CS170–Fall 2015 — Homework 5 Solutions

Nguyet Minh Duong, SID 24444722

Collaborators: Henry Kwan & James Carlson

1. Adroit Infiltrator

1. Main Idea
   We rebuild the map. For the corridors with more than one droid, we place (number of droids in the corridor - 1) "dummy" rooms in the corridor, where the corridor connecting each room together has only one battle droid. Then we run a revised breadth first search algorithm that when it finds an edge with a weight of 0, the room that the edge leads to is now in the front of the priority queue. We continue the algorithm and continue to search through the entire revised map until we reach the room that is considered to be the server room. From there, we start recording all the rooms we have passed but not considering the rooms that are dummy – hence we only record the original rooms that appeared on the original map.

2. Pseudocode

```
// C <- list of corridors
// R <- list of rooms
// e <- entrance room
// s <- ending room
Procedure Infiltrate(C, R, e, s):
  M := new blank map
  for each room in R:
    rm := create a new room in M
    rm.original() := true
  for each cor in C:
    if cor.numberOfDroids = 0:
      cr := create a corridor in M
      cr.start() := cor.start()
      cr.end() := cor.end()
      cr.numOfDroid := 0
    else if cor.numberOfDroids = 1:
      cr := create a corridor in M
      cr.start() := cor.start()
      cr.end() := cor.end()
      cr.numOfDroid := 1
    else:
      DR := list of dummy rooms
      create (cor.numberOfDroids - 1) rooms, and insert into DR
```

```
      for each droom in DR:
        droom.original() = false
      connect the dummy rooms with corridor, all with numOfDroid := 1
      r1 := dummy room such that r1.start() is null
      r1.start() := cor.start()
      r2 := dummy room such that r2.end() is null
      r2.end() := cor.end()
    BFS_Revised(M, e, s)

Proceedure BFS_Revised(map M, room e, room s):
  R := set of all rooms in map M
  C := set of all corridors in map M

  Q := a queue that contains rooms
  Q.insert(e) // put the room e inside the queue
  while Q is not empty:
    rm := Q.pop()
    // assume pop() removes the first element in the queue, and returns it
    for all cor in C, such that cor.start() is rm:
      s1 := cor.end()
      if s1 is e:
        s1.previous() := rm
        exit the while-loop
      if cor.numOfDroids is 0:
        // assume firstInsert(s1) will insert s1 into the front of the queue
        Q.firstInsert(s1)
        s1.previous() := rm
      else:
        s1.previous() := rm
        Q.insert(s1)
  paths := empty list of rooms
  while s is not e:
    if s.original():
      paths.insert(s)
    e := e.previous()
  paths.insert(e)
  paths.reverse() // reverses the list paths
  return paths
```

3. Proof of Correctness
   In the first step, we simply just created a map with small revisions in it. And for the second procedure, we revised how breadth first search should be run. We removed the distances as we just want to go through the Map in a breadth first search manner. We go through all points within the queue, similar to how we do in the original breadth first search. In our breadth first search, we will record the previous room that the corridor takes us down instead of its distance. The main idea of what we changed within this is that if the corridor's (call it C_1) number of droids is zero, we place the room (call this R_1) the C_1 leads to in front of the queue. The reason we do this is because we should consider R_1 as the same thing as the room C_1 starts

at since there are no droids in the way – making it the better path to walk. It is our way of merging the two rooms together while recording the path.

The addition of other rooms that does not satisfy this condition will be added into the queue the same way the original breadth first search does, and so it should run through the same way. But the algorithm search should end when we reach our server (ending room) because we will have gotten all the paths that it requires to get there. When we break out, we can traverse all the previous rooms starting for the server room. We record this into a list if the room is considered as an original, and then reverse it. The result will give us a list, where the first element is the starting point and the last element is the ending point, and everything in between are the rooms we should go through in order to get to server such that we pass the least number of droids.

4. Running Time Analysis
$O(|R| + |C|)$

When we create our new map, we look at every single room and create it. This is $O(|R|)$. Then when we look at each corridor to add the new rooms and corridors in between, this will take $O(|C|)$ to look through. Since our algorithm is based off of breadth first search, where the loop is the same thing as the loop within breadth first search, the running time is the same: $O(|C| + |R|)$. Then when we add all of these values together, and remove the constants, we get $O(|R| + |C|)$.

2. Galaxy Quest

3. Better Bartering

(a) 1. Main Idea

This algorithm is recursive, such that we remember the best possible path where the best possible path has the highest trade value. At the end of the algorithm, we will return the path that has the highest trade value as the answer.

