Quiz 2 Solutions

- Do not open this quiz booklet until you are directed to do so.
- This quiz ends at 3:55 p.m.
- When the quiz begins, write your name on the top of *EVERY* page in this quiz booklet, because the pages will be separated for grading.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Do not put part of the answer to one problem on the back of the sheet for another problem.
- Plan your time wisely. Do not spend too much time on any one problem. Read through all of them first and attack them in the order that allows you to make the most progress.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- When describing an algorithm, describe the main idea in English. Use pseudocode only to the extent that it helps clarify the main ideas.
- Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

Name: ________________________________________________________________

Please circle your TA’s name and recitation:

Nitin  Brian  Mihai  Yoav

10am  11am  12pm  1pm  2pm
Problem 1. True or False, and Justify [30 points] (6 parts)
Circle T or F for each of the following statements to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is incorrect, explain why. Your justification is worth more points than your true-or-false designation.

T   F Assume you are given a magical priority-queue data structure, which performs extract-min, insert and decrease-key in constant time. Then you can implement Dijkstra’s algorithm to run in $O(E + V)$ time.

Solution: TRUE. Dijkstra’s algorithm can be analyzed in terms of three operations. Each vertex is initialized and added to the queue once - $\Theta(V)$ operations. Each edge is relaxed at most once - $O(E)$ operations. Each vertex is extracted from queue at most once - $O(V)$ operations. Thus, the running time is

$$\Theta(V) \cdot T_{insert} + O(V) \cdot T_{extract-min} + O(E) \cdot T_{decrease-key}$$

If all times are constant, the running time is $O(E + V)$.

T   F Yoav claims the following modification to Dijkstra’s shortest paths algorithm will allow it to work for negative edge weights as well. The idea is to find the smallest edge weight, say $w < 0$, and add $-w$ to all the edges, thus making all of them non-negative. Now run Dijkstra’s algorithm on these new edge weights. To argue correctness, Yoav claims that the addition doesn’t change the relative order between edges, i.e., if $a < b$ then $a + w < b + w$.

Yoav’s algorithm for the shortest paths problem is correct.

Solution: FALSE. A simple counterexample demonstrates that this is false. Imagine a graph with edge weights as in the table:

<table>
<thead>
<tr>
<th>weight</th>
<th>orig.</th>
<th>after mod.</th>
</tr>
</thead>
<tbody>
<tr>
<td>w(a,c)</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>w(a,b)</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>w(b,c)</td>
<td>-2</td>
<td>0</td>
</tr>
</tbody>
</table>

The original graph has shortest path $a \rightarrow c$ through vertex $b$, whereas in the modified graph it is direct. This change in shortest path invalidates the modification procedure. Intuitively this is so because adding a constant changes the relative weight of paths consisting of different numbers of edges.

Another way to answer this question is to note that any negative weight cycles in the original graph will be removed by the modification, creating a well defined shortest path where none existed originally; this thus changes the problem to one with a different solution rather than solving the original problem.
T F Yoav claims that this approach works for MST as well. I.e., after making the above changes to the weights and running any MST algorithm, the algorithm outputs correct MST for the original graph.
Yoav’s algorithm for finding MST is correct.

Solution: TRUE. Both Kruskal’s and Prim’s algorithm depend only on the relative order of edge weights and so will be unaltered by adding a constant to each edge. We were asked to show that the modification will work for any MST algorithm, however. Since each MST has the same number of edges, the MST is invariant to adding a constant to each edge weight. Thus, any correct MST algorithm will produce a correct MST when all edge weights are augmented by a constant.

T F Consider the family \( \mathcal{H} \) of hash functions that contains all functions \( h_{a,b} : \{0 \ldots 10q\} \rightarrow \{0 \ldots q - 1\} \) of the form \( h_{a,b}(x) = x + a + b^2 \mod q \), where \( a, b \in \{0 \ldots q - 1\} \), \( q \) prime.
The family \( \mathcal{H} \) is universal.

