Midterm Exam

CSC 173

23 October 2001

Directions

This exam has 8 questions, several of which have subparts. Each question indicates its point value. The total is 100 points. Questions 5(b) and 6(c) are optional; they are not part of the 100 points, but will count for extra credit at the end of the semester.

Please show your work here on the exam, in the space given. You shouldn't need to write on the backs or in the margins; if your answer won't fit in the space given then you're trying to write too much. Put your name on every page. Scrap paper is available if you need it, but the proctor will collect only the exams.

I have tried to make the questions as clear and self-explanatory as possible. If you do not understand what a question is asking, make some reasonable assumption and *write that assumption down* next to your answer. The proctor has been instructed not to try to answer any questions during the exam.

You will have the entire class period to work. Good luck!

1. (9 points). Name and briefly describe (one sentence each) three different data structures that could be used to implement (in three different ways) a *relation* abstraction.

   array – *Each tuple goes in a separate, arbitrary slot of the array.*

   linked list – *Tuples are dynamically allocated and linked to one another with pointers or references.*

   hash table – *Tuples occupy buckets of a lookup table indexed by some primary key.*

   balanced tree – *Tuples are dynamically allocated and organized into a binary (or n-array) hierarchy.*

   characteristic array – *Tuples go into slots of the array indicated by values of some primary key, which must form a dense, ordered, finite set.*

2. Consider the following relation R:

A	B	C	D
red	10	large	x
blue	12	large	y
green	6	small	z
red	9	small	w
yellow	20	small	z

(a) (7 points). Judging from the tuples shown, which attributes or sets of attributes might be keys for R? How can you tell?

B, AC, AD are the minimal sets of columns that have unique values for all tuples. They could be keys, or we might need additional columns, to make sure that we will never see any tuples with identical "key" values. Without semantic information (what do the columns mean*), all we can really be sure of is that A, C, D, and CD are* not *keys.*

(b) (5 points). Give the result of the operation $\sigma_{D=z \vee B<10}(R)$.

A	*B*	*C*	*D*
green	*6*	*small*	*z*
red	*9*	*small*	*w*
yellow	*20*	*small*	*z*

(c) (5 points). Give the result of the operation $\pi_{C,D}(R)$.

C	*D*
large	*x*
large	*y*
small	*z*
small	*w*

3. (9 points). What does the following program print? Explain.

```
#include <stdio.h>
int main ()
{
    char *s = "string 1";
    char *t = "string 2";

    if (s == t) printf("yes\n"); else printf("no\n");
    if (*s == *t) printf("yes\n"); else printf("no\n");
}
```

It prints

```
no
yes
```

Variables `s` *and* `t` *are pointers to strings (arrays of characters). The first* `if` *statement checks to see whether* `s` *and* `t` *have the same value* as pointers*—whether they point to the same location in memory. They do not. The second* `if` *statement checks to see whether the characters pointed at by* `s` *and* `t`*—the first characters of the two strings—are the same. They are (they're both* `s`*'s). If we really wanted to compare the content of the strings, we'd need to write a loop or use the strcmp library function:*

```
if (!strcmp(s, t)) printf("yes\n"); else printf("no\n");
```

4. We discussed three possible implementations of the join operation: a *nested loop join*, which iterates over all pairs of tuples, a *sort join*, which sorts all tuples (of both relations) by the join attribute prior to finding matching pairs, and an *index join*, which iterates over the tuples of one relation and performs lookup operations on the other relation.

   Suppose that we are performing a join on relations R_1 and R_2 to produce relation R_3. Suppose further that relation R_1 has N_1 tuples, relation R_2 has N_2 tuples, and relation R_3 has N_3 tuples.

