Assignment 06

Dynamic Programming

Ehu Shubham Shaw
Joe Johnson
April 12, 2025

Question 1: Maximum Weight Independent Set in Path Graphs

A graph G = (V, E) If the nodes of G=(V,E) may be labelled v1, v2,…,vn v1,v2,…,vn, and each node vi v i is connected by an edge solely to its immediate neighbors vi−1 v i−1 and vi + 1 v i+1 (if these neighbors exist), then G=(V,E) is a path. Weight w i w i is assigned to each node v i v i. Any subset of nodes in which no two have the same edge is an independent set of G G. Finding an independent set with the highest feasible total weight is our goal. The next sections (a), (b), and (c) deal with various facets of this issue. So according to the statement of the algorithm the heaviest-first greedy algorithm analyzes the graph as follows below there are still nodes in the graph choose the node with the highest weight add v to the independent set delete v, v and its neighbors from the graph return the chosen nodes as the claimed maximum-weight independent set.
So why it cant fail, as we provide a **simple path** and its node weights to show a case where the heaviest-first approach is **not** optimal Consider a path with 5 nodes identified as v1, v2, v3, v4, and v5.
Assume the weights are: w(v1) = 8, w(v2) = 9, w(v3) = 3, w(v4) = 1, and w(v5) = 6.
w(v 1) = 8, w(v 2) = 9, w(v 3) = 3, w(v 4) = 1, and w(v 5) = 6.
The edges in this path are (v 1 , v 2) , (v 2 , v 3) , (v 3 , v 4) , (v 4 , v 5) (v 1 ,v 2),(v 2 ,v 3),(v 3 ,v 4), (v 4 ,v 5).
What the Heaviest-First Algorithm Does actually that it is working so to start we choose v 2 v 2 (weight 9 is the largest).remove v 2 v 2 from the graph, along with its neighbors v 1 v 1 and v 3 v 3 the remaining nodes are: v 4 v 4 and v 5 v 5 the remaining graph has weights w(v4) = 1 and w(v5) = 6 select v 5 v 5 (weight 6), then remove v 4 v 4 and v 5 v 5 algorithm ends.
Result: The chosen set is { v 2 , v 5 } {v 2 ,v 5 } with total weight 9 + 6 = 15 9+6=15.