Solution: FALSE. Consider collisions from \( u \) and \( v \).

\[
u + a + b^2 = v + a + b^2 \pmod{q}
\]

\[
u = v \pmod{q}
\]
Consider \( u = 0 \) and \( v = q \). These values will always collide for any \( a, b \). Thus, the number of collisions is \( |\mathcal{H}| \), which is larger than \( |\mathcal{H}|/q \), so the family \( \mathcal{H} \) is not universal.
Consider a complete directed graph $G(V, E)$ over the vertex set $V = \{1\ldots100\}$.
That is, all edges $(i, j), i, j \in V, i \neq j$ are present in $G$. Assume that the capacity of each edge is equal to 1.
The maximum flow from the source vertex 1 to the sink vertex 2 has value 100.

**Solution:** FALSE. Consider the cut $(S, T)$ with $S = \{s\}$ and $T = V - S$. The capacity of this cut $c(S, T) = 99$ because $c(S, T) = \sum_{s \in S, t \in T} c(s, t) = \sum_{i=2}^{100} c(1, i) = \sum_{i=2}^{100} 1 = 99$. Since the flow is bounded from above by the capacity of any cut, the maximum flow can not be any larger than 99, which is less than the value of 100 asserted in the problem statement. Thus, the maximum flow can not be 100.

Assume you are given two binary search trees $T_1$ and $T_2$. Let $E_1$ be the set of elements in $T_1$, and $E_2$ be the set of elements in $T_2$. The **union** of $T_1$ and $T_2$ is a binary search tree $T$ that contains the elements from $E_1 \cup E_2$.

**Solution:** TRUE. An in-order tree walk of both trees can be carried out in $\Theta(n_1 + n_2)$ time; using a merge-sort approach, the lists can be merged in $\Theta(n_1 + n_2)$ time and duplicates can be removed at this stage without altering performance. If the binary search tree is constructed by successive insertion, the running time is greater than linear; however, because the array is sorted, a binary search tree can be created in linear time. One example would be to create an extremely unbalanced tree in which each successive element was added as the right-child of the previous element. A second example, producing a balanced tree, would be to select the middle element as the root and build the left-subarray as the left child and the right-subarray as the right child recursively.
Problem 2. Making money [15 points]
You have $1000 and want to invest it for $n$ months. At the beginning of the $t$-th month, $t \geq 1$, you must choose from the following three options:

- Purchase a savings certificate from the local bank. Your money will be tied up for one month. If you buy it at the beginning of the $t$-th month, there will be a fee of $BankFees(t)$ and after a month, it will return $BankRate(t)$ for every dollar invested. That is, if you have $c$ at time $t$, then you will have $(c - BankFees(t)) \cdot BankRate(t)$ at time $t + 1$.

- Purchase a state treasury bond. Your money will be tied up for six months. If you buy it at the beginning of the $t$-th month, there will be a fee of $BondFees(t)$ and after six months, it will return $BondRate(t)$ for every dollar invested. That is, if you have $c$ at time $t$, then you will have $(c - BondFees(t)) \cdot BondRate(t)$ at time $t + 6$.

- Store the money in a sock under your mattress for a month. That is, if you have $c$ at time $t$, then you will have $c$ at time $(t + 1)$.

Suppose you have predicted values for $BankFees$, $BankRate$, $BondFees$, $BondRate$ for the next $n$ months. Devise an algorithm that computes the maximum amount of money that you can have after $n$ months. Your algorithm should run in time $O(n)$.

Solution:
This is a dynamic programming problem. In order to solve it, several realizations have to be made:

- It is never to your advantage to split the money (we will show this formally later)

- There is a clear subproblem. At any point in time, we either got here by having put our money in a sock under a mattress, by having cashed out a treasury bond, or by cashing a certificate we got at the bank. Since we’re trying to get rich, we took the one that yielded the most profit. This represents the core of our dynamic programming solution.

- Greedy approaches fail, because if you choose to buy a treasury bond at time $t$, you’re preventing yourself from examining the option of buying said bond at time $t + 1$.

