UNIVERSITY of OSLO

Faculty of Mathematics and Natural Sciences

Exam in:	**INF4130/9135: *Algorithms: Design and efficiency***
Date of exam:	**18th December 2018**
Exam hours:	**09:00 – 13:00 (4 hours)**
Attachment:	**One**
Exam paper consists of:	**7 pages (including the attachment)**
Permitted materials:	**All written and printed'**

Including proposals for solutions

Make sure that your copy of this examination paper is complete before answering.

You can give your answers in English or in Norwegian, as you like. Tear the attachment page out, fill in your answers to Question 6.b, and deliver it with the white version of your answers.

Read the text carefully, and good luck!

Assignment 1 Undecidability (18 %)

Alice and Bob discuss the curriculum concerning undecidability. They define the following two formal languages:

NO-MACHINE = {<*M*> | Turing machine *M* is a decider, and rejects every input}.

MAYBE-NO-MACHINE = {<*M*> | If Turing machine *M* halts, then it rejects its input}.

Question 1.a

For the four Turing machines defined below, help Alice and Bob to decide if $<M_i>$ is in the language NO-MACHINE, MAYBE-NO-MACHINE, both, or none.

1. M_1 = "On input *w*:
   (1) reject."

2. M_2 = "On input <*M*, *x*>:
   (1) Simulate *M* on *x*.
   (2) If *M* accepts, reject.
   (3) If *M* rejects, reject."

3. M_3 = "On input *w*:
   (1) if $w \neq w$: accept
   (2) else: reject."

4. M_4 = "On input *w*:
   (1) if $w = 010$: accept
   (2) else: reject."

1. $<M_1>$ is in both NO-MACHINE and MAYBE-NO-MACHINE, as it always rejects all inputs.
2. $<M_2>$ is only in MAYBE-NO-MACHINE, as it may loop for ever while simulating M on x.
3. M_3 will reject all inputs, so $<M_3>$ is in both languages.
4. M_4 accepts 010 so $<M_4>$ is not a member of any of the two languages.

Question 1.b
Alice claims that NO-MACHINE $\subseteq$ MAYBE-NO-MACHINE. Bob claims that Alice is wrong. Explain briefly why Alice is correct.

Alice is correct since a member of NO-MACHINE is a TM that always halts and rejects. Thus the same TM's will be members of MAYBE-NO-MACHINE (the set of TM's that if they halt, reject).

Question 1.c
Prove via a reduction from the halting problem that NO-MACHINE is undecidable.

This is done via a straight forward reduction, like we did in the lectures. On input <M, w>, we create a Turing machine M'. The machine M' simulates M on input w, then rejects.

Now, if M' is in NO-MACHINE, then we know that M' is a decider, so that M halts on w. If <M'> is not in NO-MACHINE then M cannot have halted on w, since otherwise M' would have rejected its input.

Question 1.d
Now, Alice reasons as follows: Since NO-MACHINE is undecidable and MAYBE-NO-MACHINE contains NO-MACHINE, we can conclude that MAYBE-NO-MACHINE is also undecidable. Bob says that Alice's proof is invalid. Who is correct? Explain your answer briefly.

Alice's claim does not hold. A counterexample could be the language containing all strings over the alphabet. This language obviously contains NO-MACHINE, and it is trivially decided by the Turing machine that accepts any input. Therefore Bob is correct.

Assignment 2 NP-completeness (15 %)

Question 2.a
We will here investigate a decision problem called **Multi-Way-Partitioning** [MWP]. In MWP you are given a multiset (that is, a set where values can be repeated) of natural numbers S, and a natural number m. The question is whether or not it is possible to partition S into m subsets summing to the same. Let

$$\text{MWP} = \{<S, m> \mid S \text{ can be partitioned into } m \text{ subsets, all summing to the same value}\}.$$

Is MWP NP-complete? Prove your answer. (You may assume that P ≠ NP.)

The language MWP is NP-complete. First, we need to show that MWP is in NP. Given a yes-instance of MWP, we can use a partition of size m as our certificate. Such a certificate has polynomial size in the length of the input and with it, we can confirm that we have a yes-instance in polynomial time.

