15-122: Principles of Imperative Computation, Fall 2010
Assignment 7: ROBDDs and a Look at C

William Lovas (wlovas@cs) Tom Cortina (tcortina@cs)

Out: Wednesday, November 10, 2010
Due (Written): Tuesday, November 16, 2010 (before lecture)
Due (Programming): Wednesday, November 17, 2010 (11:59 pm)

1 Written: BDDs and C (25 points)

The written portion of this week's homework will give you some practice reasoning about ROBDDs and programming in C. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

1.1 Binary Decision Trees and ROBDDs

Exercise 1 (8 pts)**.** Consider the implication operation $a \Rightarrow b$, where $a \Rightarrow b$ is defined as $\neg a \vee b$.

(a) Draw an ROBDD for $x_1 \Rightarrow x_2$ and an ROBDD for $x_2 \Rightarrow x_1$. For both ROBDDs, use the ordering x_1, x_2 (that is, x_1 should be at the root with x_2 below it). [2 pts]

(b) Using the ROBDDs you derived in the previous part, build an ROBDD for the double-implication operation $x_1 \Leftrightarrow x_2$, where $x_1 \Leftrightarrow x_2$ is defined as $(x_1 \Rightarrow x_2) \wedge (x_2 \Rightarrow x_1)$. Show where phantom nodes are needed to perform the conjunction. [3 pts]

(c) Using the ROBBD you derived in the previous part, build an ROBDD for the boolean expression $(x_1 \Leftrightarrow x_2) \vee x_3$. Show where phantom nodes are needed to perform the disjunction. Use the ordering x_1, x_2, x_3 for your ROBDD. [3 pts]

Exercise 2 (9 pts)**.** In the *graph coloring* problem, we have a graph with n nodes and we want to color the nodes of the graph with k colors so that no two adjacent nodes have the same color. For example, map makers are interested in this problem since

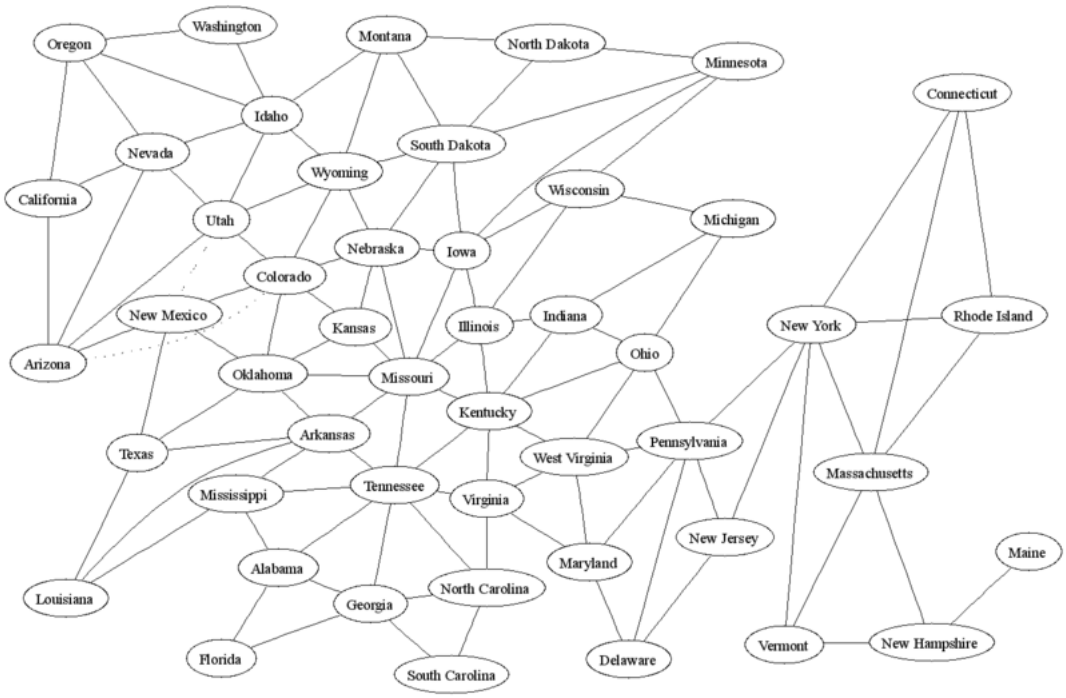

Figure 1: The contiguous United States as a graph. (From Wikipedia; ignore the dotted lines in the graph: these states touch at a point only.)

a map can be modeled as a graph, with nodes representing individual regions and edges connecting regions that share a common border. Finding a solution helps map makers color the map so no two adjacent regions share the same color, making it easier to distinguish regions (see Figure 1).

(a) Describe a way to encode a graph coloring problem with 4 colors into a binary decision tree. Define what your variables represent in the tree. [3 pts]

(b) For the contiguous 48 states of the United States, exactly how big would the binary decision tree be? That is, how many nodes will the tree have, not counting the terminal 0 and 1 nodes? Explain your answer. Remember: this is a binary decision tree; the reduction to an ROBDD has not been done yet. [2 pts]

(c) We can reduce the size of our decision structure by eliminating the testing of some variables entirely. That is, an entire portion of the binary decision tree can be removed and replaced with a zero node. Give an example using the US graph when this might occur. Explain clearly. [2 pts]

(d) We can reduce the size of our decision structure even more by merging equivalent subtrees. Give an example using the US graph when this might occur. Explain clearly. [2 pts]

1.2 C Programming

Exercise 3 (8 pts)**.** For each of the following problems, state what is wrong with the code and show how to correct it, if possible. Do not just try to compile it and write down the error message. (Some of these will compile without error!) Read the code and explain what is being done wrong, conceptually.

