Greg Yeutter
January 30, 2015
CS510 HW2
Part 1: Written Problems

1.A: Forward-Backward Search

To effectively solve this problem, one must consider the branching factor of the forward search vs. that of the backward search. In this case, searching the ancestral tree from Ben Franklin would be in the forward direction and searching from Eloise would be in the reverse direction. The branching factor is likely to be smaller if the search begins with Eloise (backward), because a smaller number of nodes (parents, grandparents, etc.) lead to Eloise, each with a branching factor of 2. This is likely smaller than if we consider descendants of Ben Franklin. Especially in those times, individuals were likely to have many children, each resulting in a branching factor that is likely greater than two. Because of this, over generations, the search tree is likely to get very wide.

If Eloise's age is known, a depth bound can be established to terminate the search in either direction, in case the goal has not been found. Because of this, a forward search should not take too much longer than a backward search, but since the backward tree is likely narrower, it is still easier to search in the reverse direction.

1.B: Uninformed Search with Negative Action Costs

a. An optimal algorithm is one that finds the least-cost solution. If actions can have negative step costs, this means that taking an additional step reduces the path cost, and taking many steps makes the path cost much lower. Since the optimal solution is the least-cost solution, the algorithm will search the entire space because exploring every state will result in the lowest path cost (assuming all values are negative). Even if only some values are negative, it is plausible that the algorithm would still attempt to search the entire space, as exploring another node may or may not reduce the path cost.
b. Even if a global minimum for negative step costs is set, the algorithm may still explore the entire space as each node may reduce the path cost.
c. If a set of actions with negative step costs form a loop, the path cost will decrease with each iteration and run forever, since the optimal algorithm seeks to minimize the path cost.
d. Humans do not drive around in scenic loops indefinitely, as doing so in a problem with all positive step costs would increase the path cost. This problem could be modeled so that more than distance, time, and fuel are considered. For example, desirable routes may have lower (but still positive) step costs than more efficient but less-scenic routes. This way, artificial agents may choose the scenic route over the most efficient one, based on the path cost of all the steps, and also avoid looping.

1.C: Backtracking

To find all solution paths, an empty list of solutions is created, as well as an integer starting at 0. If a path is found, the current solution is stored in position i of the solutions list, and i is incremented for the next solution. The next time a solution is found, it will be appended to the next-highest position in the list, and the process will repeat until no more moves are possible.

```
new List solutions
int i = 0
backTrack(stateList , depthBound) {
  state = first element of stateList
  if state is a member of the rest of stateList, return FAIL
  if goal(state), return NULL
  if length(stateList) > depthBound, return FAIL

  moves = getPossibleMoves(state)
  for each move m in ruleSet {
    newState = applyMoveCloning(state,m)
    newStateList = addToFront(newState,stateList)
    path = backTrack(newStateList, depthBound)
    if path != FAILED solutions[i].append(path,m), i++
  }
}
```

1.D: Baskets of Marbles

a. Using the given backtrack algorithm with a depth limit of 3 and an initial state of (6,5,1), a solution was found. The first move on the initial state (-1,0,1) failed because each following move (-2,0,2) and (0,-2,2) failed, as no solution was found down this path. Starting back at the initial state, the move (0,-1,1) was performed, followed by (-2,0,2) to get the final state (4,4,4).

Total failures: 3 (from state (5,5,2) cannot complete (-2,0,2) or (0,-2,2): +2 failures;
Backtrack to (6,5,1) for +1 failure on the first move)

Calls to backtrack: 6

Path: [(6,5,1); null] [(6,4,2),(6,5,1); (0,-1,1)] [(4,4,4),(6,4,2),(6,5,1); (-2,0,2)]

b. The results for a depth bound of 5 are identical to the previous problem, since the three failures are due to an inability to move any further and not due to exceeding the depth limit. If the solution required one more move, however, the depth limit of 3 would have been exceeded.

c. If multiple solutions are allowed via the algorithm in problem 3, two solutions are given. These are:

Path 1: [(6,5,1); null] [(6,4,2),(6,5,1); (0,-1,1)] [(4,4,4),(6,4,2),(6,5,1); (-2,0,2)]

Path 2: [(6,5,1); null] [(6,4,2),(6,5,1); (0,-1,1)] [(6,2,4),(6,4,2),(6,5,1); (0,-2,2)] [(4,4,4),(6,2,4),(6,4,2),(6,5,1); (-2,2,0)]

Failures increase because the last possible move has a state already in stateList (6,4,2), which would lead into a loop.

Total failures: 6 (In addition to the 3 from before, +1 when (6,4,2) encountered, which propagates back up the hierarchy +2)

Calls to backtrack: 9

d. For depth-first search, six nodes are expanded. The path is the same as that found in a and b.

Path: [(6,5,1); null] [(6,4,2),(6,5,1); (0,-1,1)] [(4,4,4),(6,4,2),(6,5,1); (-2,0,2)]

Failures: 3

e. For breadth-first search, six nodes are expanded before a solution is found.

Path: [(6,5,1); null] [(6,4,2),(6,5,1); (0,-1,1)] [(4,4,4),(6,4,2),(6,5,1); (-2,0,2)]

Failures: 3

This path is the same as that found in a, b, and d. For other initial states, the solution and/or number of nodes expanded may change.