The next step is to prove that MWP is NP-hard. We prove this via a reduction from PARTITION (which is an NP-complete language discussed in the lectures). An instance of PARTITION is a multiset S and the question is whether S can be partitioned into two subsets, both summing to the same value. We see that PARTITION is just a special case of MWP, so proving PARTITION $\leq_p$ MWP should not be too tricky. Given an instance <S> of PARTITION we create the instance <S, 2> of MWP. If <S> is a yes-instance of PARTITION, then S can be partitioned into two subsets summing to the same, so <S, 2> is a yes-instance of MWP. If <S, 2> is a yes-instance of MWP then, by the same argument, <S> is a yes-instance of PARTITION. The reduction is trivially done in polynomial time.

Question 2.b
We now define the following problems. Notice that we are defining one language for each *k*, so that *k* is not a part of the input (like for 2-SAT, 3-SAT, etc.). Let

MWP-*k* = {<*S*, *m*> | S can be partitioned into m subsets of size at most *k*, all summing to the same value}.

Example: <{1, 2, 3}, 2> is a yes-instance of MWP-2, but a no-instance of MWP-1, since one subset will need to have size 2.

Finally, we define the problem L to be the union of all MWP-k problems. That is:

$$L = \bigcup_{k=1}^{\infty} \text{MWP-}k$$

Is L NP-complete? Prove your answer. (You may assume that P ≠ NP.)

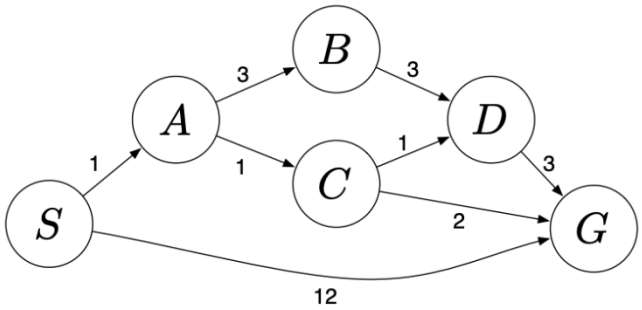
By the definition, L equals MWP, which was proven NP-complete in the last question above.

Question 2.c
Is MWP-3 NP-complete? Explain your answer. (You may assume that P ≠ NP.)

The language MWP-3 is NP-complete. However, the proof was more complicated than first assumed, so it is not given here. Thus, during grading this question was given a very low weight, but those candidats that made relevant remarks got some extra credit.

Assignment 3 A*-search (18 %)

We are given the following graph and want to use the A* algorithm to find the shortest path from *S* to *G*.



The proper A* algorithm requires a so-called *monotone* heuristic. With a monotone heuristic the algorithm never needs to move nodes back into the priority queue once they have entered the tree.

(The text below was meant as Question 3.a, but the corresponding headline disappeared during the finishing process. This doesn't seem to have caused any trouble)

Below are two proposed heuristics h_1 and h_2. For each node *N*, the *h*-value is an estimate of the distance between *N* and *G*.

Node	$\boldsymbol{h_1}$	$\boldsymbol{h_2}$
S	4	4
A	3	3
B	6	7
C	1	1
D	3	3
G	0	0

Answer the following questions:

i) Is h_1 monotone? Justify your answer.
ii) Is h_2 monotone? Justify your answer.

Heuristic h_1 is not monotone. The monotonicity requirement is broken in A.

Node	$\boldsymbol{h_1}$	
S	4	
A	3!	Must have H(A) ≤ H(C) + cost(AC) Have 3 > 1 + 1 Shortest path is 3, still an underestimate
B	6	OK, har H(B) ≤ H(D) + cost(BD) 6 ≤ 3 + 3 Shortest path is 6, still an underestimate
C	1	OK, shortest path is 2
D	3	OK, shortest path is 3
G	0	OK

Heuristic h_2 is not monotone. We can right away see that the heuristic is not even an underestimate for B. The monotonicity requirement is broken in B and A.

Node	h_1	
S	4	
A	3!	Must have H(A) ≤ H(C) + cost(AC) Have 3 > 1 + 1 Shortest path is 3, still an underestimate
B	7!!	Must have H(B) ≤ H(D) + cost(BD) Have 7 > 3 + 3 Shortest path is 6, not an understeimate
C	1	OK, shortest path is 2
D	3	OK, shortest path is 3
G	0	OK

Question 3.a

Both heuristics h_0 and h_{exact} given below are monotone. (You need not prove this.) Create another monotone heuristic h' for the graph by filling in a table with your values. (The misprint for D below doesn't seem to have caused any trouble. A few students noticed it, and got some extra credit)

Node	h_0	h_{exact}
S	0	4
A	0	3
B	0	6
C	0	2
D	0	4 (misprint: Should have been 3)
G	0	0

A simple monotone heuristic can be constructed by viewing all edges as a step of length 1, in concentric circles out from G. But there are many possibilities.