To solve this problem, we need to solve for a function $M(t)$ which represents the maximum amount of money we have at time $t$. We know that $M(0) = 1000$. We want to solve for $M(n)$. We also know that $M(1..5)$ were not reached by buying a bond. To make life easier in the definition, we will assert that $M(t < 0) = -\infty$

The following is our recursive definition of $M(t)$
\[
M(t) = \begin{cases} 
-\infty & \text{if } t < 0 \\
1000 & \text{if } t = 0 \\
\max \left[ M(t - 1) \\
(M(t - 1) - \text{BankFee}(t - 1))\text{BankRate}(t - 1) \\
(M(t - 6) - \text{BondFee}(t - 6))\text{BondRate}(t - 6) \right] & \text{if } t > 0
\end{cases}
\]

We want to solve for the above recursion at \(M(n)\). We can do so by building up a one-dimensional array. The following pseudocode does so, where we call \(\text{Max\_Money}(n, 1000)\):

\begin{verbatim}
Max_Money(time, start_money)
1  A[0] ← start_money
2  for x ← 1 to time
3     do if x < 6
4         A[x] ← MAX(A[x - 1],
5              (A[x - 1] - BankFee(x - 1)) * BankRate(x - 1))
6        else
7         A[x] ← MAX(A[x - 1],
8              (A[x - 1] - BankFee(x - 1)) * BankRate(x - 1),
9              (A[x - 6] - BondFee(x - 6)) * BondRate(x - 6))
10 return A[time]
\end{verbatim}

The above code performs a loop \(n\) times, and inside this loop it looks up a constant number of values in the array and computes a \(\max\) operation. It therefore runs in time \(\Theta(n)\).

Finally, we need to justify the correctness of our algorithm. We start from the splitting claim.

**Claim:** Splitting money does not help. I.e., for any solution that splits money, there exists a solution with the same or lesser cost, that does not use any splits.

Note that, for the purpose of grading, we were satisfied with an intuitive justification of this claim. The proof below is just for the sake of completeness.

**Proof:** Consider an optimal solution that splits money. Let \(i\) be the last day in which the solution splits the money. Suppose we put money in the sock and the bank. Let \(\epsilon > 0\) such that at least \(\epsilon\) money are put in the sock and at least \(\epsilon\) money are put in the bank. Let \(f(x) = a \cdot x + b\) be the return function for the bank. If \(a > 1\) we can route \(\epsilon\) money from the sock to the bank and we get a better solution. If \(a < 1\) we route \(\epsilon\) money from the bank to the sock and we get a better solution. Since we assumed our solution is optimal, it must be that \(a = 1\). In this case we can route the money from the bank to the sock, and get a solution at least as good as optimal, since the fees can’t be negative.

Therefore without loss of generality we can assume that the money are split between the bonds and sock/bank. Since \(i\) is the last day in which the solution splits, the money put
into bank/sock will be put into bank/sock for the next 6 months. The cumulative return function for these money is
\[
f(x) = f_1(f_2(f_3(f_4(f_5(f_6(x)))))),
\]
where \(f_j(x)\) is the return function used for these money at month \(i + j - 1\). Since \(f_j\)'s are linear, \(f\) is also linear. Because the fees cannot be negative and the rates are positive, \(f\) is of the form \(f(x) = a \cdot x + b\), where \(b \leq 0\). Now, if we look at the money put into the bonds, they also have a linear return function \(g(x) = a' \cdot x + b'\). If \(a > a'\) we route money from the bonds into the sequence of bank/sock used for the other money and we get a better solution. Similarly, if \(a' > a\), we route money from the sequence of bank/sock to the bonds and we get a better solution. Therefore, it must be the case that \(a = a'\). In this case, by routing all the money to the bonds or to the bank/sock, we get a solution at least as good as optimal which doesn’t split the money at month \(i\).

We repeat the argument for each month from \(n\) to 1 until we get an optimal solution which does not split the money.

\(\square\)

With the idea presented above that splitting is at most as good as not splitting, we know that we only have one of three choices at any point. To prove the correctness of our algorithm we will use an induction argument to justify our code.

We will assert that if the array \(A\), up till time \(t\) is filled with the correct maximal profit reachable by time \(t\), then it will also be filled with the correct maximal value for time \(t + 1\). The base case is at \(A[0]\) which clearly contains the $1000. All other positions are built up correctly from the lower optimal solutions to the subproblems.

**Incorrect Solutions:**

- The most common mistake involved attempting to solve this with a greedy solution. Either by directly picking something, or by breaking the problem up into subproblems of length 6. Neither of these solutions work, as they fail to deal with the possibility of buying 6-month-long objects a month or more in.

- Another common mistake was to get the recurrence correctly, but then to fail to correctly analyze the algorithm (as is always required). No running time analysis, or no correctness argument.