   (a) (6 points). Give the asymptotic complexity (Big-O running time) of the three join implementations, in terms of N_1, N_2, and N_3.

   nested loop: $O(N_1 \times N_2)$.

   sort: $O(N_1 \log N_1 + N_2 \log N_2 + N_3)$ *or* $O((N_1 + N_2) \log(N_1 + N_2) + N_3)$*, depending on details of the implementation.*

   index: $O(N_1 + N_3)$ *or* $O(N_2 + N_3)$*, depending on whether you use the* R_2 *index or the* R_1 *index, respectively, and assuming a hash table index. With a balanced tree index, it's* $O(N_1 \log N_2 + N_3)$ *or* $O(N_2 \log N_1 + N_3)$.

   (b) (5 points). Can you think of a situation in which a sort join would be preferred over an index join?

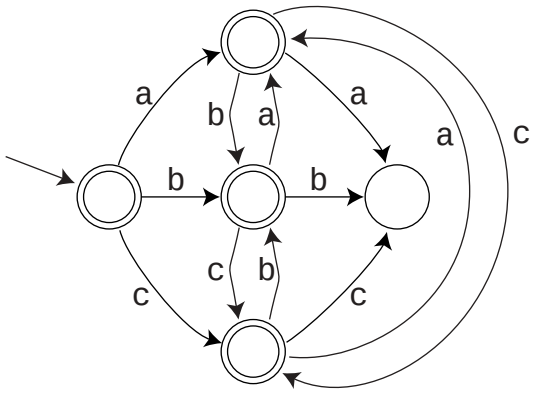
   If you're joining on an attribute for which you don't have an index for either relation, and don't want to create one. Alternatively, if R_1 *and* R_2 *are known to be sorted on the join attribute already, or* R_3 *needs to be. This latter case (known to be sorted already) might hold if your indices were balanced trees indexed by the join attribute: exploiting the already-sorted nature of the tree could be faster than repeatedly traversing it down from the root.*

   (c) (5 points). Can you think of a situation in which a nested loop join would be preferred over a sort join?

   If one of the relations is much *smaller than the other, so* $N_1 < \log N_2$ *or* $N_2 < \log N_1$*, and thus* $N_1 \times N_2 < N_1 \log N_1 + N_2 \log N_2$*. Alternatively, if most of the tuples in both relations have the same value for the attribute on which you're performing the join, so* $N_3 \approx N_1 \times N_2$*. Note, though, that you're unlikely to be able to predict this in advance. NB: I gave partial credit for "if both relations are really small". Here the nested loop is faster, but only by a small constant factor.*

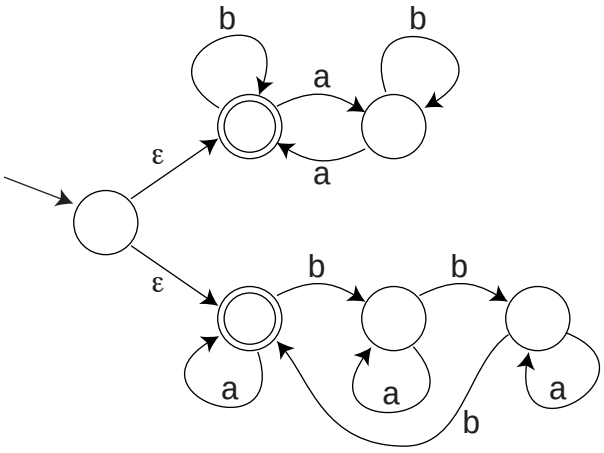
5. Consider the language consisting of all strings of `a`'s, `b`'s, and `c`'s, with no two identical consecutive letters.

   (a) (7 points). Give a DFA that accepts this language.



   (b) (EXTRA CREDIT, up to 8 points). Give a regular expression that describes this language. Hint: this is hard. Start by creating a regular expression R that describes strings of alternating `a`'s and `b`'s. Then create a regular expression that describes strings of alternating R's and `c`'s.
   Let $R = b\ (ab)^*\ (a \mid \epsilon) \mid a\ (ba)^*\ (b \mid \epsilon)$ *Our answer is then* $(R \mid \epsilon)\ (cR)^*\ (c \mid \epsilon) =$ $((b\ (ab)^*\ (a \mid \epsilon) \mid a\ (ba)^*\ (b \mid \epsilon)) \mid \epsilon)\ (c\ (b\ (ab)^*\ (a \mid \epsilon) \mid a\ (ba)^*\ (b \mid \epsilon)))^*\ (c \mid \epsilon)$. *Note that R must generate* non-empty *strings, but our answer must include the empty string.*