2. Pseudocode

```
dt := a dictionary with its keys as valuables
      and its values as a tuple (best possible ratio trade, path)
Procedure BestTrade(list V, s, t):
  if s is t:
    dt[s] = (1, s)
  else:
    max := initially 0, represents the max possible valuable trade
    p := a list of valuables that represents the path of the trades
    for all x in V not s:
      R := ratio of i to x
      V' := V without s
      if dt[x] exists:
        if dt[x][1] * R > max:
          max = dt[x][1] * R
          P = makeList(i, dt[v][2])
          // makeList(p1, p2) will attach p2 to the end of p1
      T = BestTrade(V', x, f)
      if T[1]*R > max:
        max = T[1]*R
        P = makeList(i, T[2])
    dt[1] = (max, P)
    return dt[1]
```

3. Proof of Correctness

This is a proof by induction, on N, the number of valuables, with V_1 and V_2 in the such we find the best possible ratio going from V_1 to V_2.

Base case: $N = 1$, this case is trivial where we're comparing it to itself, so the ratio is obviously 1.

N = 2, assuming we want to go to two different valuables, that means we want to go from V_1 to V_2. This means we need to start at the root, in this case V_2, and run the base case $N = 1$ on that, and then multiply the ratio between V_2 to V_1. This will give us the ratio in trading.

Note that we store all of these values in the dictionary, such that the ratio stored is the biggest one possible, along with the path that goes along with getting the biggest ratio.

Induction hypothesis: Assume this works for $N \leq K$, where it will return to us the ratio to go from V_1 to V_2. You will also store the biggest ratio with the correct valuable along with the path to get there.

Induction step: For $N = K + 1$. We know that our dictionary has already stored all the best possible paths such that we get the best ratio trade values, starting from V_1 to V_2. So when we add in the additional valuable, we add into another case that the initial value has to compare. In this case, everything it comes to below it is already written within

the dictionary. If we stick the new values randomly in anywhere, it will always the path below and we can pull from dictionary, then update all the values above it if it satisfies the conditions.

4. Running Time Analysis
   $O(n^2)$

   Essentially we are trying to find all the possible paths we can take, hence we make a fully connected graph. Given N nodes, we can make approximately N^2 edges. As a result, we're going to traverse that amount of edges. Hence it gives a $O(n^2)$ algorithm.

(b) 1. Main Idea
   We will find all the edges that has the trade value, meaning that the value is 1 or higher. Allow the trade the happen, and then find the best possible route from the resulting trade back to the original item we started out with. Then this will give us the best return rate.

2. Pseudocode

```
Procedure GetMoney(list V):
  M := list of all trades such that the ratio is higher than 1
  if M.size() < 1:
    return "Not Possible"
  max := initially 0, the maximum profit
  path := initially empty,
      list of valuables in order such that
      it creates a path that is a beneficial trade
  for all t in M:
    T = BestTrade(V, t.end(), t.start())
    // start() gives us the valuable with start the trade with
    // end() gives us the valuable we receive at the end of the trade
    if T[1] > max:
      max = T[1]
      path = makeList(t.start(), T[2])
  return (max, path)
```

3. Proof of Correctness
   We store all of the trades that will give us. This is trivial as we just run through all of the trades and find trades that is greater than a 1 ratio. If there isn't any, then we can't find one that makes us profitiable. Otherwise we set our max to be initially 0, so that any trades will be good. We create a path variable so we can remember how to go through the trades.

   Given the trade, we should return the tuple that BestTrade() gives us. We know BestTrade works. We run it on the set of valuables V, and we start at the end of the trade because we want to eventually loop back to the valuable we originally started with (we want to get it back but for more). What this means is that if the max is ever higher, we get the better trade off. And if the trade route is better, we should update our values. We attach the starting position to the ending path we get from running BestTrade().

   After traversing through all the possible routes for all of the trades that are beneficial,

the highest one will return as the max and path values will only change if it satisfies the conditions above.

4. Running Time Analysis
   $O(n^4)$

   In our worst case, every single edge we have is a positive. That means we have to run through all of it. As mentioned in part A, we create a fully connected graph. In order to do this, it takes n^2 run through which gives us $O(n^2)$. But because for every single run we do, we call our previous method, we have to multiply the two together, which ultimately gives us $O(n^4)$.

4. Road Construction, with the Force

1. Main Idea
   We run a revised version of Dijkstra's algorithm. In our algorithm, we have four different cases: we went through more than broken road, we went through only one broken road, we didn't through any broken road, and we didn't traverse the road to the end point at all. Given this, we have a priority of what to look for. We will choose the one that has the higher priority. We will have one for broken roads, and non-broken road.

2. Pseudocode

```
// inf := infinity value
// ninf := negitive infinity value
// undef := undefined
Procedure FindPath(R, R', I, s, t):
  P := list that has the union of R and R';
       elements input into P can be asked if it is "broken()" or not

  s.broken = (0, False)
  s.n-broken = (0, False)
  s.prev := undef
  s.prev-b := undef

  Q := a list, initially empty, that will hold endpoints
  for each i in I:
    if i is not s:
      i.broken = (inf, False)
      i.n-broken = (inf, False)
      i.prev := undef
      i.prev-b := undef
    add i to Q

  while Q is not empty:
    u := endpoint that has the lowest broken-value
```

5. Ewok Celebration