- Some people tried to argue that this was like the 0-1 knapsack problem. While this problem also required a dynamic programming solution, it was not a 0-1 knapsack problem.
Problem 3. Scheduling [15 points]

Prof. Tidor is in charge of 26-100. There are many lecturers who want to lecture in it. He is given a list of \( n \) lectures in the following format:

“lecture \( i \) needs to run from \( S_i \) to \( E_i \) (start time and end time)"

For all lectures we have \( E_i > S_i \). Note that different lectures can have different length.

Prof. Tidor wants to come up with a schedule that maximizes the number of lectures on a given day. Help the professor by designing an \( O(n \log n) \)-time algorithm that finds the maximum number of lectures that can be given that day. Justify the correctness of your algorithm.

Solution:

A greedy algorithm suffices to solve this problem. The key idea is to sort the lectures in increasing order of end times and then pick lectures greedily as long as they do not overlap with any previously chosen lectures. The following pseudocode accomplishes this:

\[
\text{Greedy Schedule}(L[1 \ldots n])
\begin{align*}
1 & \text{ Sort the array of lectures } L \text{ by end time} \\
2 & \text{ Schedule } \leftarrow \emptyset \\
3 & \text{ Current End Time } \leftarrow -\infty \\
4 & \text{ for } i \leftarrow 1 \text{ to } n \\
5 & \quad \text{ do if } (S_i \geq \text{ Current End Time}) \\
6 & \quad \quad \text{ Schedule } \leftarrow \text{ Schedule } \cup i \\
7 & \quad \quad \text{ Current End Time } \leftarrow E_i \\
8 & \text{ return Schedule }
\end{align*}
\]

The sorting step takes \( O(n \log n) \) time while the greedy scan is accomplished in \( O(n) \) time. Thus the overall running time is \( O(n \log n) \).

It is obvious that Greedy Schedule returns a valid schedule, since it only picks lectures that are disjoint from those picked so far. We prove optimality of the solution using a standard exchange argument. Let \( G \) be the greedy schedule and \( O \) be an optimal schedule that differs from \( G \) and has more lectures than \( G \). Let \( G = (l_1 l_2 \ldots l_k) \) and \( O = (l_1' l_2' \ldots l_m') \) be the respective schedules arranged in increasing order of endtimes. Let \( i \) be the first position at which the schedules differ. Clearly \( l_i \) has no conflicts with any lectures \( l_j', j < i \) and since \( E[l_i] \leq E[l_i'] \), it does not have conflicts with any lectures \( l_j', j > i \). Thus we can safely replace \( l_i' \) by \( l_i \) in the optimal solution and still have a valid solution with the same number of lectures. By repeating this argument inductively, we create a solution identical to the greedy solution which has the same number of lectures as the optimal. This contradicts
our “greedy not being optimal” assumption. Hence, the greedy solution must indeed be the optimal solution.

Many students had the right intuition about using a greedy algorithm, but their greedy choice was incorrect. In particular, greedily choosing lectures either by their length or start time doesn’t work. For the former case, \{ (10, 12), (8, 11), (11, 20) \} provides a counterexample. The Greedy by length algorithm returns \{ (10, 12) \} while the optimal solution is \{ (8, 11), (11, 20) \}. For the latter case, a counterexample is \{ (8, 12), (9, 10), (10, 11) \}. The Greedy by start time algorithm returns \{ (8, 12) \} while optimal is \{ (9, 10), (10, 11) \}.

Yet another popular approach to this problem involved the use of dynamic programming. Suppose we initially sort the lectures by ending time. As subproblems, we let \( M(j) \) denote the maximum number of lectures from lectures 1 \ldots j that we can schedule. We can compute \( M(j) \) recursively according to the following formula:

\[
M(j) = \max \{ \max \{ M(i) + 1 : E[i] \leq S[j] \}, M(j - 1) \}
\]

The two expressions in this formula correspond to the optimal solutions when we respectively either use lecture \( j \) or don’t use lecture \( j \). A straightforward dynamic programming algorithm based on this formulation runs in \( O(n^2) \) time, since there are \( n \) subproblems each of which takes \( O(n) \) time to solve. However, we can reduce the running time to \( O(n \log n) \) by noting that \( M(j) \) is a non-decreasing function of \( j \) (this is clear from the recursive formula above, since \( M(j) \) cannot be less than \( M(j - 1) \)). Since this is the case, the value of \( i \) that maximizes \( M(i) + 1 \) over all \( E[i] \leq S[j] \) will simply be the largest \( i \) such that \( E[i] \leq S[j] \), and we can find such an \( i \) via binary search since the array of lectures is sorted by ending time.