Node	h'
S	1
A	2
B	2
C	1
D	1
G	0

Question 3.b

We now look at the general case, not the specific graph above. As stated in the introduction, the A*-algorithm requires a monotone heuristic. Heuristics with other properties will have

some form of negative effect on the algorithm. They may cause the algorithm to fail, or behave differently. It will then technically no longer be the A*-algorithm.

What is the effect of the following three general kinds of heuristics? Explain briefly why the resulting algorithm will fail, or what the difference to the A*-algorithm will be if the heuristic can be used.

i) h_{i} is not monotone, but an underestimate. That is, *h is* never larger than the actual shortest path to the closest goal node.

ii) h_{ii} is neither monotone, nor an underestimate. That is, *h* may be both larger and smaller than the actual shortest path to the closest goal node.

iii) h_{iii} is always 0.

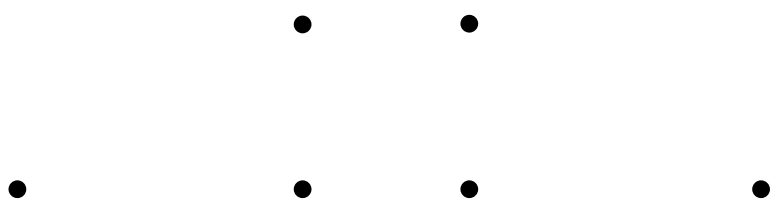
With h_{i}, which is an underestimate, the algorithm will work. We will technically have a different algorithm, called A. With the A-algorithm we have to re-process nodes in the tree, nodes that we thought we were done with, if we find a shorter path to them (we would have been done with them in the A*-algorithm). With a monotone heuristic the A*-algorithm finds the shotest path the fist time. With a heuristic that is only an underestimate, we may find shorter paths to nodes we have processed earlier. The A-algorithm will find the correct answer, but will not be as (time) efficient as the A*-algorithm.

With h_{ii} that is not even an understeimate we cannot guarantee a correct answer. A heuristic value that is too high can trick the algorithm to follow a path that looks "low" and promising, but is wrong. (And the algorithm terminates when it reaches G.)

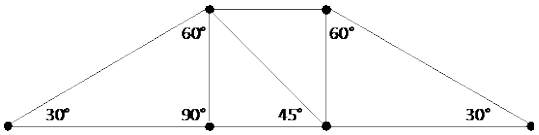
With $h_{\text{iii}} = 0$ we "only" have Dijkstras algorithm, which lacks the efficiency we get from focusing our search with a heuristic.

Assignment 4 Triangulation (10 %)

We are given the following points in the plane.

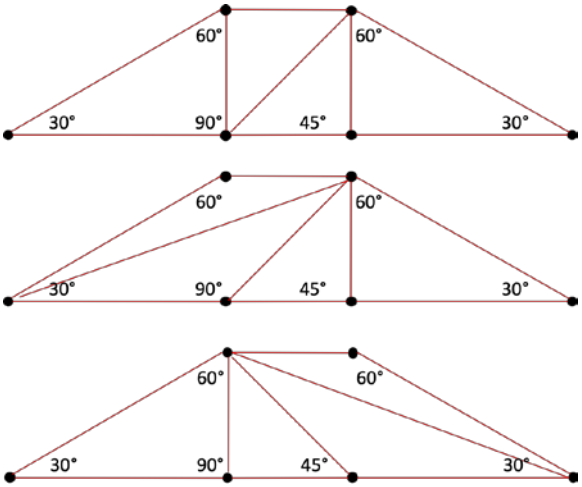


Below is a possible triangulation. All angles are shown or immediately inferable, and they also define the shape and the relative distances of the figure formed by the above points.



Question 4.a

There are more possible triangulations. Draw all the remaining ones, including symmetric variants.



Question 4.b

Which triangulations are Delaunay triangulations, if any? Explain why they are of the Delaunay type.

