UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Exam in:	**INF 4130/9135: *Algoritmer: Design og effektivitet***
Date of exam:	**16th December 2013**
Exam hours:	**09:00 – 13:00 (4 hours)**
Exam paper consists of:	**5 pages (plus appendices)**
Appendices:	**2 pages (two copies, use one as scratch paper)**
Permitted materials:	**All written and printed**

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

The first assignment below is to be answered using the attachment in the appendix. There are two copies of the attachment, one for INF 4130 and one for INF 9135. Tear out and deliver the copy with the appropriate course code together with the white copy of your answers (and you can use the other copy of the attachment to sketch solutions).

Read the text carefully, and good luck!

Assignment 1 *String Search with Boyer-Moore and Knuth-Morris-Pratt* (*11 %*)

Question 1.a (5 %)
We use the simplified Boyer-Moore algorithm (called Horspool on the slides) to search for the pattern P = `idefix` in the string T = `abcdefgfhifjk`. Obviously we will never get a full match, but your task is to show how the pattern is shifted when the algorithm is run. Use (or copy) the table in the appendix to give your answer as a series of shifts, like the first shift shown in the example below (note that the example uses a different pattern). In the Reason column just write the letter and shift value that *caused* the shift on that line.

String													Reason
a	**b**	**c**	**d**	**e**	**f**	**g**	**f**	**h**	**i**	**f**	**j**	**k**	
a	a	.	.	.	.								Initial
	a	a	.	.	.	.							Shift(?)=1 *

* You must of course indicate the correct shift value of an actual letter.

Answer 1.a

String													Reason
a	**b**	**c**	**d**	**e**	**f**	**g**	**f**	**h**	**i**	**f**	**j**	**k**	
i	d	e	f	i	x								initial
		i	d	e	f	i	x						Shift(f) = 2
				i	d	e	f	i	x				Shift(f) = 2
					i	d	e	f	i	x			Shift(i) = 1
							i	d	e	f	i	x	Shift(f) = 2

Question 1.b (6 %)

We now use the Knuth-Morris-Pratt algorithm to search for the pattern P = `ababac` in the string T = `ababbabaabbac`. (Harder to make funny words with this one…) It is still obvious that we never get a full match, and your task is again to show how the pattern is shifted when the algorithm is run. Use (or copy) the table in the appendix to give your answer as a series of shifts, like the first shift shown in the example below. In the Reason column indicate where the mismatch occurs, and what the overlapping prefixes and suffixes that *caused* the shift on that line are (if there is any overlap). Assume that $|T| = n$ and $|P| = m$, and that T is indexed $T[0], \ldots, T[n\text{-}1]$ and P is indexed $P[0], \ldots, P[m\text{-}1]$.

String													Reason
a	**b**	**a**	**b**	**b**	**a**	**b**	**a**	**a**	**b**	**b**	**a**	**c**	
a	a	a	.	.	.								Initial
	a	a	a	.	.	.							$T[i] \neq P[j], P[x..y] = P[z..w]$*

* Again, you must of course indicate the correct values of actual letters and prefix/suffixes.

Answer 1.b

String													Reason
a	**b**	**a**	**b**	**b**	**a**	**b**	**a**	**a**	**b**	**b**	**a**	**c**	
a	b	a	b	a	c								Initial
		a	b	a	b	a	c						T[5]≠P[5], P[2..3]=P[0..1]
				a	b	a	b	a	c				T[4]≠P[2], ---
					a	b	a	b	a	c			T[4]≠P[2], ---
							a	b	a	b	a	c	T[8]≠P[3], P[2]=P[0]

Note that the shift patterns are equal.

Assignment 2 *The A*-algorithm used on a new form of edit distance* (*28 %*)

In class we used dynamic programming to find the edit distance between strings when we were allowed to insert, delete and substitute letters, one at the time. We are now going to look at a slightly different type of edit operations, and use the A*-algorithm to find a new type of edit distance called the "substi-distance" (see below).

