Advanced Algorithms Assignment 1

Anthony Medico

Question 1

a) Optimal Substructure

- Consider the fastest way possible to get from the start through stop 1 at step i (s[i][1])
- If i = 1, there is only one solution: e[1] + a[1][1]
- For step i = 2,…,n, there are k possibilities (number of stops at each step) at each step:
    - The previous destination was one of the k stops at step i – 1
    - Total time = time required to get from start through one of the k stops at previous step,
+ travel time from that stop to stop 1 at step i (s[i][1]),
+ time to perform the task at s[i][1]
- Similarly for stop 2,…,k
- So, if the fastest way to get from the start through stop 1 at step i (s[i][1]), is through s[i-1][j], where j = 1,…,k, then we must have taken the fastest route from the start through s[i-1][j]
- If there was another faster way, we would have calculated and recorded it when calculating fastest times at step i – 1
- Similarly for stop 2,…,k
- Therefore the optimal solution to the overall problem must contain within optimal solutions to subproblems, and exhibits optimal substructure

b) Recursive Definition

- Let min_times[i][j] denote the minimum time to get to stop s[i][j]
- Let f* denote the minimum overall time to travel from the starting location to the final destination:
    - f* = min(min_times[n][j] + x[j]), where j = 1,…,k
- Recursive definition for values of min_times[i][j]:
- min_times[i][j] = e[j] + a[1][j] if i = 1
min(min_times[j'][i-1] + t[i-1][j'][j] + a[i][j]) otherwise
    - where j' is the stop at step i – 1 (1…k)

c) Dynamic Programming Algorithm

```
# Ranges are inclusive at both endpoints.
# This pseudo code is not using 0-indexing, but the
# corresponding program does and handles it
# appropriately.

# mt = minimum times
# bmt = best minimum time
# bps = best previous stop
# cmt = current minimum time
# ps = previous stops (to get overall path)

# DP Algorithm
for j = 1 to k
  mt[j][1] = e[j] + a[1][j]

for i = 2 to n
  for j = 1 to k
    bmt = infinity
    bps = 0
    for j' = 1 to k
      cmt = mt[j'][i-1] + t[i-1][j'][j] + a[i][j]
      if cmt < bmt
        bmt = cmt
        bps = j'
    mt[j][i] = bmt
    ps[j][i] = bps

f_star = infinity
s_star = 0

for j = 1 to k
  cmt = mt[j][n-1] + x[j]
  if cmt < f_star
    f_star = cmt
    s_star = j

# To construct route
stops.push(s_star)
j = s_star

for i = n to 2
  j = ps[j][i]
  stops.push(j)

for i = 1 to n
  print "Step {i}: stop {stops.pop()}"
```

Question 2

a) Optimal Substructure

- If ($t_1 \ldots t_k$)($t_{k+1} \ldots t_n$) is an optimal shift split point, then the split points within ($t_1 \ldots t_k$) and ($t_{k+1} \ldots t_n$) must be optimal too.
- If not, then there is an alternate set of split points that would produce a lower overall cost.
- The two subproblems also do not affect each other in any way.
- If either alternate set of split points did produce a lower cost, then you could just substitute them in to get a resulting lower overall cost, which is a contraction.
- Therefore, the subproblems must also be optimal.

b) Recursive Definition

- Let m[i][j] be the minimum cost of assigning tasks $t_i \ldots t_j$
- The cost of the cheapest overall solution is m[1][n]
- If i = j: m[i][j] = 1000
- If i < j: m[i][j] = min(m[i][k] + m[k+1][j], c[i][j])   (for some k)
    - where c[i][j] is the cost of scheduling tasks i…j in one shift
        - c[i][j] = infinity if they don't all fit in 1 shift
- Find the best value of k, which represents an optimal split point
- If c[i][j] is the best overall k, then there is no split point (it is instead optimal to have all tasks within 1 shift)

c) Dynamic Programming Algorithm

```
# DP algorithm

for i = 1 to n
  m[i][j] = 1000

for t = 2 to n
  for i = 1 to (n-t+1)
    j = i + t - 1
    m[i][j] = infinity
    for k = i to j-1
      q = min(m[i][k] + m[k+1][j], c[i][j])  # see cost function below

      split_point = 0
      if m[i][k] + m[k+1][j] < c[i][j]
        split_point = 0

      if q < m[i][j]
        m[i][j] = q
        s[i][j] = split_point
```

```
# Cost function to populate c[i][j]
# Calculates cost from task i to task j
# Returns cost if the tasks fit in 1 shift, or infinity otherwise

# num_gaps = number of gaps required
# req_time = total required time for all tasks including minimum gap of 1
# base_gap_time = splitting total gap time evenly (remainder not included)

function cost(i, j, tasks, M) => int
  total_task_time = 0
  for task_num = i to j
    total_task_time += tasks[task_num]

  num_gaps = j - i
  req_time = total_task_time + num_gaps

  if req_time > M
    return infinity

  total_gap_time = M - total_task_time
  base_gap_time = floor(total_gap_time / num_gaps)

  for i = 1 to num_gaps
    gaps[i] = base_gap

  remainder_to_add = total_gap_time % num_gaps
  for i = 1 to remainder_to_add
    gaps[i] += 1

  cost = 0
  foreach k in gaps
    cost += (k - 1)²

  return cost
```

Question 3

a) Greedy Choice Definition

- At each step i, choose the stop with the minimal task time.
- This problem has the greedy choice property because travel times to and from a step are the same for any chosen stop at that step, and therefore has no effect on your choice of which stop.
- Since the travel times add the same constant amount for any chosen overall path, then we can ignore all travel times when looking for an optimal solution.
- This leaves us with the sum of task times across the entire race.
- If you make the greedy choice defined above at each step (which doesn't rely on any other steps), you will get the optimal solution.
- Choosing any other task instead can only add time to the overall race, making it not optimal anymore.

b) Represented as a Matroid

- S contains all stops across the entire race (i.e. s[i][j] for all i and j).
- A is in I iff A contains no more than 1 stop for a given step.
    - Examples:
        - A can contain any single stop
        - If |A| > 1, there cannot be more than 1 stop for a given step.
        - A may contain stops (that satisfy the above property) that are not connected, such a set containing only 1 stop from step 1 and 1 stop from step 3.
    - Hereditary:
        - With the above property, I will also contain all subsets of A.
    - Exchange property:
        - Also satisfied with above property
        - If a string of stops B = {A, B, C, D} is in I, then so is {A, B, C}, and if A = {A, B}, then with the exchange property, A ∪ {C} = {A, B, C}
    - Maximal Independent Subset:
        - A subset of size n containing exactly one stop from each step
- w(s) is the task time associated with that stop (from a[i][j])
    - $w'(s) = w_0 - w(s)$, where $w_0 > w(s)$ for every s
    - $w'(A) = n*w_0 - w(A)$ for any maximal independent subset A
        - n is total number of steps

The general greedy algorithm for matroids will produce an optimal answer for this problem. When S is sorted into non-increasing order using w'(s), they end up sorted in non-decreasing order of task time. Since I contains subset combinations where there is no more than 1 stop for a given step, then {x} won't be unioned with A if there is already a stop in A at the same step as {x}. Since we iterate through all stops, this proves that we will get a valid solution. To prove the solution is optimal, the ordering of S will ensure that the tasks with the smallest times are checked first, so the first task we come across for a given step will be the smallest time task for that step, and after that no more tasks from that step can be chosen (the subset won't be in I).