UNIVERSITY of OSLO

Faculty of Mathematics and Natural Sciences

Exam in:	**INF 4130/9135:** ***Algoritmer: Design og effektivitet***
Date of exam:	**14th December 2012**
Exam hours:	**09:00 – 13:00 (4 hours)**
Exam paper consists of:	**4 pages**
Appendices:	**None**
Permitted materials:	**All written and printed**

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

Read the text carefully, and good luck!

Assignment 1: Search in strings with Boyer-Moore (18%)

Question 1.a (6%)
We will work with an alphabet that consists of the nine first letters of the standard a-z alphabet: { a, b, c, d, e, f, g, h, i }. We are given a string P = "abcifbei" to search for in a longer string T. Set up the table of Shift values for the Boyer-Moore algorithm (the simplified version discussed in the textbook, which is called Horspool on the slides). You can use the letters themselves as indices to the Shift table/array.

Answer 1.a

The Shift-table will be as follows:

a	b	c	d	e	f	g	h	i
7	2	6	8	1	3	8	8	4

Question 1.b (6%)
Suppose that we do string search with the Boyer-Moore algorithm (the same as in 1.a), and that m is the length of the pattern P and n is the length of the string T (and that n is much larger than m). By coincidence in a search, the last symbol of P does not occur elsewhere in P and not at all in T. Estimate the number of tests for symbol equality that will be made during the search.

Answer 1.b

The Shift-table entry for the last charcter in P will be n. During the search we newer get a match in T for the last symbol in P, and we therefore get (one test for equality and a full shift) repeted until we reach the end of T. Thus we get about n/m tests for symbol equality (and all of them gives **false**).

Question 1.c (6%)
We will study what will happen if P consists of the letter 'a' repeated m times (that is: a^m), and T is a string that is a repetition r times of the string b a^{m-1}. How many test for symbol equality will be made during the search?

Answer 1.c

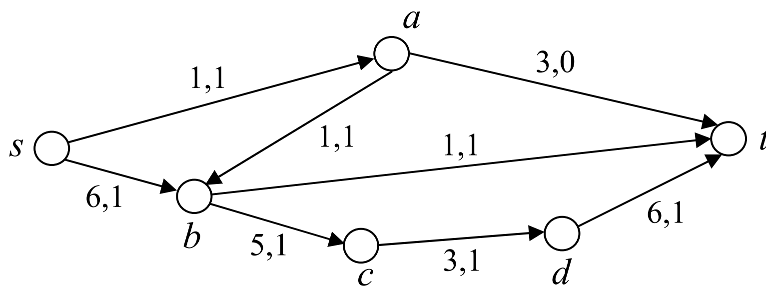
The Shift-table for P will be 1 for a, and m (full shift) for all other symbols. We assume that $r \geq 1$. The first test on the last symbol of P and the corresponding symbol in T will give **true**, and this will result in tests for all the rest of P, together that is m tests. Only the last one will give **false**. We then get a shift of 1, but after this the first test will give **false** (T has b), and we get a full shift. Then the test will again be **false** (again T has b), and this will continue to the end of T. The number of tests will therefore become: $m + r - 1$

Assignment 2: Flow in networks (18%)

Question 2.a (9%)

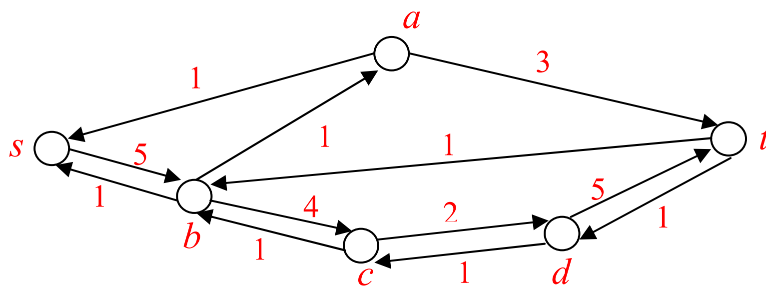
Consider the network N with capacities given in the picture below. We want to maximize the flow from s to t. On each edge two numbers are written which are the capacity and the current flow, respectively. We want to use the Ford-Fulkerson algorithm, but we will (contrary to what Edmonds and Karp proposed) use the rule that we in each step use the augumenting path that gives largest increase in flow.

What, then, will be the next step from the flow indicated in the figure below? As part of the solution, you should show the graph N_f.



Answer 2.a

The graph N_f will look like this:



Here we see that the path from s to t that can increase the flow as much as possible, is < s b c d t > The flow can here be increased by 2, while we can only increase it by 1 on the only other path <s b a t>. Thus the next step will be to increase the flow along < s b c d t > with 2.

