Foundations of Algorithms by Arunabha Sen

Final (Fall 2024)

Instructions

Final Exam - Requires Respondus LockDown Browser + Webcam

Due Dec 3 at 2:45pm **Points** 75 **Questions** 6
Available Dec 3 at 1:30pm - Dec 3 at 2:45pm 1 hour and 15 minutes
Time Limit 75 Minutes Requires Respondus LockDown Browser

Instructions

The duration of the exam is 75 minutes. The exam contains 6 questions, consisting of 2 multiple choice/multiple answer type questions, 1 fill in the blanks (from dropdown), and 3 Long Answer Type Questions.

This is a closed book, closed internet, and strictly an individual effort (No cheatsheets). You will be provided two blank sheets of 8.5x11 inch or A4 paper to do some rough work. You'll have to write your name and student ID on these sheets as these will be collected at the end of the exam.

The marking scheme for each multiple-answer type question is as follows:

Each correct option marked will award points equal to the total points for that question divided by the total number of correct options.
Each incorrect option marked will deduct points equal to 0.5 times the points for each correct option marked.

For example, if a question is of 10 points and the correct options are A and C.
If you mark AC: 10 points.
If you mark A: 5 points.
If you mark AD: 2.5 points.
If you mark ABD: 0 points.
If you mark BD: 0 points.

The fill-in-the-blank and long answer type questions don't have negative markings.

This quiz was locked Dec 3 at 2:45pm.

Q-1. Multiple Correct

Select all the correct statements regarding the closest pair of points Divide and Conquer algorithm taught in the course.

- [x] Suppose there is a set of n points each with the same x-coordinate. The smallest possible asymptotic time complexity required to find the closest pair of points in this instance is $O(nlogn)$.
- [ ] Suppose there is a set of n points each with the same x-coordinate. The smallest possible asymptotic time complexity required to find the closest pair of points in this instance is $O(n)$.
- [x] Suppose we modify the combine step of the closest pair of points algorithm such that distance δ from dividing line *L* is updated immediately to δ' whenever the distance between two points on either side of *L* is discovered to be less than δ. In this case, the time complexity of the closest pair of points algorithm may not improve.
- [ ] The reason why the combine step of the closest pair of points algorithm can be solved in linear time is because comparing every two points within distance δ of dividing line L will always require linear time.
- [ ] Suppose we modify the combine step of the closest pair of points algorithm such that distance δ from dividing line *L* is updated immediately to δ' whenever the distance between two points on either side of *L* is discovered to be less than δ. In this case, the time complexity of the closest pair of points algorithm is guaranteed to be improved by a constant factor.
- [x] The reason why the combine step of the closest pair of points algorithm can be solved in linear time is because the number of comparisons made between points within distance δ of dividing line L can be shown to be a constant.

Suppose we modify the combine step of the closest pair of points algorithm such that distance δ from dividing line *L* is updated immediately to δ' whenever the distance between two points on either side of *L* is discovered to be less than δ. In this case, the time complexity of the closest pair of points algorithm is guaranteed to be improved by a constant factor.

The reason why the combine step of the closest pair of points algorithm can be solved in linear time is because the number of comparisons made between points within distance δ of dividing line L can be shown to be a constant.

Q-2. The question asked in final, asks for justification.

My friend owns a factory equipped with two identical machines, each capable of processing jobs. This setup allows my friend to schedule up to two jobs during any given time interval. Additionally, there is always a set of jobs available to choose from, but each job is constrained by a specific time interval during which it can be executed, defined by a fixed start time and a fixed finish time. My friend mentioned that he knows a greedy algorithm capable of producing an optimal schedule for this variant of interval scheduling, where at most two jobs can be scheduled simultaneously. If J represents the set of all available fixed job intervals, the algorithm proceeds as follows:

Algorithm 1 DOUBLE GREEDY INTERVAL SCHEDULING

```
Input: Set J
Sort the elements of J in non-decreasing order of start time
A_1 ← ∅
for j = 1, 2, ..., |J| do
    if job j does not overlap at any point with any job in A_1 then
        A_1 ← A_1 ∪ {j}
Set J ← J \ A_1 (i.e. remove the contents of A_1 from J)
A_2 ← ∅
for j = 1, 2, ..., |J| do
    if job j does not overlap at any point with any job in A_2 then
        A_2 ← A_2 ∪ {j}
Return A_1 ∪ A_2
```

Does this algorithm find the optimal solution for the problem? If your answer is "Yes", then provide arguments that justify your claim. If your answer is "No", provide an example where the algorithm fails to find the optimal solution.