The triangulation shown in the text, and it symetric variant, are Delaunay triangulations. This is an example of a situation where the Delaunay triangulation is not unique (four points along the circumference of a circle like we have in the middle of the drawing).

Sorting the angles of the triangualtion in the text we get:

30, 30, 45, 45, 45, 45, 60, 60, 90, 90, 90, 90

Looking at the angles in the other triangulation we cannot immediately know what the angles are, but we can see that at 30-degree angle is divided in two, and that a 45-degree angle is divided in two, compared to the triangulation in the text. (The actual values are 9,9 degrees, 20,1 degrees and 24,9, but we can call the values X, Y and Z, with X+Y=30 and Y+Z=45.

Sorting the angles of that triangulation we get:

X (<30), Y (<30), Y (<30<45), Z (<45), 30, 45, 45, 60, 90, 90, 135, 150.

The first sequence is obviously lexicographically larger than the second. (and there are no other triangulations). This the definition (one of them) of a Delaunay triangulation.

Assignment 5 Dynamic programming (18%)

An array “integer array L[N]”, indexed from 0 to N-1, contains positive integers. We want to find the largest sum we can get from a subset of these integers, when a subset is not allowed to contain numbers that occur next to each other in the array.

Question 5.a

Explain what sort of table or tables you want to use for solving this task with dynamic programming. What is the meaning of the different entries of the table(s)?

Hint: At each point during the process you have to make a difference between the sum you can obtain by including the current value of L and by not including it.

(The hint above was meant to indicate the solution given below, but there are also other solutions, even one using only one array “integer B[ ]”, where B[i] simply contains the best sum possible (with the given restriction) for the interval [0 : i]. Also solutions with a two dimentionl boolean array are possible, but more difficult to get correct.)

You can use two integer arrays “include” and “exclude”with the same indexing as L, that is: “integer array include, exclude [N]” (or something similar).
Here

- “include[i]” should have the best sum you can get (under the above conditions) of L[0], …, L[i] when including L[i]
- “exclude[i]” should have the best sum you can get (under the above conditions) of L[0], …, L[i-1]. (Note: Here L[i] is NOT included).

Question 5.b

- Write down the recurrence-formula you will use to fill the table(s).
- Indicate how you want to initialize the table(s).
- Draw the suitable table for the example L = [3, 1, 1, 4, 1, 5] and fill it in according to your initialization and formula

The recurrence formula will be

include[i] = exclude[i-1] + L[i]

Reason: L[i-1]) should not be included, but otherwise the best from L[0], …, L[i-2].

exclude[i] = max(exclude[i-1], include[i-1])

Reason: The best we can get by either including or not including L[i-1]

The initialization will be:

include[0] = L[0]

exclude[0] = 0

Index L	0 3	1 1	2 1	3 4	4 1	5 5
include	3	1	4	7	5	12
exclude	0	3	3	4	7	7

Question 5.c

Write a method/function that receives an array L as a parameter, and computes the answer to the above question. Make sure that the maximal sum for the actual L is delivered as a result of the method. You can e.g. use statements like “int A[ ] = new int[L.length]”.

```
integer maxSum ( int[ ] L) {
    int N = L.length;
    int include[ ] = new int [N];
    int exclude[ ] = new int [N];

    include[0] = L[0]
    exclude[0] = 0

    for (int i =1, i < N, i++) {
       include[i] = exclude[i-1] + L[i];
       exclude[i] = max( exclude[i-1], include[i-1] );
    }
    return max(include[N-1], exclude[N-1]);
}
```

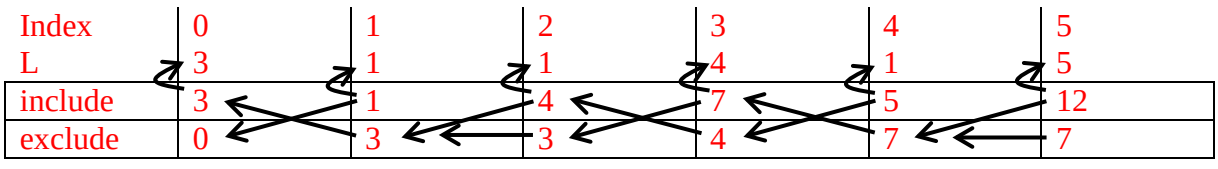
Question 5.d

Suppose we also want the indices in L of the numbers included in the sum. How can you compute these indices, and how much extra space do you need?