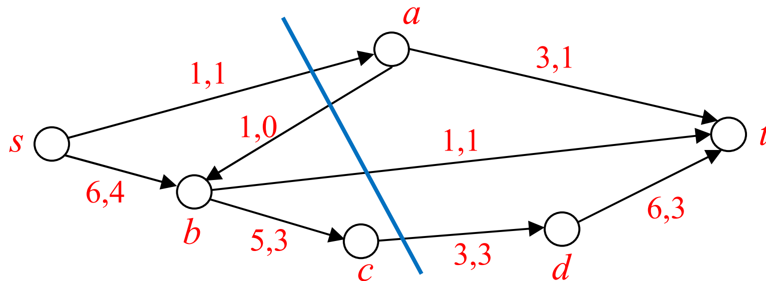
Question 2.b (9%)

Find a maximum flow for the network above. Draw your own copy of the network above with the

same capacities, but with what you claim is a maximum flow. You don't need to show how you found that flow, but you should give a proof showing that it really is as large as possible.

Answer 2.b

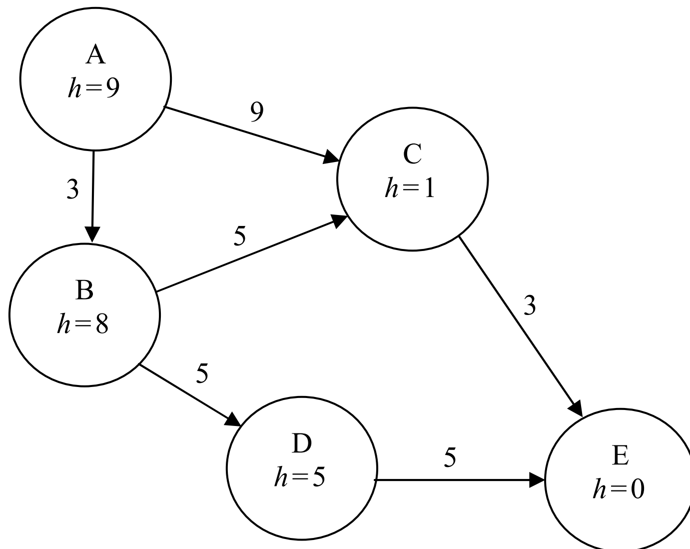
We claim that the following flow of 5 is a maximum flow. (You can obtain it from the result of 2.a by increasing the flow along the path <s b a t> by one).



The proof that this flow is the largest possible is the CUT marked with a blue line above, which has capacity 5. Thus no flow can be larger than 5.

Assignment 3: A-search and heuristic functions (20%)*

We want to find the shortest path from A to E in the directed graph given below. For this we will use the procedure *A*-SearchMH(...)* in the textbook.



Here the number on each edge is the *length* (or *cost* or *weight*) of that edge, and the value for h in a node is the heuristic value for the distance to E from that node.

Question 3.a (7%)

Perform the procedure A*-SearchMH to make a search from A, with E as the goal node (and make sure you follow that procedure exactly). Describe the steps you pass through by using one line for each time a node leaves the priority queue. Indicate to the left at this line the node that leaves the queue and to the right the set of new nodes that then enter the priority queue. The first step (and line) should be to insert A into the priority queue.

Also: Make your own drawing of the above graph, and indicate in it the f and g values you get for

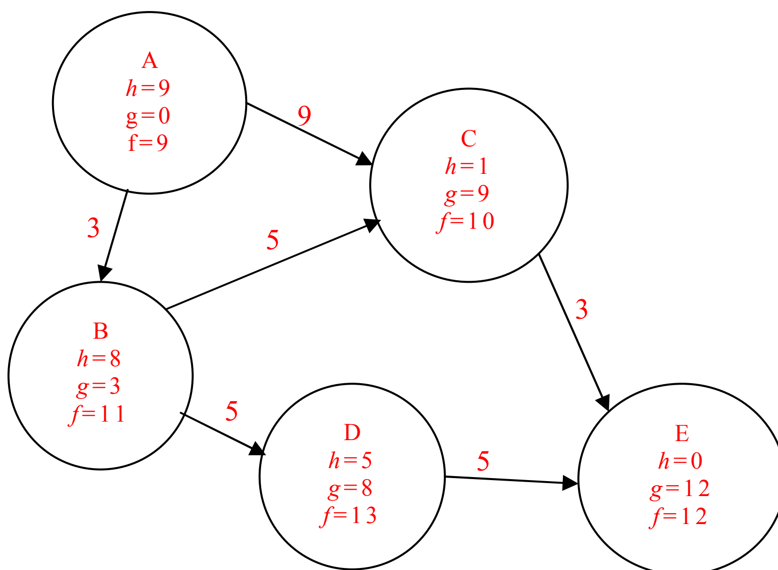
the nodes.

