James Villemarette
CISC 481 Homework 1
Due: March 15, 2022
Extension: March 20, 2022

Problem 1

There is no exercise 2.9 in our textbook, I assume this is problem is referencing a different textbook.

A simple reflex is to quote 2.4.2, “the simplest kind of agent […] select[ing] actions on the basis of the current precept, ignoring […] percept history” (Russel & Norvig 121). The example given in the textbook looks like

```
Function reflexVacuumAgent(location, status)
      If status == dirty then return Suck
      Else if location == A then return Right
      Else if location == B then return Left
```

The pseudocode for the thermostat would look like

```
Function reflexThermostat(setting, temperature)
      If temperature < setting – 3 then TurnOnFurnace
      Else if temperature > setting + 3 then TurnOffFurnance
      Else DoNothing
```

A model-based reflex agent would be stateful, moving between states based on what it *can* observe to keep track of things it can’t observe, or things it needs to remember.

A goal-based agent is the “sub class” of a model-based agent, where it not only has states, but goals and some element of path finding to that goal.

A model-based reflex and goal-based agent can both probably solve this problem.

But I argue it is best to solve this problem with the simplest option that we have available to us that completely solves it—a simple reflex agent. It might be nice to have a goal-based agent that can possibly “learn” from the ebbs and flows of the environment, maybe the temperature outside, and any possible energy saving goal. But as the problem is described, a simple reflex agent nicely solves the problem. My pseudocode above even looks almost example like the pseudocode provided in the textbook.

Problem 2

I am referencing Chapter 3.1.1 and 3.2.1 as an example for solving this problem.

Define the problems

- NxN grid of squares
  - States = N * N * 3
    - For the N * N grid, there is a state for at least each square
    - There are 3 states for every square:
      - Unpainted,
      - painted,
      - and bottomless pit
    - Ex: State N[0, 0]unpainted; N[0,0]painted; N[0,0]bottomlessPit; …
  - Initial State = Any state that is not
  - Actions =
    - Paint
    - Upward
    - Right
    - Downward
    - Left
  - Transition Model =
    - Paint: Pain the square that the agent is located in
      - Can only do this if the current square is unpainted and not a bottomless pit
    - Upward: Move the agent forward a single floor square
    - Right: Move the agent right a single floor square
    - Downward: Move the agent backward a single floor square
    - Left: Move the agent left a single floor square
  - Goal States = When all non-bottomless pit tiles are painted
  - Actions cost = Each action cost 1
- Container ship in port
  - States =
    - Objects are the containers and the crane
    - Each container can either be on the dock or the ship
  - Initial State = 13 x 13 x 5
    - There's the 13 x 13 grid on the ship, times the 5 levels that a container could be at
    - There are 0 containers on the dock
  - Actions =
    - GrabContainer
    - DropContainer
    - CraneDriveForward
    - CraneDriveBackward

        - CraneDriveRight
        - CraneDriveLeft
        - CraneLiftUpward
        - CraneLiftDownward
        - CraneMoveToDock
        - CraneMoveToShip
    - Transition Model =
        - GrabContainer: Can pick up a container onto the crane from the ship or the dock, depending on where the crane is (affected by CraneMoveToDock and CraneMoveToShip)
            - Can only grab a container if the crane arm is in the position above a container
        - DropContainer: Releases a container (if holding on) to wherever the crane is (dock, or ship position)
            - Can only release a container if the crane arm is holding a container
        - CraneDriveForward: Drive the crane position forward (horizontally) one position
        - CraneDriveBackward: Drive the crane position backward (horizontally) one position
        - CraneDriveRight: Drive the crane position to the right (horizontally) one position
        - CraneDriveLeft: Drive the crane position to the left (horizontally) one position
        - CraneLiftUpward: Move the crane arm up one level (vertically)
        - CraneLiftDownard: Move the crane arm down one level (vertically)
        - CraneMoveToDock: Move the crane to the dock yard
        - CraneMoveToShip: Move the crane to the ship
    - Goal States = {}
    - Actions cost =
        - The following actions cost 1 (since they're going just one space)
            - LiftCont
            - CraneDriveForward
            - CraneDriveBackward
            - CraneDriveRight
            - CraneDriveLeft
            - CraneLiftUpward
            - CraneLiftDownward
        - The following actions may cost 2 or more (since it takes longer to move across the dock to the ship, rather than just one position within the dock or the ship)
            - CraneMoveToDock
            - CraneMoveToShip