There are two reasonable ways to do this:

1. To look at the content of the arrays “include” and “exclude” after they are filled, and use this to trace backwards what choices were made along the way, and thereby find the way the maximum sum was formed.
2. Use some extra data structure to register along the way information that makes it easy to afterwards find the indices of the included values.

We look at the first variant, and different versions of the second variant can easily constructed following the same idea. For the first variant we look at the example from 5.b, and we indicate by arrows from which alredy computed values the values of exclude[i] and include[i] (i = 0, …, N-1) are copied or computed. See drawing below.



The only thing that varies from index to index here is whether exclude[i] is taken from exclude[i-1] or include[i-1]. That is, whether it best to include L[i-1] or not. We can then trace backwards from the actual best sum we can get from the whole of L, and highlight the path used. We the get the following picture.

Index	0	1	2	3	4	5
L	3	1	1	4	1	5
include	3	1	4	7	5	12
exclude	0	3	3	4	7	7

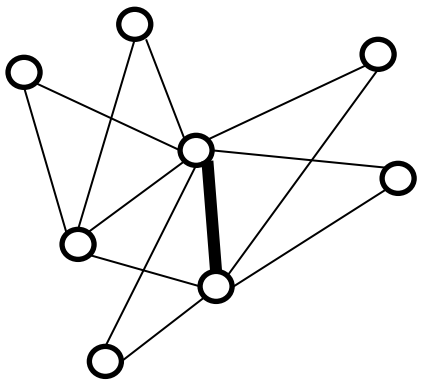
It here becomes obvious that the values used to get the sum 12, are those at index 1, 3 and 5. A program computing these indices could look like the one below (but this was not expected in an answer!)

```
boolean [ ] usedIndices (int[ ] include, exclude ) {  // The arrays should be filled as above
    int N =include.length;
    boolean[ ] answer  =  new boolean[N];
    UseCur = false;      //  Tricky initialization, useCur for index: N

    for (int i =N-1, i >=0, i--) {
       if (useCur ) {                              // Value of useCur for index:  i+1
           useCur = false;  answer[i] = false;  // Cannot include this index
       } else {
           useCur = include[i]  > exclude[i];   // If equality, anyone can be used
           answer[i] = useCur;
       }
    }
    return answer;
}
```

Assignment 6 Matching (14%)

We are given the following graph G, with the indicated matching M with one edge.



Question 6.a

Give an intuitive reason for why no perfect matching can be found in this graph.

We can observe that the three nodes: two to the right and one under, is connected to only two nodes in the rest of the graph, and that there are no connections between these three nodes.

Thus all of these three nodes can obviously not be an end-node of an edge in a matching. This is the same argument as can always be used for bipartite graphs, but by coincidence it can also be used locally on this general gtraph.

Question 6.b

Run the Extended Hungarian Algorithm on the above graph, starting at the indicated matching, and making larger and larger matchings for as long as possible. For each larger matching make a new drawing of the graph, and indicate in the previous drawing what augumenting path you used to obtain it.

The algorithm will stop before we have a perfect matching (see above). Your task is to show how the situation is (it will differ, depending on your choices) when the algorithm stops with the conclusion that no larger matching exists.

Notice: In the Appendix a neutral version of G is drawn in a number of copies, and you can use these to show the situation as you get larger and larger matchings. You should draw the final situation on the appropriate copy. Tear out the sheet, and deliver it together with the *white* copy of your answers.

Directions for drawing the final situation:

- The root node of the tree you are building should be indicated with an extra circle around it. The edges of the current matching should be marked with a thick line.
- "Colors" of the nodes made during the treebuilding process should be indicated as follows:
    - The root and all nodes with the same color (red on the slides) should be given a cross within the circle.
    - All nodes with the other color (blue on the slides) should have their circles fully filled.
    - Nodes that are not seen by the tree building process should be empty inside.
- The edges that are included in the tree, but are not in the matching, should be given an arrow showing in which direction the edge was followed during the (last) tree building.
- Subsets of nodes that, during the treebuilding, are «collapsed» to one «super-node» should get a closed curve around them, but the original nodes should retain their "old" color (contrary to what was done on the slides).

(The slides that explain the Extended Hungarian algorithm (for finding the largest matching in a general gtraph) can be found at slides 13, 14 and 15, for September 27, 2018.