Let our alphabet be the letters A, B, C and D, and let our operations be four substitutions, as defined by the following four rules, all with cost 1. Note that we always substitute two *successive* letters with one or two new letters:

1: AB ⇒ BA
2: AD ⇒ BC
3: BC ⇒ D
4: CD ⇒ DC

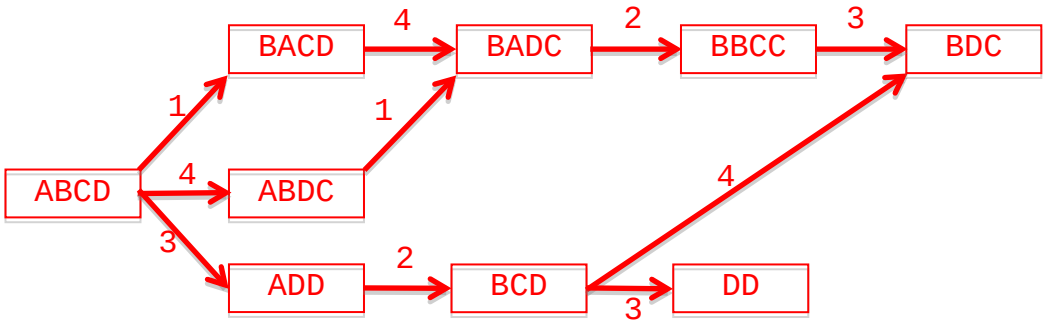
Example: Starting with the string ADB we can substitute AD with BC (rule 2) to get BCB, and then substitute BC with D (rule 3) to get DB, and now we can't substitute any more.

Definition: We define the *substi-distance* from a string *S* to another string *T* as the length of the shortest sequence of the above four operations that will transform *S* to *T*. If it is impossible to transform *S* to *T* with the above four operations, the *substi-distance* is defined as infinite. (If *T* is longer than *S*, the distance is obviously infinite.)

In the example above the substi-distance from ADB to BCB is 1, and from ADB to DB it is 2 (and zero from ADB to itself). Starting with ADB those are the only strings we can make, so the substi-distance to all other strings is infinite.

Question 2.a (5 %)
Draw the directed graph of all possible strings we can make with the above rules when starting from the string ABCD. There shall be one and only one node v_s for every distinct string s we can make, and there will be an edge from node v_s to node v_t if there is a rule we can apply once to string s to get string t. Label the edges with 1, 2, 3 or 4, to indicate which rule was used (see the table above).

Answer 2.a



Question 2.b (5 %)
Answer the following questions:

i) What is the substi-distance from ABCD to BBCC ?
ii) What is the substi-distance from ABCD to BDC ?
iii) What is the substi-distance from BACD to BDC ?
iv) What is the substi-distance from BACD to DD ?

Answer 2.b

i) 3
ii) 3
iii) 3
iv) ∞

Question 2.c (6 %)
We now want to find the substi-distance from a string S to a string T, by searching for T in a graph corresponding to the one in 2.a, with the A*-algorithm. We then need a heuristic function, and use the difference in length between the current string and the goal string T as our heuristic. For a node v_u with string u in the graph $h(v_u)$ = Abs($|u|$ - $|T|$) will be our heuristic value. (The difference in length between u and T; we take the absolute value since in a general setting, u may be both shorter and longer than T.)

Decide whether this heuristic is monotone! You need not give a formal proof, only a brief explanation of why or why not.

Answer 2.c

It is monotone. To verify this we must show: (1) That the heuristic value in the goal is zero, and this is obvious. (2) That $h(u) \leq c(uv) + h(v)$ for any pair of strings u and v, where one can produce v form u in one substitution step (so that $c(uv) = 1$). Because of the form of the four legal substitutions, we also know that $|v| = |u|$ or $|v| = |u| - 1$. Thus $|v| - |T|$ and $|u| - |T|$ will differ by at most one, and this is therefore also the case between