Several students came up with the innovative idea of attempting to reduce this problem to a shortest path problem. Specifically, we can create a graph with a node corresponding to each lecture, and an edge from lecture \( i \) to lecture \( j \) if \( E[i] \leq S[j] \). Additionally, we add a source node \( s \) with an edge to every lecture node, and we add a sink node \( t \) and an edge from every lecture to the sink. Observe for any path through this graph from \( s \) to \( t \), the nodes along this path will correspond to a valid (i.e. disjoint) set of lectures, so we want to find the longest path from \( s \) to \( t \) in this graph. Many students argued, incorrectly, that the longest path problem can be solved via trivial modifications to Dijkstra’s algorithm or breadth-first-search, but this is not the case. The longest path problem is equivalent to a shortest path problem when we negate the length of every edge. For most graphs in practice, this transformation creates negative-length cycles and hence the resulting shortest path problem is not well-defined (this is why we tend to study the shortest path problem rather than the longest path problem). In our case, our graph will be acyclic, so the transformation creates negative-length edges but no negative-length edges. In such a graph, we can solve for shortest paths using either the Bellman-Ford algorithm or the Floyd-Warshall algorithm, both of which will take \( O(n^3) \) time. In fact, one can solve for shortest paths through an acyclic graph in linear time (we didn’t learn this algorithm in class, and nobody attempted to use it), which in this case is \( O(n^2) \), since the graph can have \( \Theta(n^2) \) edges. In any case, although this approach is correct it cannot be made to run in \( O(n \log n) \) time.
Problem 4. Bottleneck paths [20 points]
Consider the following data structure problem. Initially, you are given a threshold \( T > 0 \), and an undirected graph \( G \) with a set of vertices \( V = \{1\ldots n\} \) and no edges. Then, you are given a sequence of requests of the following two types:

- **ADD\((u, v, c)\):** add an undirected edge \( \{u, v\} \) to \( G \); the edge has capacity \( c > 0 \).
- **TEST\((u, v)\):** check if there is a path between \( u \) and \( v \) in \( G \) such that each edge in the path has capacity at least \( T \)

Your goal is to design efficient data structure(s) for this problem. You are allowed to spend polynomial time to initialize your data structure (i.e., before any ADD or TEST operation is performed).

(a) [14 points] Assume that each edge is added at most once. Design an efficient data structure for this problem, with low amortized running times. For full credit, your data structure should perform both operations in amortized \( o(\log n) \) time (i.e., better than \( \Theta(\log n) \)-time). Partial credit for less efficient data structures will be given.

(b) [6 points] Assume now that each edge can be added several times, and the capacities sum up. I.e., if we add an edge \( \{u, v\} \) \( k \) times with capacities \( c_1, \ldots, c_k \), then the capacity of the edge is equal to \( \sum c_i \). Design an efficient data structure for this case. For full credit, the amortized time per operation should be again \( o(\log n) \). Partial credit for less efficient data structures will be given.

Solution:

(a) We start from the intuition behind the solution. The first thing to note is that the data structure can safely ignore all edges with capacities less than \( T \). This is because the operation TEST\((u, v)\) returns true if and only if there is a path between \( u \) and \( v \) consisting exclusively of edges with capacities \( \geq T \); edges with capacities \( < T \) could as well not exist at all. Thus, the ADD\((u, v, c)\) procedure can do nothing at all if \( c < T \). If \( c \geq T \), then the edge should be recorded, since it could be a part of a “high-capacity” path connecting some two vertices. However, the actual capacity value is not important; it is only relevant that it is at least \( T \).