Answer 3.a

The algorithm will produce the sequence of events given below. The values in parenthesis are the *g* and *f* values respectively.

Start	A (0, 9)
A	B (3, 11) C (9, 10)
C	E (12, 12)
B	D (8, 13) (C could also get new values, but it is already taken out of the queue)
E	Thus, the goal node is taken out of the queue, and this signals the end of the algorithm

Thus, this algorithm concludes that the length of the shortest path from A to E is 12 (the *g*-value of the goal node E).



Question 3.b (7%)
Did the search in 3.a give the correct answer (that is, did it give the length of the shortest path from A to E)? If not, explain why.

Answer 3.b

But, the length of the shortest path from A to E is not 12, but 11: < A B C E >. The reason for the wrong answer is that the algorithm A*-SearchMH requires a *monotone* heuristic for it to work correctly. And the *h*-function given above turns out not to be monotone. An example is the edge B -> C. The requirement for monotonicity here says that

$$h(B) \leq \text{length}(B,C) + h(C)$$

This amounts to $8 \leq 5 + 1$ which is false, and thus *h* is not monotone, and therefore the result could be wrong.

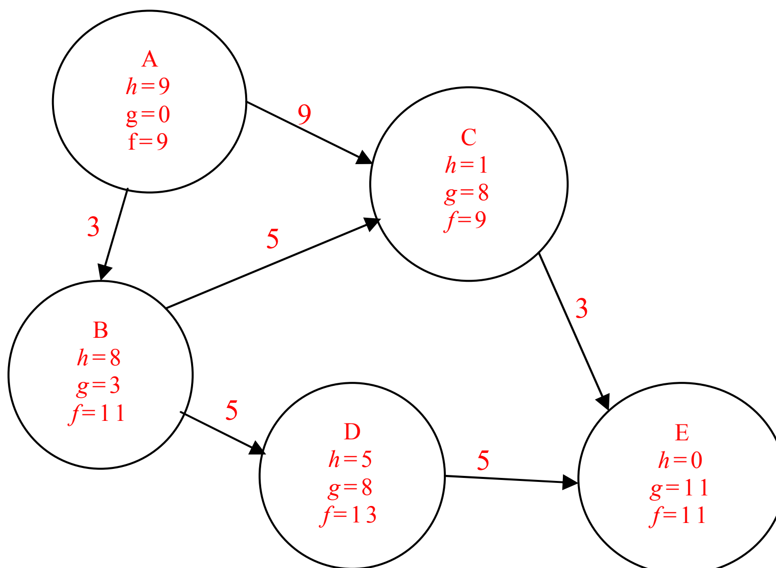
Question 3.c (6%)
There is another procedure described in the textbook that can handle the above case (with all values as they are in the figure). Explain which procedure that is, and show the steps this procedure will go through for the above case, in the same way as in 3.a, including the *f* and *g* values on a new copy of the figure above.

Answer 3.c
The solution anticipated by those producing the question was as follows: There is another algorithm, also referred to as an A*-algorithm in the textbook (and as an A-algorithm in the slides), that only requires that the h-values are less than the shortest path to the goal node (but not that it is monotone). One can easily verify that this is the case for each node in the graph given above, and thus this algorithm should give the correct answer.

The difference between this algorithm and the A*-SearchMH is that we should now move a node back into the queue if it gets a new (and better) g-value, even if it is already once taken out of queue. Thus the node C will be moved back into the queue when B is taken out, and we get the following sequence of events:

Start	A (0, 9)
A	B (3, 11) C (9, 10)
C	E (12, 12)
B	D (8, 13) C (8, 9) (Node C got a better *g*-value, and is returned to the queue)
C	The value of g-value E (in the queue) is updated to 11
E	Thus, the goal node is taken out of the queue, and this signals the end of the algorithm

Thus the values at termination of the algorithm is as below, and we can see that the correct distance is found in g(E).



One can also say that Dijkstra's algorithm is a correct answer. Then we would have to disregard then *h*-values all together, and we get the correct answer since all lengths are positive. However, one may then also say that we have *changed* the h-values to zero, which is not legal. But we think that also Dijkstra's algorithm should, more or less, be considered a correct answer, depending to some extent on the explanations given.