$h(v) = Abs(|v| - |T|)$ and $h(u) = Abs(|u| - |T|)$

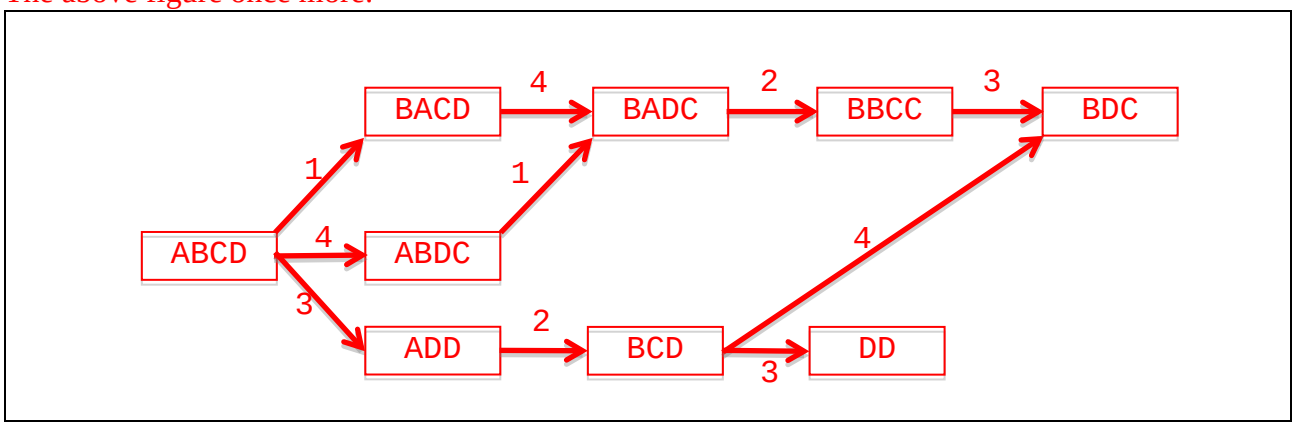
This directly implies that $h(u) - h(v) \leq 1 = c(uv)$, and thus that $h(u) \leq c(uv) + h(v)$

Question 2.d (6 %)

List the order (or one of the orders) in which the nodes are removed from the priority queue when we run the A*-algorithm to find the substi-distance from ABCD to DD with the heuristic of 2.b. You shall simply list the strings, starting with ABCD.

Answer 2.d

The above figure once more:



The order may become: ABCD,ADD,BCD,DD.

But also other orders are possible, depending on how you treat those with equal f=g+h in the priority queue. After: ABCD, ADD three nodes will have f-value 3: BACD (1+2), ABDC (1+2), and BCD (2+1). The above sequence corresponds to taking BCD first. Another order could be: ABCD,ADD,BACD,BCD,ABDC,DD.

Question 2.e (6 %)

Answer the following questions:

i) What will happen to the monotonicity of the above heuristic if we add substitution rules that transform a string into one which is longer?

ii) What will happen to the monotonicity of the above heuristic if we add substitution rules that transform a string into one which is at least two shorter?

Answer 2.e

i) It is *not* monotone. To show this we only need to give a counterexample, and for this we can add the substitution "D => DABCD". We then assume that u = D, v = DABCD, and T=DABC. Then h(u) = Abs(1 - 4) = 3 and h(v) = Abs(5 - 4) = 1, and c(uv) = 1. Thus, it is not always true that $h(u) \leq c(uv) + h(v)$.

ii) It is *not* monotone. For a counterexample here we add the substitution "ABCD => AD". We then assume that u = ABCD, v = AD, and T = D. Then h(u) = 3, h(v) = 1, and c(*uv*) = 1. Thus it is not always true that $h(u) \leq c(uv) + h(v)$.

Assignment 3 *Undecidability* (*12 %*)

Which of the following languages are undecidable? Sketch proofs for your answers.

Question 3.a (6 %)
$L_1 = \{M \mid$ Turing machine M does not halt for any input$\}$

Solution: Undecidable by standard reduction and by reversing the YES and NO on output.

M': Simulate *M* on input *x*

(Notice that nothing else is needed; *M*' halts for every input if *M* halts on input *x*; it halts for no input otherwise.)

Question 3.b (6 %)
$L_2 = \{M \mid$ Turing machine M decides $L_1\}$

Solution: Since L_1 is not decidable, L_2 is trivially decidable (by an algorithm that always answers NO).

Assignment 4 *Complexity* (*12 %*)

We have seen in class that Hamiltonicity – to determine whether the graph given as input has a simple cycle containing *all* vertices of the graph – is an NP-complete problem. (By a 'simple cycle' we mean a cycle where no repetition of vertices or edges is allowed, except for the starting and ending vertex.) What is the complexity of the following problems? Give complete proofs.