No. Counterexample: Job 1: [1, 3] - Job 2: [2, 4] - Job 3: [3, 5] - Job 4: [4, 6]

Explanation: Using the greedy algorithm, it would select Job 1 and Job 3 first, leaving no room for Job 2 and Job 4. However, the optimal schedule would be to select Job 2 and Job 4, as they do not overlap.

Same question on Chegg, but its an MCQ

Answer to the Chegg Question:

Option 1: - Job 1: [1, 3] - Job 2: [2, 4] - Job 3: [3, 5]

Explanation: If we use the greedy algorithm, it would select Job 1 and Job 3 first, leaving no room for Job 2. However, the optimal schedule would be to select Job 2 and Job 3, as they do not overlap.

Option 2: - Job 1: [1, 3] - Job 2: [2, 4] - Job 3: [3, 5] - Job 4: [4, 6]

Explanation: Using the greedy algorithm, it would select Job 1 and Job 3 first, leaving no room for Job 2 and Job 4. However, the optimal schedule would be to select Job 2 and Job 4, as they do not overlap.

Option 3: - Job 1: [1, 3] - Job 2: [2, 4] - Job 3: [3, 5] - Job 4: [4, 6] - Job 5: [5, 7]

Explanation: Using the greedy algorithm, it would select Job 1 and Job 3 first, leaving no room for Job 2, Job 4, and Job 5. However, the optimal schedule would be to select Job 2 and Job 5, as they do not overlap.

Correct Answer: Option 2

Q-3.

The following is a list of statements about flow networks. Let $G(V, E, c, f)$ be a flow network with source node $s \in V$ and sink node $t \in V$, and let e be a function that gives the capacities $c(e)$ and let f be a function that gives the flow size $f(e)$ for all edges in E.

Select all statements that are true.

- [ ] The net flow across any s,t-cut (A, B) in G is the sum of the flows on all the edges exiting A.

- [ ] For any node (except source & sink), the sum of incoming link capacities equals to sum of outgoing link capacities.

- [ ] Suppose $G(V, E, c, f)$ has exactly one minimum s, t-cut (A, B) and its max flow value is F. If we increase the capacity of all the edges by an amount $b > 0$, i.e., $c'(e) = c(e) + b, \forall e \in E$ to form a network $G'(V, E, c', f')$, then we can be sure the max flow value in G' will be $b + F$.

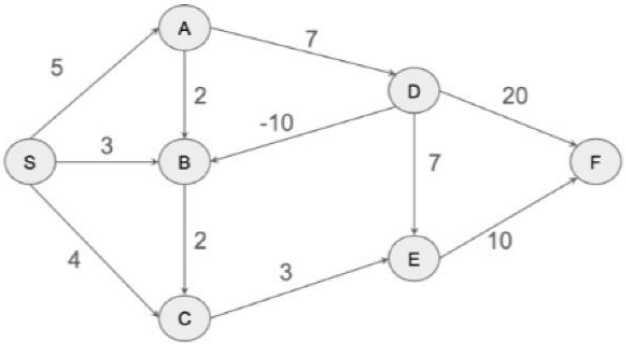
- [ ] For any node (except source & sink), the sum of the incoming flows equals to sum of the outgoing flows.

- [ ] The capacity of any s,t-cut (A, B) in G is the sum of the capacities of all edges exiting A.

- [ ] During the execution of Ford-Fulkerson algorithm, we can increase or decrease the flow on an edge.

Q-4.

We have run the first three iterations of the outer for-loop in the Bellman-Ford shortest-path dynamic programming algorithm for i = 1, 2, 3 on graph G below, obtaining the configuration of the Bellman-Ford dynamic programming matrix shown below. Run two more iterations of the outer for-loop in the Bellman-Ford algorithm for i = 4, 5, and fill out the empty cells in the following table:



	S	A	B	C	D	E	F
0	∞	∞	∞	∞	∞	∞	0
1	∞	∞	∞	∞	20	10	0
2	∞	27	∞	13	17	10	0
3	17	24	15	13	17	10	0
4	i.	ii.	iii.	iv.	v.	vi.	0
5	vii.	viii.	ix.	x.	xi.	xii.	0

Answer 1:

17

Answer 2:

17

Answer 3:

15

Answer 4:

13

Answer 5:

5

Answer 6:

10

Answer 7:

17

Answer 8:

12

Answer 9:

15

Answer 10:

13

Answer 11:

5

Answer 12:

10

Q-5.

