in which the geography of the environment – its extent, boundaries, and obstacles – is unknown, as is the initial dirt configuration. (The agent can go $Up$ and $Down$ as well as $Left$ and $Right$.)

- Can a simple reflex agent be perfectly rational for this environment? Explain.

  Because the agent does not know the geography and perceives only location and local dirt, and cannot remember what just happened, it will get stuck forever against a wall when it tries to move in a direction that is blocked – that is, unless it randomizes.

- Can a simple reflex agent with a randomized agent function outperform a simple reflex agent? Explain. Describe the design of your randomized agent.

  Yes, because it can avoid the stuck position mentioned above for the simple reflex agent. One possible design cleans up dirt and otherwise moves randomly. This design would be: “If Dirty, then Suck, else randomly move either $Left$, $Right$, $Up$, or $Down$.” This is actually fairly close to what the Roomba commercial vacuum cleaner does (except Roomba only randomizes when it hits an obstacle). It works reasonably well in compact environments.

- Describe an environment in which your randomized agent will perform very poorly. Explain why this environment is bad for your randomized agent.

  The randomized agent will perform very poorly in maze-like environments or environments with small connecting passages; it can take a very long time to cover all the squares. Here is an example environment for which random motion results in poor performance:

4. The vacuum environments above have all been deterministic. Discuss possible agent programs for each of the following stochastic versions:
- **Murphy’s law**: 25% of the time, the *Suck* action fails to clean the floor if it is dirty and deposits dirt onto the floor if the floor is clean. How is your agent program affected if the dirt sensor gives the wrong answer 10% of the time?

For a reflex agent, this presents no *additional* challenge, because the agent will continue to *Suck* as long as the current location remains dirty. For an agent that constructs a sequential plan, every *Suck* action would need to be replaced by “*Suck until clean*”. If the dirt sensor can be wrong on each step, then the agent might want to wait for a few steps to get a more reliable measurement before deciding whether to *Suck* or move on to a new square. Obviously, there is a trade-off because waiting too long means that dirt remains on the floor (incurring a penalty), but acting immediately risks either dirtying a clean square or ignoring a dirty square (if the sensor is wrong). A rational agent must also continue touring and checking the squares in case it missed one on a previous tour (because of bad sensor readings). It is not immediately obvious how the waiting time at each square should change with each new tour. These issues can be clarified by experimentation, which may suggest a general trend that can be verified mathematically. This problem is a partially observable Markov decision process, which we’ll talk about in Chapter 17. Such problems are hard in general, but some special cases may be amenable to careful analysis.

- **Small children**: At each time step, each clean square has a 10% chance of becoming dirty. Describe a rational agent design for this case.

In this case, the agent must keep touring the squares indefinitely. The probability that a square is dirty increases monotonically with the time since it was last cleaned, so the rational strategy is, roughly speaking, to repeatedly execute the shortest possible tour of all squares. (We say “roughly speaking” because there are complications caused by the fact that the shortest tour may visit some squares twice, depending on the geography.) This problem is also a partially observable Markov decision process.

**Chapter 3**

5. Give the initial state, goal test, successor function, and cost function for each of the following. Choose a formulation that is precise enough to be implemented.

- **You have to color a planar map using only 4 colors, in such a way that no two adjacent regions have the same color.**  
  
  Initial state: No regions colored.  
  Goal test: All regions colored, and no two adjacent regions have the same color.  
  Successor function: Assign a color to a region.  
  Cost function: Number of assignments.

- **A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. He would like to get the bananas. The room contains 2 stackable, movable, climbable 3-foot-high crates.**  
  
  Initial state: Same as problem definition.  
  Goal test: Monkey has bananas.  
  Successor function: Hop on crate; Hop off crate; Push crate from one spot to another; Walk from one spot to another; Grab bananas (if standing on crate).  
  Cost function: Number of actions.

- **You have a program that outputs the message “illegal input record” when fed a certain file of input records. You know that processing of each record is independent of the other records. You want to discover what record is illegal.**  
  
  Initial state: Considering all input records.
**Goal test:** Considering a single record, and it gives “illegal input” message.

**Successor function:** Run again on the first half of the records; run again on the second half of the records.

**Cost function:** Number of runs.

Note: This is a *contingency problem*; you need to see whether a run gives an error message or not to decide what to do next.

- You have 3 jugs, measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet. You can fill the jugs up or empty them out from one to another or onto the ground. You need to measure out exactly one gallon.

  **Initial state:** jugs have values [0, 0, 0]

  **Goal test:** jugs have values \([i, j, k]\), where one of \(i, j, k\) is 1.

  **Successor function:** given values \([x, y, z]\), generate \([12, y, z]\), \([x, 8, z]\), \([x, y, 3]\) (by filling); \([0, y, z]\), \([x, 0, z]\), \([x, y, 0]\) (by emptying); or for any two jugs with current values \(x\) and \(y\), pour \(y\) into \(x\); this changes the jug with \(x\) to the minimum of \(x+y\) and the capacity of the jug, and decrements the jug with \(y\) by the amount gained by the first jug.