Therefore, we can solve the problem as follows. We are going to maintain an unweighted undirected graph \( G' = (V, E') \). Whenever the operation ADD\((u, v, c)\) is performed, we check if \( c \geq T \). If not, we do nothing. If yes, we add an edge \( \{u, v\} \) to \( G' \). In order to answer TEST\((u, v)\), we check if there is any path between \( u \) and \( v \) in \( G' \). In other words, we check if \( u \) and \( v \) are in the same connected component of \( G' \).

To implement the above, we need a data structure that maintains \( G' \) and supports the “add” and “check” operations. This is precisely the problem of Maintaining Dynamic Connectivity Information, described in lecture 10; a variant of it can be found in CLRS, p. 500. It can be efficiently solved (see below) using a data structure for maintaining disjoint sets.
In particular, if we use the implementation of disjoint sets based on forest of trees with path compression, then both add and check can be done in amortized $O(\alpha(n))$ time, where $\alpha(n) = o(\log n)$ is the inverse Ackermann function.

For completeness, we describe how to implement $\text{add}$ and $\text{check}$ in $G'$ using disjoint sets. To add $\{u, v\}$, we simply perform $\text{Union}(u, v)$. To check if $u$ and $v$ are in the same connected component of $G'$, we simply check if $\text{Find}(u) = \text{Find}(v)$.

One could wonder what to do about the $\text{Makeset}$ operations, needed to initialize the set $V$. It could be done “on-line”: whenever an edge containing a new vertex $u$ shows up, we could perform $\text{Makeset}(u)$. However, since we know $n$ in advance, it is much simpler to perform $\text{Makeset}(1), \text{Makeset}(2) \ldots \text{Makeset}(n)$ during the preprocessing phase. Note that this takes only linear time.

(b) We can solve this more general case by “accumulating” the capacities of the edges, and performing the steps as above only if the “accumulated” capacity of an edge exceeds $T$. This can be done as follows. During the preprocessing, we create a matrix $C[1 \ldots n, 1 \ldots n]$ and initialize all of its entries $C[u, v]$ to 0. To implement $\text{Add}(u, v, c)$, we perform $C[u, v] = C[u, v] + c$. If $C[u, v] \geq T$, then we perform $\text{Union}(u, v)$. Note that it is OK to perform the $\text{Union}(u, v)$ operation several times (when the value of $C[u, v]$ keeps increasing). In practice, though, one would probably optimize it a little bit.

To implement $\text{Test}(u, v)$, we again check if $\text{Find}(u) = \text{Find}(v)$.

**Alternative solutions:** The above approach (and its variants) appears to be the only way to get the optimal solution. However, several people used the code of disjoint-set implementations directly, without realizing (or mentioning) it. The resulting solutions were less clean, but still worth full credit if they achieved the $o(\log n)$ amortized time. Note that using other implementations of disjoint sets (e.g., linked lists or forest of trees without path compression) leads only to $O(\log n)$ time.

A significantly less efficient solution (still worth half of the points for part (a)) is to implement the $\text{Test}$ operation by recomputing connected components every time we perform $\text{Add}$ or $\text{Test}$. This results in $O(n)$ time bound. Variants of this approach were proposed. In particular, the graph $G'$ was often not explicitly defined. Instead, the comparison of the capacities with $T$ was often done inside DFS.

An even less efficient (but correct) solution is to check if $u$ and $v$ are connected by enumerating all possible paths between $u$ and $v$ and checking if all edges on them have capacity at least $T$. However, this results in exponential time.

**Incorrect solutions:** Many people gave an algorithm that tests if there exists an edge between $u$ and $v$ that has capacity at least $T$. There were numerous implementations of this approach, involving adjacency matrix, as well as hashing, B-trees, Red-Black trees etc. Unfortunately, the resulting data structures solve a different (and much simpler) problem.

A few people tried to use a shortest path algorithm (Dijkstra, Bellman-Ford etc). Unless this was done on $G'$ (in which case the shortest path problem is essentially equivalent to the
connectivity problem), the resulting algorithm was incorrect, since the shortest path could easily consist only of edges with very low capacities. A similar comment holds for the longest path (although the counterexample is a little more complex) and max-flow.

Some people tried to augment specific implementations of disjoint sets with capacities (instead of using them as a black box). The resulting solutions were either incorrect or not very efficient.

**Conclusion:** When possible, it is good to try to solve a problem by a clean, black-box reduction to a known problem. Very often, the result is not only simple, but also efficient.