Define the following classes of problems:

(i) Class P (1 point)
(ii) Class NP (1 point)
(iii) Class NP-Complete (1 point)

Problem 2B(i): Consider two Decision problems P_1 and P_2 one of which, say P_1, is generally *accepted* to be a "hard" problem (in the sense that no polynomial time algorithm is known for P_1) and the other one P_2 is *suspected* to be a hard problem. Suppose that one came up with a transformation (i.e., a set of rules) that in polynomial time can construct from any instance of the problem P_1 an instance of the problem P_2, such that the answer to the decision question for the problem P_1 is "Yes" if and only if the answer to the decision question for the problem P_2 is also "Yes". If such a construction is possible, can it be claimed that if P_1 is hard, P_2 must also be hard? If your answer is "Yes", then provide arguments in support of your claim. If your answer is "No", then provide arguments in support of your claim. (5 points)

Problem 2B(ii): Consider two Decision problems P_1 and P_2 one of which, say P_1, is generally *accepted* to be a "hard" problem (in the sense that no polynomial time algorithm is known for P_1) and the other one P_2 is *suspected* to be a hard problem. Suppose that one came up with a transformation (i.e., a set of rules) that in polynomial time can construct from any instance of the problem P_2 an instance of the problem P_1, such that the answer to the decision question for the problem P_2 is "Yes" if and only if the answer to the decision question for the problem P_1 is also "Yes". If such a construction is possible, can it be claimed that if P_1 is hard, P_2 must also be hard? If your answer is "Yes", then provide arguments in support of your claim. If your answer is "No", then provide arguments in support of your claim. (5 points)

Just in case you want to ask AI for an answer, here is the transcribed text of the questions:

Q 2B(i) Consider two Decision problems P1 and P2 one of which, say P1, is generally accepted to be a "hard" problem (in the sense that no polynomial time algorithm is known for P1) and the other one P2 is suspected to be a hard problem. Suppose that one came up with a transformation (i.e., a set of rules) that in polynomial time can construct from any instance of the problem P1 an instance of the problem P2, such that the answer to the decision question for the problem P1 is "Yes" if and only if the answer to the decision question for the problem P2 is also "Yes". If such a construction is possible, can it be claimed that if P1 is hard, P2 must also be hard? If your answer is "Yes", then provide arguments in support of your claim. If your answer is "No", then provide arguments in support of your claim.

Q 2B(ii) Consider two Decision problems P1 and P2 one of which, say P1, is generally accepted to be a "hard" problem (in the sense that no polynomial time algorithm is known for P1) and the other one P2 is suspected to be a hard problem. Suppose that one came up with a transformation (i.e., a set of rules) that in polynomial time can construct from any instance of the problem P2 an instance of the problem P1, such that the answer to the decision question for the problem P2 is "Yes" if and only if the answer to the decision question for the problem P1 is also "Yes". If such a construction is possible, can it be claimed that if P1 is hard, P2 must also be hard? If your answer is "Yes", then provide arguments in support of your claim. If your answer is "No", then provide arguments in support of your claim.

Answers

(i) P: Problems solvable in polynomial time by a deterministic Turing machine.
(ii) NP: The problems that, if given a solution, can be verified in polynomial time fall in NP.
(iii) NP-Complete: An NP problem to which all other NP problems can be reduced in polynomial time, making it as hard as any problem in NP.

2B(i). Yes, this is the basis for relating the hardness of decision problems, particularly in the context of NP problems. If an instance I2 of decision problem P2 is constructed in polynomial time from an instance I1 of P1, such that I2 is "Yes" if and only if I1 is "Yes" (and similarly for "No"), then P2 is at least as hard as P1. This means that if P2 can be solved in polynomial time, then P1 can also be solved in polynomial time, preserving their relative complexity.