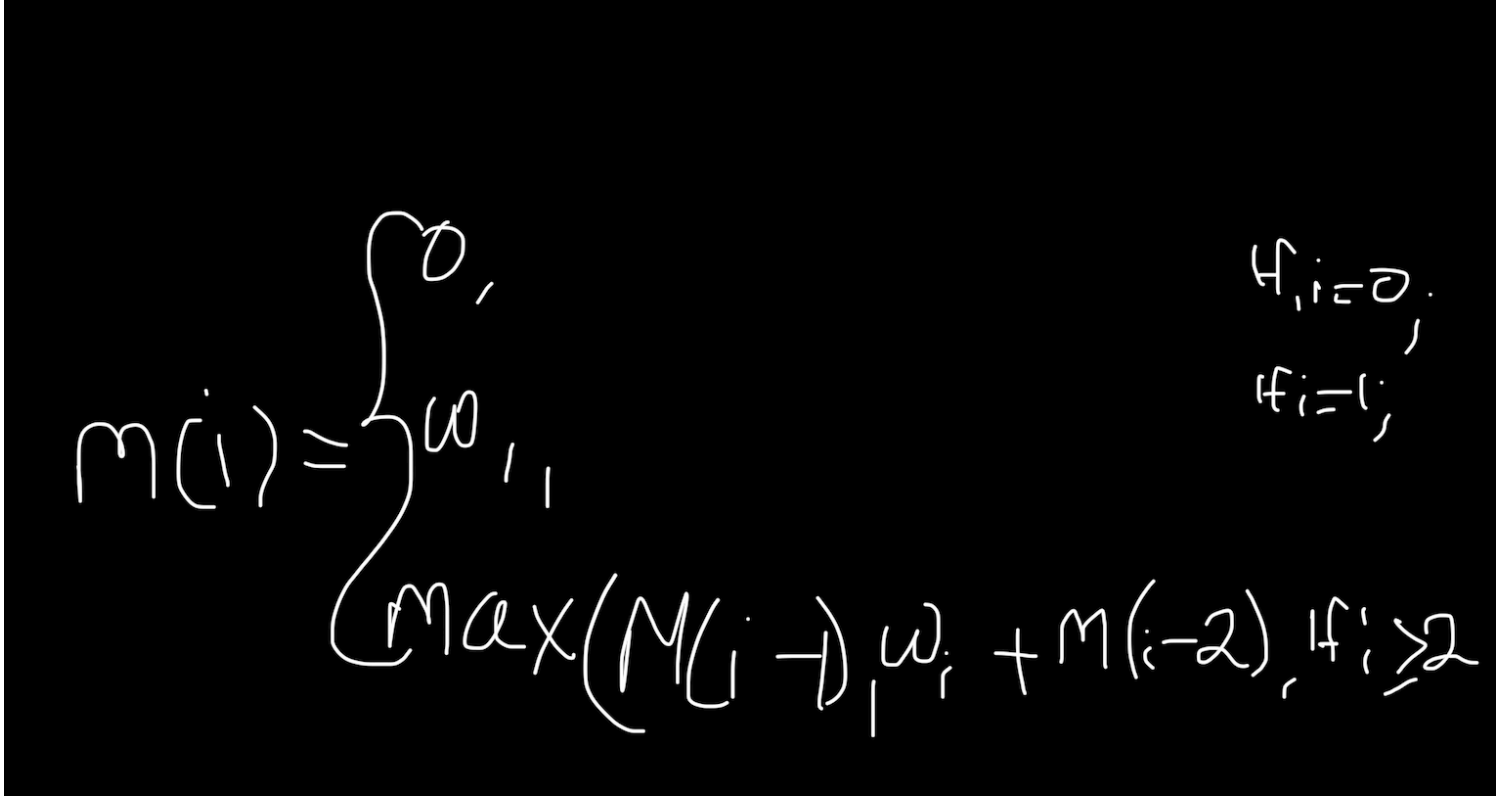
The optimal independent set take the independent set { v 1 , v 3 , v 5 } {v 1 ,v 3 ,v 5 }. None of these nodes are adjacent to one another in the path.
v 1 is only next to v 2 and is not part of the set.
v 3 v 3 is near to v 2 v 2 and v 4 v 4 but none are in the set.
v 5 v 5 is near to v 4 v 4 but not in the set.
The matrix { v 1 , v 3 , v 5 } {v 1 ,v 3 ,v 5 } is independent, and its total weight is:
W = w(v1) + w(v3) + w(v5) = 8 + 3 + 6 = 17.
This exceeds the 15 obtained by the heaviest-first technique, indicating that a greedy approach may not yield the best solution.

(b) A Dynamic-Programming Algorithm for a Path

Despite the failure of the heaviest-first strategy, the pathways remain "simple" enough for efficient optimization. We employ a classic dynamic programming (DP) approach to find the greatest weight independent set of a path.
Notation and Recurrence :
M(i) is the maximum weight of an independent set among the first i nodes, such as {v 1 ,…,v i }.
We need M(n), the maximum weight for the entire path.

We apply the following recurrence:

M (i) = { 0 , if i = 0 ; w 1 , if i = 1 ; max (M (i - 1), w i + M (i - 2)) , if i > 2.



Intuitively:

If we skip the i-th node, the most we can accomplish is M(i-1).
To avoid adjacency, the i-th node must be skipped, therefore we add w i w i and M (i – 2) M(i–2).
Computing M (i) M(i) We can fill in a table M [0.. n] iteratively: M (0) = 0. M(0)=0. M (1) = w 1 . M(1)=w 1 . For i = 2 i=2 to n n: M (i) = max (M (i – 1) , w i + M (i – 2)) . M(i)=max(M(i−1),w i +M(i−2)).
After computing M (n) M(n), we have the maximum total weight of an independent set in the entire path

To reconstruct which nodes form this maximum-weight set, we can trace back through the table: Start from i = n i=n. If M (i) = M (i – 1) M(i)=M(i−1), then we did not include node i i. Move i ← i – 1 i←i−1. If M (i) = w i + M (i – 2) M(i)=w i +M(i−2), then we did include node i i. Record node i i in the solution and move i ← i – 2 i←i−2. This process continues until i <= 0 I <= 0.