Question 4.a (6 %)
To determine whether the graph given as input has a simple cycle containing at least one half of its vertices.

Solution: NP-complete. The reduction is an easy reduction from Hamiltonicity, where n/2 isolated vertices are added to the original instance. The students should prove that the reduction is polynomial and 'proper' (mapping positive instances of Hamiltonicity into positive instances of the new problem and vice versa); these proofs are trivial, but the idea is to check that the students understand what a complete proof entails.

Question 4.b (6 %) (***Not straight forward!*** *You might want to look at this as the last one.*)
To determine whether the graph given as input has a simple cycle containing at most one half of its vertices.

Solution: This was an opportunity to grapple with an open problem, and earn partial credit for good or promising ideas.

Assignment 5 *Understanding concepts and ideas* (*20 %*)

Give short (no longer than three sentences) answers to the following questions:

Question 5.a (5 %)
When discussing complexity of algorithms, we model problems by formal languages. In what way is an ordinary problem (consisting of input-output pairs) represented as a formal language?

By representing all problems as formal languages we represent them in a uniform way—as elements of a single set, and are then able to organize them into classes. An ordinary problem is represented as a formal language when we represent it by the corresponding decision problem, and put all the strings representing positive instances into a set—i.e. a formal language.

Question 5.b (5 %)
What is Church's, or the Church-Turing, thesis? Why is this a 'thesis' and not a theorem?

Solution: The Church-Turing thesis states that a function is computable (a problem is solvable) if and only if it is computable (solvable) by a Turing machine. This is a 'thesis' because it combines real-world concepts ('computable'—by any algorithm or machine) with theoretical /abstract / mathematical ones (Turing machine). Only statements about mathematical concepts are provable.

Question 5.c (5 %)
Name several strategies that are available for coping with intractability ('solving', in some specified, efficient way, NP-hard and harder problems).

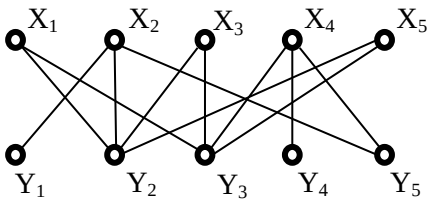
Solution: We talked about reducing the range of instances to the ones that arise in practice (limited numbers, graphs with limited max. degree etc.); approximation, algorithms that work well on average, probabilistic algorithms and heuristics (simulated annealing, genetic algorithms, neural nets...).

Question 5.d (5 %)
Can we solve NP-complete problems efficiently by using parallel computers? Explain!

Solution: Only with exponential hardware (which is unrealistic), unless P=NP. A single machine can simulate a parallel machine with polynomial hardware in polynomial time.

Assignment 6 *Matchings* (*17 %*)

Question 6.a (6 %)
We look at the following bipartite graph:



Find the largest number of edges a matching can have in this graph, and explain why your answer is correct. You don't have to show how you found the elements that go into your explanation.

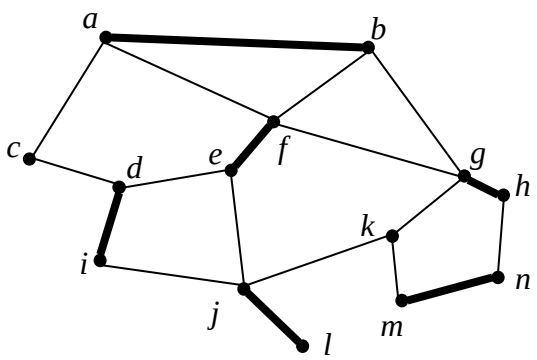
Answer 6.a

The largest possible matching has four edges. A simple proof for this could be (1) to display a matching with four edges, and (2) to show that it cannot have a perfect matching. The former could be the edges: X1-Y2, X2-Y1, X3-Y3, X4-Y4. The latter can be shown through Hall's Theorerm. We can observe that if S = {X1, X3, X5}, then *Gamma(S)* = {Y2, Y3}.
As | S | >= | *Gamma(S)* |, there cannot be a perfect matching, and thus no matching of size five.