Assignment 4: Decidability and NP-completeness (*22 %*)

Question 4.a (11%)
Determine whether the following languages are decidable. Prove your answers.

1. L1 = { M | Turing machine M writes a $ after no more than100 steps of computation for every input }

2. L2 = { M | Turing machine M decides whether the first 100 characters of its input contain a $ }

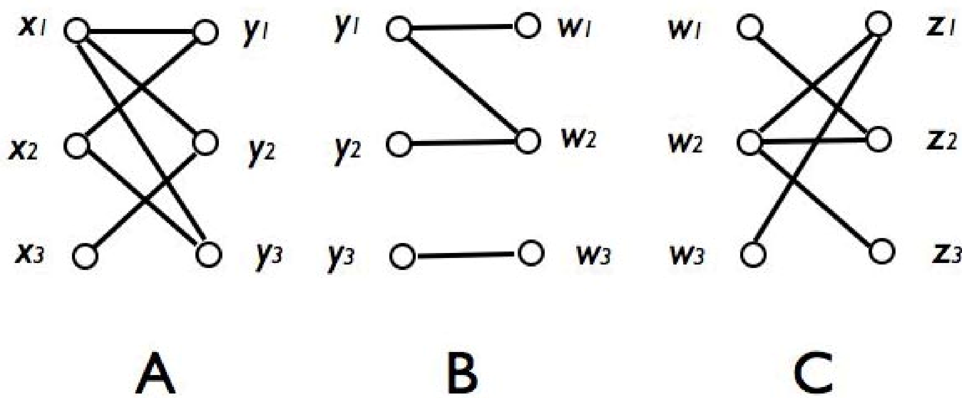
Answer 4.a.1
Decidable. Since in 100 a Turing machine can 'see' only 100 characters of its input, the decision procedure simulates M on input x for 100 steps (by using the Universal TM) in a loop where x ranges over all inputs of length 100 or smaller. If at any point M writes a $ the procedure halts scanning a 'YES', otherwise it halts scanning a 'NO'. (This procedure in fact runs in constant time, but this constant is astronomically large.)

Answer 4.a.2
Not decidable. The proof is by the following simple modification of the standard reduction from the Halting Problem (the students are expected to describe it in a bit more detail): Given an instance of the Halting Problem (M, x), the reduction algorithm R outputs the code of a machine M' and submits it as input to a presumed algorithm for L2. M' saves its input, then simulates the given M on the given input x (by using the Universal TM), and then (if M halts on x) restores its input and runs an algorithm that checks whether one of the first 100 characters of its input contain a $. The argument is by contradiction: Assuming that L2 exists, the described procedure decides the Halting Problem. Since the Halting Problem has been proven unsolvable, this assumption must be false.

Question 4.b (11%)
The four-dimensional matching (4DM) problem is a straight-forward generalization of three-dimensional matching: An instance of 4DM consists of four disjoint sets X, Y, W and Z of size *k,* and a set Q of quadruples Q = { (x, y, w, z) | x ∈ X, y ∈ Y, w ∈ W, z ∈ Z }. The associated question is whether there is a "perfect" matching in Q, i.e. whether there is a subset M ⊆ Q such that each of the elements of X, Y, W and Z is in exactly one quadruple in M.

1. Prove that 4DM is NP-complete. Hint: prove that 3DM is polynomial-time reducible to 4PM. Can you generalize this result to KDM, where K ≥ 3 ? Explain.

2. We change the problem above so that we still want a perfect 4D matching M, but we replace the set Q of quadruples by three bipartite graphs, e.g. like A, B, and C below. The requirement for M is then that each of its quadruples is a concatenation of edges in A, B and C (ex. x_1, y_2, w_2, z_3). What is the complexity of this problem? Explain.



Answer 4.b.1
Given a 3DM instance, the reduction creates a 4DM instance by adding a distinct fourth set Z to the existing three sets (Z has the same cardinality m as those three sets), and for each triple q in Q outputs n quadruples, by linking the last element of q to each of the m elements of Z. It is clear that this reduction procedure is polynomial-time (the time complexity is $O(n\ m) = O(n^2)$), and that the resulting instance of 4DM has a perfect matching if and only if the original 3DM instance has one. The generalization to KDM is straight forward.

Answer 4.b.2
TM4DM is solvable in polynomial time by computing the bipartite matching three times, for each of the three bipartite graphs, and by returning 'YES' if and only if all of them return a positive answer. A TM4DM matching, if it exists, is a simple concatenation of the three bipartite perfect matchings.