(C) Optimal Algorithm for Maximum Weight Independent Set.
We can solve this problem using dynamic programming. Let OPT(i) be the maximum weight of an independent set in the path consisting of nodes v1 through vi.
Algorithm MaxWeightIndependentSet(G):

```
  OPT[0] = 0
  OPT[1] = wi
   for i = 2 to n:
      OPT[i] = max(OPT[i-1], OPT[i-2] + w)
  S = empty set
   i = n
   while i > 0:
      if i == 1 or OPT[i-2] + wi > OPT[i-1]:
         Add vi to S
         i = i - 2
      else:
         i = i - 1

   return S
```

This algorithm uses the following recurrence relation:
OPT(0) = 0 (empty set)
OPT(1) = w1 (just include the first node)
For i >= 2: OPT(i) = max(OPT(i-1), OPT(i-2) + wi)
The interpretation of this recurrence is:
Either we don't include node vi, in which case the maximum weight is OPT(i-1)
Or we include node v_i, in which case we can't include v_{i-1} (since they're adjacent), so the maximum weight is OPT(i-2) + wi
The algorithm runs in O(n) time and uses O(n) space, which is polynomial in the size of the input and independent of the values of the weights.

Question 2: Consulting Team Job Selection

We need to schedule jobs for a consulting team over n weeks. Each week, we can choose either a low-stress job "earning li dollars', a high-stress job 'earning hi dollars', or no job. The constraint is that a high-stress job in week i requires no job in week i-1. So for a part a

a - Counterexample for the Given Algorithm

The algorithm evaluates each week i independently, and if hi+1 > li + li+1, it chooses no job in week i and a high-stress job in week i+1.

Counter example :- consider n = 3 with the following values:

- Week 1: L1 = 5, h1 = 10
- Week 2: L2 = 6, h2 = 15
- Week 3: L3 = 7, h3 = 8
- Week 1: L1 = 5, h1 = 10
- Week 2: L2 = 20, h2 = 15
- Week 3: L3 = 7, h3 = 30

The algorithm would make decisions:

- For i = 1: Check if h2 > l1 + L2. Since 15 < 5 + 20 = 25, choose low-stress in week 1.
- For i = 2: Check if h3 > L2 + L3. Since 30 > 20 + 7 = 27, choose no job in week 2 and hi-stress in week 3.

So the algorithm's plan is: low-stress, none, high-stress with value 5 + 0 + 30 = 35.

The optimal plan would be high-stress, low-stress, low-stress with value 10 + 20 + 7 = 37.

Therefore, the given algorithm does not always find the optimal solution.

Part (b): Efficient Algorithm for Finding an Optimal Plan

Solution: We can use dynamic programming to solve this problem. Let OPT(i) represent the maximum value obtainable from weeks 1 through i.

```
Algorithm OptimalPlan(l1...ln, h1...hn):
    OPT[0] = 0
    OPT[1] = max(l1, h1)
    for i = 2 to n:
        option1 = OPT[i-1] + li
        option2 = OPT[i-2] + hi
        option3 = OPT[i-1]
        OPT[i] = max(option1, option2, option3)

    return OPT[n]
```

To reconstruct the actual plan, we can keep track of the choices made at each step.
The algorithm runs in O(n) time and uses O(n) space, making it an efficient solution.

Question 3: Longest Path in Ordered Graph

So we see a given an ordered directed graph G where edges only go from lower-indexed nodes to higher-indexed nodes, we need to find the length of the longest path from V1 to Vn.