Problem 3

Define the problem

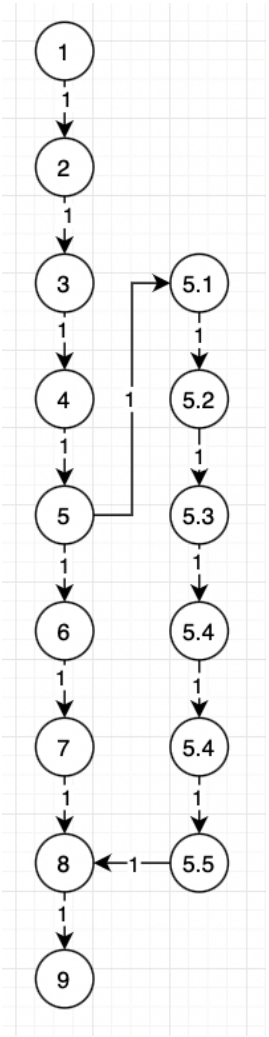
- States
  - 1: Fox, Blue Hen, Corn, and Person are all at South Campus
  - 2: Fox and Corn at South; Person carries Blue Hen to Laird
  - 3: Fox and Corn at South; Person bus back to South; Blue Hen at Laird
  - 4: Fox at South; Person carries Corn to South; Blue Hen at Laird
  - 5: Fox at South; Person carries Blue Hen bus back to South; Corn at Laird
  - 5.1: Fox at South; Person carries Corn back to South; Blue Hen at Laird
  - 5.2: Corn at South; Person carries Fox to Laird; Blue Hen at Laird
  - 5.3: Corn at South; Person carries Blue Hen back to South; Fox at Laird
  - 5.4: Blue Hen at South; Person carries Corn to Laird; Fox at Laird
  - 5.5: Blue Hen at South; Person bus back to South; Fox, Corn at Laird
  - 6: Blue Hen at South; Person carries Fox to Laird; Corn at Laird
  - 7: Blue Hen at South; Person bus back to South; Fox, Corn at Laird
  - 8: Person carries Blue Hen to Laird; Fox, Corn at Laird
  - 9: Fox, Blue Hen, Corn, and Person are all at Laird Campus
  - Summarized
    - The objects are
      - The agent
      - The fox
      - The hen
      - The corn bushels
    - The locations are
      - South Campus
      - Laird Campus
    - The states are all of the possible combinations of them
    - Hint: You have to carry some objects back from Laird to South campus, or risk having a hen or corn devoured
    - I thought about this for a long time, and I can't think of any other states or unique combinations of actions to take
- Initial State
  - State 1 (everyone at south campus)
- Actions
  - 1: Carry Fox to Laird Campus
  - 2: Carry Blue Hen to Laird
  - 3: Carry Corn to Laird
  - 4: Carry Fox to South Campus
  - 5: Carry Blue Hen to South
  - 6: Carry Corn to South
  - 7: Travel to Laird empty handed
  - 8: Travel to South empty handed

- Transition Model
    - It would be painful to describe all the actions here and how the transition model applies to them. Also, the effects are obvious
        - Carry X to Y campus results in X being at Y campus, and our person (ourselves) being at Y campus
        - Travel to Y campus empty handed results in our person (ourselves) being at Y campus
- Goal States
    - State 9
- Action cost
    - Each action costs 1

Which uninformed search strategy would I recommend, and why: I would use Dijkstra's algorithm, since it's efficient and we've used it before in other classes. It is uninformed, because it does not have a specific target node

Continued on the next page because this was becoming too long…

Trace search tree produced by our search method, assuming a graph search implementation (prune repeated states):



I'm not going to give the full distance, previous node, and visited tables break down for every step, because that would be tedious with all of the states. Suffice to say, this is what the table would look like at the end (excluding 5.x because that's obviously a bad path)

Node	Distance	Previous	Visited
1	1	Null	True
2	2	1	True
3	3	2	True
4	4	3	True
5	5	4	True
6	6	5	True
7	7	6	True
8	8	7	True
9	8	8	True

With Dijkstra, to find the path from goal to start, we just take our goal node and trace the previous back up.

From Node 9, we look at its previous, which is 8. Then 8's previous is 7. So and so forth until we build a path to 1 which looks like:

1 —> 2 —> 3 —> 4 —> 5 —> 6 —> 7 —> 8 —> 9