CS170–Fall 2015 — Homework 8 Solutions

Nguyet Minh Duong, SID 24444722

Collaborators: Henry Kwan & James Carlson

1. Coconut Programming

1. Main Idea
   As long as there are more than one coconut, do binary search type of dropping on the coconuts to narrow down the range of floors in which the coconut will break. If there is only one coconut left over, start from the bottom floor that we know that hasn't broken yet and go upwards, as this is the best we can do with only one coconut.

2. Pseudocode

```
Procedure CoconutBreak(n, h):
  f := 1, indicates the highest floor that the coconut hasn't broken on
  while n > 0 and f.isNotEqual(h):
    if n == 1:
      w = n.pop() // removes a coconut from our bundle of coconut
      drop(w, f) // drops coconut w at height f
      if w.isBroken():
        return f
      else:
        f++
    else:
      mid := findMid(f, h) // find the middle floor between f and h
      w = n.pop()
      drop(w, mid)
      if w.isBroken():
        h = mid
      else:
        f = mid + 1
  return 0
```

3. Proof of Correctness
   If we only have one coconut, the only way we can know with certainty at which floor it starts breaking from is to go from the last floor we know that the coconut has not broken. Otherwise, we can always reduce the amount of floors we have to search through by half if we drop it at the half mark between the lowest floor we know that the coconut was broken from the drop and the highest floor we know that the coconut was not broken from the drop. That means we are

testing it at the halfway mark, and by doing that, if the coconut breaks, our lowest-floor-broken record is changed. Otherwise our highest-floor-not-broken is updated. EIther way it cuts the amount of floors we have to look through by half. And we know with certainty that the floors we did not check are all floors we know as a fact that it will break or it will not break by the condition of the problem where if the drop is higher than the floor the coconut will break, the coconut will definitely break.

4. Running Time Analysis
   $O(h)$

   In the worst case scenerio is if we had two coconuts and we drop it in the halfway mark, and it breaks. Then we have to climb half of the floors to check at which floor it breaks on. In the worst case senerio of that situation, we have to climb all half of the floors before we find the floor that it broke on. This is $O(h/2)$, which is $O(h)$. But with more coconuts, the algorithm gets significantly better as we have more tries.

2. All Possible Marriages

(a) We manually create all the possible pairings by looking through every man, and woman's preference. We know to find all the possible preferences, we must multiply it against all the previous possibilities. This is from counting, however we know the last person can only get one person. The second one has two choices (at most), etc. etc. By testing it out to see every possibilities, it takes us $O(n(n!))$ time.

(b)

3. Boundedness and Feasibility of Linear Programs

(a) $a \leq 3, a \geq 4$

(b) $a = 1, b = -1$

(c) $a = \frac{1}{3}, b = 1$

4. Star-shaped Polygons in 2D

(a) Go through the list of points P. Create an inequality segment between P_t and P_{t+1}, where $t \in [1, 2, ..., n-1]$ through the line equation by using the x's and y's given, as well as which side the segment corresponds to. At the end of this, we should be given n-constraints. Then we can choose to either maximize or minimize the sum of all the variables created for each inequality. If simplex returns an answer, the polygon is star-shaped. Otherwise the polygon is not a star-shaped.

(b) The reason why it doesn't matter whether or not it is min or max is because it requires all constraints not to be infeasible. This means that all of the segment-constraints' sides overlaps at a certain point. This is the point that the problem is talking about, in which it "sees" all of the segments – which are the boundaries in this case. Meaning if simplex cannot return an answer, the constraints were either too tight, and there is no point that can be within all of the segments. Therefore there is no point that can see all of the points. Otherwise the constraints weren't bounded tight enough, and as a result, it is not a star-shaped polygon because there are no actual boundaries.

5. Max-flow Variants

(a) The vertex-capacity constraint can be thought of as another flow/edge constraint. So instead of having it inside of the edge, we make a flow coming out of it maximum is its edge capacity. What this does is guarentee us to know the maximum to come leave. How we do this is by creating another vertex next to the every single vertex with a capacity-constraint. The flow from the original to the new one is the capacity-constraint. All of the original outgoing flow/edges from the original vertex is now in the new vertex.

By doing this, we set a constraint on how much it can leave in the first place, making the ending result to get to the sink the same. Essentially if G is our original graph with the capacity, and the route it will take is F. Then the graph we have created is an version of it, call it G'. When we run Max-Flow algorithm on G', it will return to us the flow F'. It can be claimed that the flow F is the same as the flow F'. The reason for this is because we have new flow restrictions to keep in mind. Our pathing using Max-Flow will now take in consideration of the max output flow into finding the max-flow. And as a result, it will choose the path that will return the highest value, which should still be the same as F.