A :- Counterexample for the Given Algorithm

The given algorithm starts at v1 and repeatedly chooses the edge to the node with the smallest possible index:

```
Set w = v1
Set L = 0
While there is an edge out of the node w
    Choose the edge (w, vⱼ) for which j is as small as possible
    Set w = vj
    Increase L by 1
end while
Return L as the length of the longest path
```

Looking at Figure 6.29, the correct answer is 3, using the path with edges (v1, v2), (v2, v4), and (v4, v5).
However, the algorithm would make these choices:

1. Start with w = v1
2. Choose the edge (v1, v2) since v2 has the smallest index among v1's neighbors
3. Update w = v2, L = 1
4. Choose the edge (v2, v4) since v4 has the smallest index among v2's neighbors
5. Update w = v4, L = 2
6. Choose the edge (v4, v5) since v5 is the only neighbor of v4
7. Update w = v5, L = 3
8. No more edges from v5, so return L = 3

In this case, the algorithm actually returns the correct answer of 3. Let me provide a different example where the algorithm fails:

Consider this ordered graph with 5 nodes:

- Edges: (v1, v2), (v1, v3), (v2, v5), (v3, v4), (v4, v5)

The longest path from v1 to v5 has length 3: (v1, v3), (v3, v4), (v4, v5).
The algorithm would make these choices:

1. Start with w = v1
2. Choose edge (v1, v2) since v2 has smaller index than v3
3. Update w = v2, L = 1
4. Choose edge (v2, v5) as it's the only one from v2
5. Update w = v5, L = 2
6. No more edges from v5, so return L = 2

The algorithm returns 2, but the correct answer is 3.

b :- Efficient Algorithm for Finding the Longest Path

We can use dynamic programming to solve this problem. Let LongestPath[i] represent the length of the longest path from v1 to vi.

```
Algorithm LongestPathInOrderedGraph(G):
   LongestPath = array of size n
   LongestPath[1] = 0

   for j = 2 to n:
        LongestPath[j] = 0
        for i = 1 to j-1:
            if (v_i, v_j) is an edge in G:
                LongestPath[j] = max(LongestPath[j], LongestPath[i] + 1)

    return LongestPath[n]
```

This algorithm runs in O(|V|^2 + |E|) time, which is polynomial in the size of the graph. Since we're working with an ordered graph where edges only go from lower to higher indices, this is an efficient solution.

Question 4: Minimizing Cost with Relocations

A consulting business operates either from New York (NY) or San Francisco (SF) each month. The operating costs are Ni for NY and Si for SF in month i. Additionally, switching locations incurs a fixed moving cost M.

a :- Counterexample for the Given Algorithm

The given algorithm simply chooses the city with the lower operating cost for each month independently:

```
For i = 1 to n
    If Ni < Si then
        Output "NY in Month i"
    Else
        Output "SF in Month i"
    End
```

Using the example given in the problem:

- Month 1: NY = 1, SF = 50
- Month 2: NY = 3, SF = 20
- Month 3: NY = 20, SF = 2
- Month 4: NY = 30, SF = 4
- Moving cost M = 10

The algorithm would choose:

- Month 1: NY (cost 1)
- Month 2: NY (cost 3)
- Month 3: SF (cost 2)
- Month 4: SF (cost 4)

Total cost: 1 + 3 + 2 + 4 + 10 = 20 (with one move from NY to SF after month 2)

Different example:

- Month 1: NY = 15, SF = 10
- Month 2: NY = 5, SF = 10
- Month 3: NY = 15, SF = 10
- Moving cost M = 12

The algorithm would choose:

- Month 1: SF (cost 10)

- Month 2: NY (cost 5)
- Month 3: SF (cost 10)

Total cost: 10 + 5 + 10 + 2*12 = 49

However, staying in SF the entire time would cost 10 + 10 + 10 = 30, which is better than 49.

Therefore, the greedy algorithm does not always find the optimal solution.

b :- Example Requiring At Least Three Moves

An instance where every optimal plan must change locations at least three times:

Month	1	2	3	4	5	6	7
NY	1	50	1	50	1	50	1
SF	50	1	50	1	50	1	50

In this tabular example shown up, the costs alternate dramatically between NY and SF. The optimal plan would be to switch locations every month: [NY, SF, NY, SF, NY, SF, NY].