Assignment 5: Dynamic programming (22 %)

Question 5.a (6%)
We shall look at the following simplified version of editing a string S to a string T. You are now only allowed to do *one* type of edit-operation, which is to insert a symbol into S (and this may be done any number of times). Sometimes (and obviously if T is shorter than S), you cannot obtain T from S in this way. It is possible e.g. if S = "abc" and T ="aacbbac", but not if T = "aacbba". The length of S is m, and the length of T is n. For strings S and T where the above transformation is possible, the number of necessary insertions is obviously $n - m$. Our job is to decide whether this transformation is possible at all, for given S and T.

Describe how you can solve this problem with dynamic programming. Describe what sort of table you will use, and what general rule you will use for filling in the table.

Answer 5.a
A straight forward solution is to use the same scheme as for the more complex edit distance discussed in the textbook. A suitable boolean table B could then look like this:

i↓ j→	0 empty	1 a	2 a	3 c	4 b	5 b	6 a	7 c
0 empty								
1 a								
2 b								
3 c								

The general idea here is that if the string S[1: i] can legally be transformed to T[1: j], then the entry B[i, j] in the table should be **true**, else **false**. The general formula for B[i, j] can be describe as follows:

B[i, j] = if T[i, j-1] then **true** else if B[i-1, j-1] and S[i] == T[j] then **true** else **false**

The first part relies on the fact that if S[1:i] can be transformed to a prefix of T[1:j] (here T[1:j-1]), then it can also be transformed into T[1,j] by simply inserting the remaining symbols of T. Thus, we can also see that if B[i, j] is **true**, then the rest of row "i" (towards the right) will also be **true**. The last part says that if the string S[1:i-1] can be transformed T[1:j-1] and S[i] == T[j], then S[1:i] can also be transformed to T[1:j].

Question 5.b (6%)
Describe how you, if necessary, will initialize the table. Sketch a program that first does the initialization (if necessary) and then fills in the rest of the table.

Answer 5.b

i↓ j→	0 empty	1 a	2 a	3 c	4 b	5 b	6 a	7 c
0 empty	true	true	true	true	true	true	true	true
1 a	false							
2 b	false	false						
3 c	false	false	false					

We can initiate the table with **false** where S[1:i] is longer than T[1:j]. This is when i>j, as shown above. Also, the empty string S[1:0], can always be transformed to T[1:j] for any j, so the row with i=0 should all be true. Otherwise we can use the formula from 5.a. A sketch of a program could be:

```
// Initialization:
for j = 0 to n do { B[0, j] = true;}
for i = 1 to m do {
  for j = 0 to i-1 do {B[i, j] = false;}
}

// Fill in the rest:
for i = 1 to m do {
  for j = i to n do {
     if T[i, j-1] then { B[I, j] = true;}
  }else{
    if B[i-1, j-1] and S[i] == T[j] then { B[I, j] = true;} else { B[I, j] =false;}
  }
 }
```

Question 5.c (5%)
Is it possible to solve the problem using less space than for the straight-forward format of the table? Explain!

Answer 5.c
As we can fill in the table column by column (instead of row by row, as in the program above) we only need the previous column to produce the next (in addition to S and T). Thus the algorithm can

manage with space O(m).

Question 5.d (5%) (You may take this as the last one)
We make a twist on the problem from 5.a, and extend the allowed possible operations to any insertion of 1 or more occurrences of the same symbol consecutively into P to obtain T. That is, such an insertion now counts as one insertion.

Now, the number of insertions cannot be computed in advance. We here ask whether such a transformation from S to T is possible at all, and, if it is, we also want to know how many of these edit-operations are necessary. Sketch the main steps in a dynamic programming algorithm solving this problem.

Hint: You can try with a table T where each entry (at least) contains a number and a symbol.

Answer 5.d
Whether such a transformation is possible or not can be solved by the same method as described in 5.b. However, there turned out to be instances of this problem that are more difficult than the poser of the question had anticipated. As an example we can look at S = "ab" and T ="aaabbbab". Here, one would easily match "a" with the first "a" in "aaa", thus having to insert "aa", and the same for "b", where "bb" must be inserted. Finally then, we would have to insert "a" and "b" in separate operations. This gives 4 insertions. However, if we first insert "aaa" and then "bbb", we would get an exact match for the last two symbols "ab" in T, so that this altogether gives only two insertions.

We have not been able to find an algorithm (using dynamic programming or other methods) that are able to see that this type of late reorganizations will give a better result (but maybe somebody else can?). Very few students came as far as to this question, but those that had been able to say something significant about it got go a good grade for

[End]