2B(ii). No, it cannot be claimed that P2 is hard. The reduction provided transforms instances of P2 into instances of P1, ensuring that if the answer for P2's instance I2 is "Yes," then the answer for P1's instance I1 is also "Yes." This reduction shows that solving P1 helps solve P2, meaning P2 is **not harder** than P1. However, this **does not** establish P2's hardness because the direction of reduction matters. To prove that P2 is as hard as P1, we would need to reduce P1 to P2, showing that solving P2 would also solve P1 efficiently.

Q-6.

Consider the following *job-to-processor* mapping (assignment) problem:

Given a set of n jobs $J_i, 1 \leq i \leq n$, an execution time $t_i, 1 \leq i \leq n$ is associated with each job and a set of m processors $P_1, \ldots P_m$. The n jobs have to be mapped to m processors. The total execution time used by processor $P_j, 1 \leq j \leq m$ is the sum of the execution times of the jobs mapped to this processor. The *Maximum-Completion-Time* of a mapping is the maximum of the Completion-Times among the m processors. The Completion-Time in a processor is the sum of the execution times of all the jobs mapped into that processor. In this problem, one needs to find the job-to-processor mapping that minimizes the *Maximum-Completion-Time*.

The following algorithm is used to solve this problem:

Algorithm 1

```
begin
    for j = 1 to m do
    F_j = 0;
    for i = 1 to n do
        begin
            Find k, where k is the smallest integer j for which
            F_j + t_i is minimum (1 ≤ j ≤ m);
            Assign job J_i to processor k;
            Update F_k ← F_k + t_i;
        end
end
```

Question (i): Does this algorithm find the optimal solution for the problem? If your answer is "Yes", then provide arguments that justify your claim. If your answer is "No", provide an example where the algorithm fails to find the optimal solution. (5 points)

Question (ii): If the solution produced by this algorithm does not always find the optimal solution, can you establish how far from the optimal it can get? (In other words, how large the error can be). If you claim that error introduced by this algorithm cannot exceed $X\%$ of the Optimal result, then state the value of X and provide arguments to justify your claim that the error introduced by this algorithm cannot exceed $X\%$ of the Optimal result. Clearly, X is an upper bound on the error. In your answer, you should try to provide the least (or the smallest) upper bound. (5 points)

Question (iii): Consider a special case of the problem in where $m = 2$. If you think that this restricted problem is NP-complete, then provide arguments to justify your claim. Otherwise, give a polynomial time algorithm to solve the problem. (2 points)

Question (iv): Consider a special case of the problem where $m = 1$. If you think that this restricted problem is NP-complete, then provide arguments to justify your claim. Otherwise, give a polynomial time algorithm to solve the problem. (2 points)

Answers

(i) The given Greedy Algorithm does not always find the optimal solution. In the example, balancing the load more strategically achieves a better result.
Example:
Consider two processors (P1,P2) and three jobs with times (10, 20, 30):

- **Step 1**: Assign job 1 (time 10) to P1 (F1 = 10, F2 = 0)
- **Step 2**: Assign job 2 (time 20) to P2 (F1 = 10, F2 = 20)
- **Step 3**: Assign job 3 (time 30) to P1 (F1 = 40, F2 = 20)

Maximum-Completion-Time = max(F1, F2) = 40.
Optimal Solution:

- Assign job 1 (time 10) to P1.
- Assign job 2 (time 20) to P1.
- Assign job 3 (time 30) to P2.

This results in (F1 = 30, F2 = 30), which minimizes the Maximum-Completion-Time = 30.

(ii) The greedy algorithm mentioned can have a performance ratio of (2−1/m), meaning the error introduced by this algorithm cannot exceed (2−1/m) times the optimal solution. For ***m=2***, this results in an error bound of ***1.5 times*** the optimal solution

(iii) For ***m=2***, processors, the problem is still NP-complete. However, it can be solved using dynamic programming in polynomial time for this specific case. By defining states based on the subset of jobs and balancing the loads among two processors, a solution can be achieved in O(n·2^n) time complexity.

(iv) For ***m=1***, processor, the problem simplifies to sorting the jobs based on their execution times in descending order and assigning them to the processor. This can be done in O(nlogn) time complexity. Thus, the problem is not NP-complete and can be solved in polynomial time.