With this plan:

- Operating costs: 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7
- Moving costs: 6 * 5 = 30
- Total cost: 37

If we try to reduce the number of moves by staying in one location for two consecutive months:

- For instance, [NY, NY, NY, SF, NY, SF, NY]
- Operating costs: 1 + 50 + 1 + 1 + 1 + 1 + 1 = 56
- Moving costs: 4 * 5 = 20
- Total cost: 76

The difference in operating costs (56 - 7 = 49) is much greater than the savings in moving costs (30 - 20 = 10), so any plan with fewer than 6 moves will be suboptimal.

Part (c): Efficient Algorithm for Finding Minimum Cost Plan

We can use dynamic programming to solve this problem. Let MinCost[i][loc] be the minimum cost to operate for months 1 through i, ending in location loc (where loc is either NY or SF).

```
Algorithm MinimumCostPlan(n, M, N1...Nn, S1...Sn):
    MinCostNY = array of size n+1
    MinCostSF = array of size n+1
    MinCostNY[0] = 0
    MinCostSF[0] = 0
    for i = 1 to n:
        costStayNY = MinCostNY[i-1]
        costMoveToNY = MinCostSF[i-1] + M
        MinCostNY[i] = min(costStayNY, costMoveToNY) + N[i]
        costStaySF = MinCostSF[i-1]
        costMoveToSF = MinCostNY[i-1] + M
        MinCostSF[i] = min(costStaySF, costMoveToSF) + S[i]
    return min(MinCostNY[n], MinCostSF[n])
```

To reconstruct the actual plan, we would need to keep track of the decisions made at each step.

This algorithm runs in O(n) time and uses O(n) space, making it an efficient solution.

Question 17: Longest Rising Trend

Given a sequence of stock prices P[1], P[2], ..., P[n], a rising trend is a subsequence starting with the first day (i1 = 1) where each price is strictly greater than the previous one in the subsequence.

a :- example for the given algorithm

The given algorithm tracks only continuous increasing sequences

```
Define i = 1
      L = 1
For j = 2 to n
    If P[j] > P[i] then
        Set i = j
        Add 1 to L
    Endif
Endfor
```

Using the example from the problem: [10, 1, 2, 11, 3, 4, 12]

- The longest rising trend is [1, 4, 7] with values [10, 11, 12] and length 3.

Let's trace through the algorithm:

1. Initialize i = 1, L = 1 (tracking P[1] = 10)
2. For j = 2: P[2] = 1 is not > P[1] = 10, so no change
3. For j = 3: P[3] = 2 is not > P[1] = 10, so no change
4. For j = 4: P[4] = 11 > P[1] = 10, so i = 4, L = 2
5. For j = 5: P[5] = 3 is not > P[4] = 11, so no change
6. For j = 6: P[6] = 4 is not > P[4] = 11, so no change
7. For j = 7: P[7] = 12 > P[4] = 11, so i = 7, L = 3

The algorithm returns L = 3, which is the correct length of the longest rising trend.

So here consider this example = [10, 5, 20, 15, 30]:

- The longest rising trend would be [1, 3, 5] with values [10, 20, 30] and length 3.

Tracing the algorithm:

1. Initialize i = 1, L = 1 (tracking P[1] = 10)
2. For j = 2: P[2] = 5 is not > P[i=1] = 10, so no change
3. For j = 3: P[3] = 20 > P[i=1] = 10, so i = 3, L = 2
4. For j = 4: P[4] = 15 is not > P[i=3] = 20, so no change
5. For j = 5: P[5] = 30 > P[i=3] = 20, so i = 5, L = 3

The algorithm returns L = 3, which is correct as I believe the algorithm actually does correctly solve the problem of finding the length of the longest rising trend as defined in the problem statement.

b :- Efficient Algorithm for Finding the Longest Rising Trend

Since the algorithm in part (a) appears to be correct, we can formalize it:

```
Algorithm LongestRisingTrend(P[1...n]):
    L = 1
    i = 1
    for j = 2 to n:
        if P[j] > P[i]:
            i = j
            L = L + 1
    return L
```

This algorithm runs in O(n) time, which is optimal since we need to examine each price at least once. However, if we want to find the actual days in the longest rising trend (not just the length), we would need to store the indices:

```
Algorithm LongestRisingTrendWithIndices(P[1...n]):
    trend = [1] #Day 1
    lastPrice = P[1]
    for j = 2 to n:
        if P[j] > lastPrice:
            trend.append(j)
            lastPrice = P[j]
    return trend
```

Both algorithms are efficient with O(n) time complexity.