(b) A way to reword this problem is what is the maximum output of the flow? The best way is to solve this like any other max-flow problem with max flow. Because it is connected in some way to their sinks, we should think of this as an entire body of flow. Then what we have to do is solidify the graph by creating another source that has outgoing edges to all the other sources, and create another sink where all the other sinks flow to. The new flows have no constraints – it can take as little or as much as possible.

The flow, F, in the original graph G should be the same as the newly created graph G' because max-flow on the revision G. The reason for this is because the new source node will determine itself what is the maximum it should flow to the other nodes in order to satisfy at maximum what the new sink node can take. This allows our algorith to run through and figure out what is the maximum we can get to the new sink node, which, as a result, is the maximum flow output there can be since it is a connected system – so we can't just run max-flow on each and add up all the values at each sink. By integrating it with each other, we are able to see how every flow affect each other but not hold back by the constraints.

(c)

6. Dealing Cards

(a) 12, 892, 2, 9, 1, 4 is not optimal for the first player if he chooses the greedy algorithm because he'll start off with 12 but then the second player will always be able to pick the most optimal value (highest of the entire set) every time while the first player does not have a choice to.

(b) 1. Main Idea
Start from the two main possibilities, whether starting from the first card or the last card. From there, choose it based on the value with the biggest return value. In every ordering there is a certain max value. And we want to return, and continously choose these values. How we do this is by having a dictionary with each possible outline that stores the possible net points we'll have, and how we got there. We return that ordering to the player 1.

2. Pseudocode

```
Procedure BestCase(C):
  cVal := initially empty dictionary, where the
    key is the variation of the cards
    values is a pair: [max_value_in_move, move]
    where max_value_in_move tells us what our net point is and
    move is whether we chose the first or last card
  FillCVal(C)
  return cVal
Procedure FillCVal(C):
  if C.isLastCard(): // returns true if it is the only card in the set
    pos = C.cardPosition() // returns the arrangement/positions of the card(s)
    val = C[1].val() // returns the value of the card
    cVal[pos] = pair(val, null)
  else:
    pos = C.cardPosition()
    CL = C.withoutFirst() // list of cards without the card on the left
    CR = C.withoutLast() // list of cards without the card on the right
    CLVal := initially 0, holds the value of net points if choose left
    CRVal := initially 0, holds the value of net points if choose right
    if cVal(CL.cardPosition()).exist():
      CLVal := cVal(CL.cardPosition()).max_value_in_move
    else:
      FillCVal(CL)
      CL := cVal(CL.cardPosition()).max_value_in_move
    if cVal(CR.cardPosition()).exist():
      CRVal := cVal(CR.cardPosition()).max_value_in_move
    else:
      FillCVal(CR)
      CR := cVal(CR.cardPosition()).max_value_in_move
    if CR > CL:
      cVal(C.cardPosition()) = pair(C[last].val() - CR, last_card)
    else:
      cVal(C.cardPosition()) = pair(C[1].val() - CL, first_card)
```

3. Proof of Correctness

This algorithm works because we are building from the bottom up. Essentially, we are looking at the smallest case possibility. And we store the highest value that is given. From there, we slowly move up, basing our choice on the previous value. In the net points, it is always the person's previous pointversus what we can get. So our net gain is the value we get picking it up, subtracted from the best case in the previous.

By choosing whichever one is the best in the situation and storing it. The person can just glance at the dictionary, and see which (left or right) card to pick in order to place in his best case scenerio since we have drafted all card position possibilities!

4. Running Time Analysis
   $O(n^2)$
   The reason that it is $O(n^2)$ is because we look at all the possible arrangements of the cards if we pick either the leftmost card or the rightmost card. In doing so, we break down the card positions into its simplest terms – in this case it is n cards. Then from there, for ever single n-cards, there is two variation above it (whether or not we picked the leftmost or the right-most). In this case, it creates n$^2 possibilities, andeachcalculationstakes$O(1)$time. As a result, weget$
   algorithm.

7. Existence of Perfect Matching

It is important to define what a perfect matching – to make things simpler, think of it as there being two equal list on the left and the right. There is a men list, and a women list. A perfect pairing is essentially saying that we can somehow match people up (male to female) in such a way that every person can be with another person, and every pairing is unique.

By creating all possible sets on the men's side (Left), and seeing how many edges go over to the women's side (Right), by seeing how many unique females the set of males are attracted to, we are able to deduce whether or not a pairing is possible. Essentially, if there isn't enough connected edges between how many males there are to how many unique females there are – we cannot create a perfect matching in that subset because what it is essentially saying is all the males in the subset likes similar female in such a way that there are more males than females. And since we can't share... it cannot create a valid pairing.

Similarly, reversing it also creates the same situation to which we can use the previous statement to prove it. We cannot "share" the same male if it ends up that a certain amount of females like a smaller subset of males. This means that they have to share, in which it is not a perfect pairing.