The answer is given at the attatchment page at the end of this document. The current matching for each step is given by thick edges, while the augumenting path giving the next matching is indicated by dashed edges. There will be no collapses of odd cycles in the final tree building. See further comment at the attachment page

Assignment 7 String Search (7%)

We want to search for the pattern:

```
eucatastrophe
```

in a string:

```
a eucatastrophe is a dramatic event that leads to the
protagonist´s well-being
```

With the Horspool algorithm; for which letters of the alphabet will we get a maximum shift of the pattern? And what is the length of this maximum shift? Assume the English alphabet a – z.

We are only asked what letters of the alphabet we get a maximum shift. All letters not occurring in `eucatastrophe` result in a maximum shift (13). They are:

b,d,f,g,i,j,k,l,m,n,q,v,w,x,y,z.

The actual shift values are as follows, but his was not asked for:

a: 7

c: 10

e: 12

h: 1

o: 3

p: 2

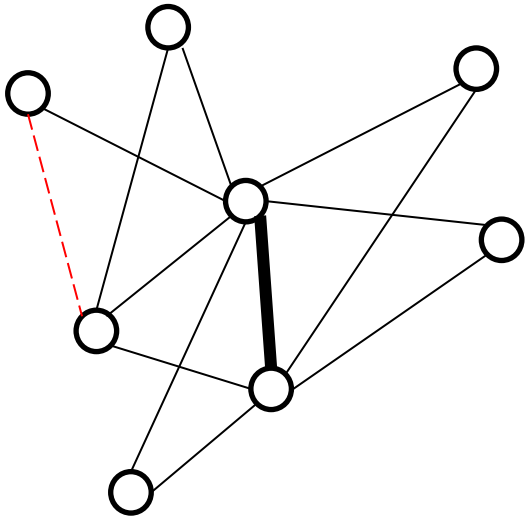
r: 4

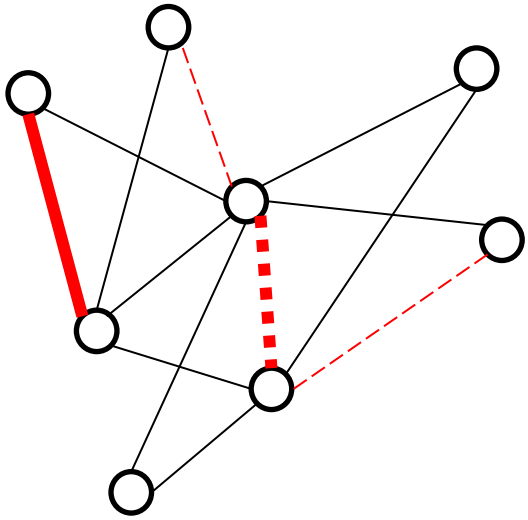
s: 6

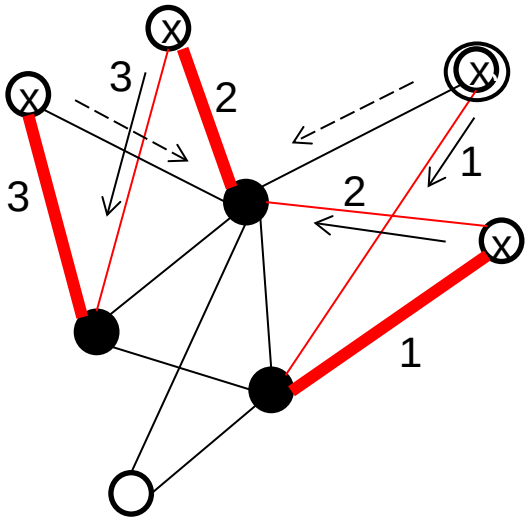
t: 5

u: 11

[End of exam. Next page is the attachment]







Comments:

With the used choice of order for looking at edges from "red" nodes (with an X inside), we get the tree indicated with red (thick or thin) edges.

The numbers on the tree edges (not required!) give the order in which the edges was inspected, and choosing a different order will produce a different, but similar, tree.

The dashed (black) arrows indicate edges that was followed, but did not lead to any action (they lead from red to an already blue node, and they were not asked for).

There will be no collapses of odd cycles!

The lower unmatched node will never be included in the tree, and therefore no augumenting path can be found between the two unmatched nodes (as was expected from the answer to 6.a).