  **Cost function:** Number of actions.

---

6. **Describe a state space in which iterative deepening search performs much worse than depth-first search (for example, \(O(n^2)\) vs. \(O(n)\)).**

Consider a domain in which every state has a single successor, and there is a single goal at depth \(n\). Then depth-first search will find the goal in \(n\) steps, whereas iterative deepening search will take \(1 + 2 + 3 + \ldots + n = O(n^2)\) steps.

---

**Chapter 4**

7. **A heuristic path algorithm** is a best-first search in which the objective function is \(f(n) = (2 - w)g(n) + wh(n)\). For what values of \(w\) is this algorithm guaranteed to be optimal? You may assume \(h\) is admissible. What kind of search does this perform when \(w = 0\)? When \(w = 1\)? When \(w = 2\)?

- \(w = 0\) gives \(f(n) = 2g(n)\). This behaves exactly like uniform-cost search – the factor of two makes no difference in the *ordering* of the nodes. \(w = 1\) gives A* search. \(w = 2\) gives \(f(n) = 2h(n)\), i.e., greedy best-first search.

To determine the values of \(w\) for which this algorithm is guaranteed to be optimal, note that we can rewrite this function as follows:

\[
f(n) = (2 - w)[g(n) + \frac{w}{2 - w}h(n)]
\]

which behaves exactly like A* search (which is optimal) with a heuristic \(\frac{w}{2 - w}h(n)\). For \(w \leq 1\), this is always less than \(h(n)\) and hence admissible, provided \(h(n)\) is admissible.

---

8. **Prove each of the following statements:**

- **Breadth-first search is a special case of uniform-cost search.**

  When all step costs are equal, \(g(n) \propto depth(n)\), so uniform-cost search reproduces breadth-first search.

- **Breadth-first search, depth-first search, and uniform-cost search are special cases of best-first search.**
Breadth-first search is best-first search with \( f(n) = \text{depth}(n) \); depth-first is best-first search with \( f(n) = -\text{depth}(n) \); uniform-cost search is best-first search with \( f(n) = g(n) \).

- Uniform-cost search is a special case of A* search.

Uniform-cost search is A* search with \( h(n) = 0 \).

9. **Prove that if a heuristic is consistent, it must be admissible. Construct an admissible heuristic that is not consistent.**

A heuristic is consistent iff, for every node \( n \) and every successor \( n' \) of \( n \) generated by any action \( a \),

\[
h(n) \leq c(n,a,n') + h(n')
\]

One proof is by induction on the number \( k \) of nodes on the shortest path to any goal from \( n \). For \( k = 1 \), let \( n' \) be the goal node; then \( h(n) \leq c(n,a,n') \). For the inductive case, assume \( n' \) is on the shortest path \( k \) steps from the goal and that \( h(n') \) is admissible by hypothesis; then

\[
h(n) \leq c(n,a,n') + h(n') \leq c(n,a,n') + h^*(n') = h^*(n)
\]

so \( h(n) \) at \( k + 1 \) steps from the goal is also admissible.

As the text proves, if a heuristic is consistent, then the cost function along any path is non-decreasing. So, any heuristic that is admissible, but which doesn’t have this property, will be an example of an admissible heuristic that is not consistent. Here is an example, in which the starting point is \( s \) and the goal is \( t \):

\[
\begin{align*}
h(s) &= 10; \quad f(s) = 10 \\
h(a) &= 7; \quad f(a) = 1 + 7 = 8 \\
h(b) &= 5; \quad f(b) = 1 + 5 = 6 \\
f(t) &= 11
\end{align*}
\]

10. **On page 108 is the definition of the relaxation of the 8-puzzle, in which a tile can move from square A to square B if B is blank. The exact solution of this problem defines Gaschnig’s heuristic (Gaschnig, 1979). Explain why Gaschnig’s heuristic is at least as accurate as \( h_1 \) (misplaced tiles), and show cases where it is more accurate than both \( h_1 \) and \( h_2 \) (Manhattan distance). Suggest a way to calculate Gaschnig’s heuristic efficiently.**

The misplaced-tiles heuristic is exact for the problem where a tile can move from square A to square B. As this is a relaxation of the condition that a tile can move from square A to square B if B is blank, Gaschnig’s heuristic cannot be less than the misplaced-tiles heuristic. As it is also admissible (being exact for a relaxation of the original problem), Gaschnig’s heuristic is therefore more accurate.

If we permute two adjacent tiles in the goal state, we have a state where misplaced-tiles and Manhattan both return 2, but Gaschnig’s heuristic returns 3.

To compute Gaschnig’s heuristic, start with \( count = 0 \), and repeat the following until the goal state is reached, incrementing \( count \) by 1 for each loop: Let B be the current location of the blank; if B is occupied by tile X (not the blank) in the goal state, move X to B; otherwise, move any misplaced tile to B. The resulting value of \( count \) is Gaschnig’s heuristic.