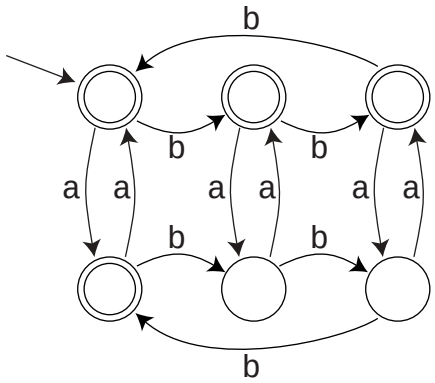
6. Consider the following NFA:



   (a) (7 points). Describe in English the language accepted by this NFA. Note: you will not receive full credit just for describing how the NFA works: "zero or more b's followed by ... or zero or more a's followed by ..." isn't acceptable.
   All strings of a's and b's in which either the number of a's is evenly divisible by 2 or the number of b's is evenly divisible by 3.

(b) (7 points). Give a regular expression that describes the same language.
$b^*(ab^*a)^*b^* \mid a^*(ba^*ba^*b)^*a^*$

(c) (EXTRA CREDIT, up to 8 points). Give a DFA that accepts the same language.



7. (8 points). When performing the subset construction to turn an NFA into an equivalent DFA, we need to adopt the convention that the NFA accepts only if it can end up in an accepting state *after consuming its entire input*. Why? What would be wrong with a convention that says you accept if you can "get stuck" (no outgoing transition) in an accepting state, even if there is some input remaining?

Suppose there are two NFA states A and B that can be reached on the same input string (prefix) w. Suppose A is accepting and has no outgoing edges, but B is non-accepting and has a transition on symbol `a` *to a non-accepting state C. A subset state containing A and B will be accepting (because A is), and will have an outgoing edge on* `a`*. So the NFA will accept when given input w*`a`*, but the DFA (subset machine) will not.*

8. Short answer.

(a) (5 points). A database representing academic information at the University might have relations with the following schemes:

student_id, student_name, student_address
student_id, course_number, semester, grade
course_number, course_name, department, credits

In principle we could store the same information in a relation whose scheme consists of the nine distinct attributes from the list above. Why don't we do this?

Because it would introduce a huge amount of redundant information, wasting space and introducing the need to guard against inconsistencies in copies.

(b) (5 points). The search (find) mechanism in many text editors allows the user to describe the target of the search with a regular expression, not just a specific string. The editor converts the regular expression to an NFA in order to drive the search. In many cases, the editor then emulates the NFA directly (keeping track of all possible current states), rather than converting the NFA to a DFA. Why do you suppose it does this? Why not create the DFA?

Because the conversion to a DFA takes time and potentially a lot of space. Since we're only going to use the machine once the cost of NFA emulation may be less than the cost of the conversion. The scanner in a compiler, by contrast, is run many, many times, and needs to be really fast; it definitely warrants conversion to deterministic form.

(c) (5 points). The Unix `make` utility is designed to manage collections of files that depend on one another. Programmers who build large systems typically arrange for `make` to call the compiler (e.g. `gcc`), rather than calling it themselves. Why? Why not just create a one-line shell command or alias that invokes `gcc` with the right arguments, and passes it all the source files?

Because typically when you make a change to a source file in a large system, only a small subset of the files really need to be recompiled. By using `make` *you can arrange to recompile all and only those files, avoiding the time required to compile the others.*

(d) (5 points). Summarize informally the general rule that determines whether it is profitable to push a project operation inside a join.

It's profitable if projecting before the join would significantly reduce the amount of data (columns and maybe rows) that must be processed by the join, but not *if it forces us to look at tuples that we would otherwise have been able to avoid inspecting, by performing an index join.*