One could also observe that there is a set of four nodes that cover all edges. Such a subset could be: {X2, X4, Y2, Y3}, and it is a theorem that no matching can have more edges than the number of nodes in an edge-covering node-set. This theorem was not directly part of the curriculum, but it was treated in some of the weekly assignments, so this type of answer is also perfectly OK.

Question 6.b (6 %)

We have the following graph, with a given matching *M* (marked with thick edges).



We will look for a larger matching, and use the Extended Hungarian Algorithm described on the slides. We start by making the unmatched node *k* the root, and start building an alternating tree by looking at the edges in the following order:

k-g, k-j, k-m, h-n, g-f, e-d, e-j

For each edge you look at you should complete the step according to the algorithm, and sometimes this will include that a set of nodes is merged (or collapsed) into one node. When a node occurs as an end node of an edge in the list above and this node is already merged with others, we shall mean the whole merged node it has become a part of.

The question is: How many mergings will occur during the steps given above, and what node-sets are merged?

Answer 6.b

Step by step, the following will happen:

Initially: The node *k* will be painted red, and all other nodes are colorless
k-g: *k-g* and *g-h* will be included in the tree, *g* blue and *h* red.
k-j: *k-j* and *j-l* will be included in the tree, *j* blue and *l* red
k-m: *k-m* and *m-n* will be included in the tree, *m* blue and *n* red
h-n: This is an edge between two already red nodes, and this will lead to merging of the odd cycle *k, g, h, n, m* to one red node, Usually one will then also repaint *g* and *m* to red nodes in an illustration, but this is not done below. The important fact is that they from now on together

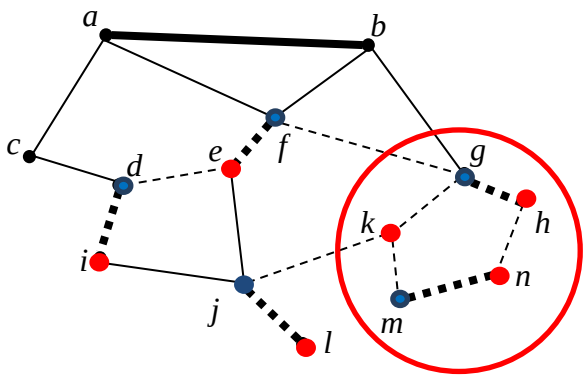
constitute one red node.

g-f: Here *g* represents the newly merged red node "*kghnm*", and the edges *kghnm-f* and *f-e* is included in the tree, *f* blue and *e* red

e-d: *e-d* and *d-i* will be included in the tree, *d* blue and *i* red

e-j: This is an edge between a red and a blue node, and will cause no action

Thus, the tree will look as shown below, and only *one* merging has occurred, of the nodes *k, g, h, n, m* . The tree consists of the dashed edges.

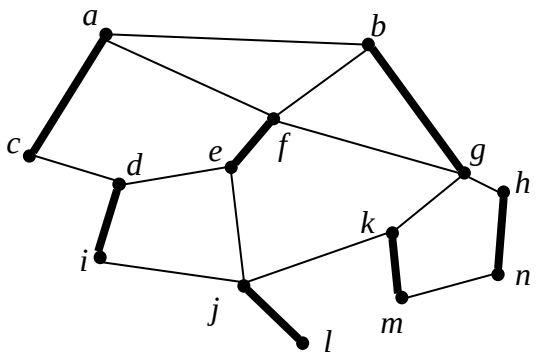


Question 6.c (5 %)

Is there a larger matching than *M* in the graph? If so, draw the graph with the larger matching. (Not asked for in the exam: How can you find a larger matching by further extending the tree above according to the aolgorithm).

Answer 6.c

It is an OK answer if a larger matching is simply given as follows:



One could find this matching by continuing the search performed in 6.b. Next step could be to look at the edge *kghnm-b*, which also adds *b-a* to the tree. Finally, we could look at *a-c*, and since *c* is uncolored, we have an augmenting path back to *k*. However, to find this path you have to "open up" the merged odd cycle, and traverse it in the correct direction, which is to start with a matched edge when you enter it on the way back to the root. That is: We start with *g-h*, and proceed back to *k* through *n, m, k*. Thus, the augmenting path from *c* to *k* is: *c, a, b, g, h, n, m, k*, and by "using" this, we get the matching above.

[END]