Question 28: Maximum Schedulable Subset

Given n jobs with deadlines d_i and processing times t_i, we need to find a schedulable subset of maximum size, where a subset is schedulable if all jobs can be completed before their deadlines.

a :- Proof of Optimal Solution with Deadline Ordering

Proof: We need to prove that there exists an optimal schedulable subset J where jobs are scheduled in increasing order of their deadlines so lets assume we have an optimal schedulable subset J' where jobs are not scheduled in deadline order. This means there exist consecutive jobs i and j in the schedule such that di > dj. Let's consider the effect of swapping these jobs in the schedule so:

- Before swap: job i starts at time s_i and finishes at time fi = si + ti
- Job j starts at time sj = fi and finishes at time fj = sj + tj = fi + tj

After swapping:

- Job j would start at time s_i and finish at time fj = si + tj
- Job i would start at time si = fj and finish at time f'i = s'i + ti = f'j + ti = si + tj + ti = si + ti + tj = fj

Job j had an earlier deadline (dj < di) and now finishes earlier (f'j < fi), thus it will fulfill its deadline. Job i now complete the same time as job j (f'i = fj). Since job j completed its deadline in the original timetable (fj ≤ dj) and dj < di, we obtain f'i = fj ≤ dj < di, implying that job i will also meet it.

Thus, changing the jobs retains the subset's schedulability. By continuously performing this swapping procedure, we can convert any optimal schedule into one in which jobs are ordered by deadlines while not diminishing the size of the subset. Thus, there exists an optimal solution where jobs are scheduled in increasing order of their deadlines.

Part (b): Algorithm to Find Maximum Schedulable Subset

We will now discuss a greedy method (also known as Moore's method) that generates an optimal subset of jobs that can be scheduled to satisfy all deadlines. The fundamental idea is to handle jobs in ascending order of deadlines while maintaining a candidate set. If, after adding a new job, the total processing time exceeds the job's deadline, we remove the job with the longest processing time from our applicant pool. This guarantees that we "free up" the greatest processing time, allowing for more jobs.

So we sort in depending d1 <= d2 <= ⋯ <= d

Initialize L ← FI L←FI (an empty set for selected jobs).

To track the overall processing time of jobs in L L, use sumT ← 0.

For i = 1 to n:

Add Job (I):

Set: L ← L U { I } L ← LU{I}.

Update: sumT → sumT + t.

I sumT ← sumT + t I .

Check feasibility:

If the total processing time exceeds the current job's deadline (sumT > dI), then:

Remove the job with the highest processing time:

Let L = arg, max, j E L.

tjL= argmax jEL t j . Remove job L from L and update: sumT ← sumT – t L . sumT←sumT–t L . Now lets Schedule the accepted jobs at the end of the iteration, the set L L holds the jobs that were not eliminated. Schedule jobs in L L based on their deadlines (which corresponds to the order in which we sorted them initially). Construction ensures timely completion of all jobs in L.

```
def schedule_jobs(jobs):
  jobs.sort(key=lambda job: job.d)
   L = []
   sumT = 0
   maxHeap = []
   for job in jobs:
      L.append(job)
      sumT += job.t
      heapq.heappush(maxHeap, -job.t)
      if sumT > job.d:
        largest_t = -heapq.heappop(maxHeap)
         sumT -= largest_t
        remove_job_with_processing_time(L, largest_t)
  return L
```

Run Time for the sorting jobs based on deadlines takes O(nlogn). Iteration: Inserting each job into the set L and the max-heap takes O(logn) time. In the worst-case situation, each job requires one insertion and maybe one deletion from the max-heap, resulting in O(nlogn) time. The whole running time is O(nlogn).