(a)

```
#include <stdio.h>

int main()
{
  int *p = 7;
  printf("%d\n", *p);
  return 0;
}
```

(b)

```
#include <stdio.h>

#define CUBE(X) (X * X * X)

int main()
{
  int c = CUBE(2+3);
  printf("2+3 cubed = %d\n", c);
  return 0;
}
```

(c)

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
  int a[] = {1,1,2,3,5,8,13,21,34,55};
  printf("%d\n", *(a+2)+*(a+5));
  free(a);
  return 0;
}
```

(d)

```
#include <stdio.h>

int main()
{
  int a[50];
  int *i;

  for (i = &a[0]; i < &a[51]; i++) {
    *i = 0;
  }
  return 0;
}
```

(e) The standard string library function `strncpy(dest, src, n)` copies the specified number of characters `n` from the source string `src` to the destination string `dest`.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
  char *letter_data = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  char a[16];
  strncpy(a, letter_data, sizeof(a));
  printf("The first sixteen letters are: %s\n", a);

  return 0;
}
```

(f) This code fragment shows a C function that is called from another function. It is supposed to return the result only if no overflow occurs.

```
#include <assert.h>

int oadd(int x, int y) {
   int result = x + y;
   if (x > 0 && y > 0) assert(result > 0);
   if (x < 0 && y < 0) assert(result < 0);
   return result;
}
```

(g) This code fragment shows a C function that is called from another function.

```
#include "xalloc.h"
#include <stdlib.h>

#define TABLESIZE 100
int *table = NULL;

int insert_in_table(int pos, int value) {
  if (table == NULL)
    table = (int *)xcalloc(TABLESIZE, sizeof(int));
  if (pos >= TABLESIZE)
    return -1;
  table[pos] = value;
  return 0;
}
```

(h) This code fragment shows a C function that is used as part of an implementation for stacks. Assume that `is_stack` returns true.

```
#include "contracts.h"
#include <stdlib.h>

int stack_size(stack S) {
   REQUIRES(is_stack(S));
   list L = malloc(sizeof(struct list));
   int size = 0;
   for (L = S->top; L != NULL; L = L->next)
      size++;
   return size;
}
```

2 Programming: Memory Management in C (25 points)

For the programming portion of this week's homework, you'll get some practice at managing memory manually by implementing some generic data structures in C.

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Starter code. Download the file `hw7-starter.zip` from the course website.

Compiling and running. Compile your code using GCC. The following set of options will catch many common mistakes at compile-time:

```
gcc -Wall -Wextra -std=c99 -pedantic -Werror <files...>
```

For details on how we will compile your code, see the file `COMPILING.txt` included in the starter code. To enable assertion checking, ensure that `DEBUG` is defined using the `-DDEBUG` option. **Warning:** *You will lose credit if your code does not compile.*

To detect any invalid memory accesses or memory leaks, you can run your compiled binary through Valgrind:

```
valgrind ./a.out
```

Use the `-v` option for verbose output, or the `--leak-check=full` option to generate a more complete report of possible memory leaks. **Warning:** *You will lose credit if your code has invalid memory accesses or memory leaks.*

Submitting. Once you've completed some files, you can submit them by running

```
handin -a hw7 <file1> ... <fileN>
```

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Use the macros in `contracts.h` to write appropriate annotations for your code in a style similar to what we've been doing in C_0. Remember that writing these annotations *before* writing the code will help you understand the problem more clearly and save debugging time later. **Annotations are part of your score for the programming problems; you will not receive full credit if they are weak or missing.**

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

2.1 Queues

Recall the C_0 implementation of queues, included in the starter code as the files `queues.h0` and `queues.c0`. A queue is represented as a pair of pointers to list cells: one to the front of the queue and one to the back.

```
typedef struct list* list;
struct list {
  elem data;
  list next;
};

struct queue {
  list front;
  list back;
};
```

Remember that the `back` list cell is a dummy cell containing no useful data; it is used to store the new element when one is enqueued.

Task 1 (10 pts)**.** Translate the C_0 implementation of queues into C. Submit a header file, `queues.h`, and an implementation file, `queues.c`. Use the special type `void *` to make your implementation generic over the type of elements. Remember to free any memory you allocate, and be sure to include a function

```
void queue_free(queue Q, void (*elem_free)(void *))
```

that frees all the memory associated with a queue, freeing the elements with the function pointer parameter `elem_free` if it is non-`NULL`.

Don't forget to update the `//@`-annotations using the macros in `contracts.h`, and be sure to implement some test code so you can run your implementation through `valgrind`!

2.2 Hashtables

Thursday's lecture showed a C implementation of fixed-size hash tables. As you learned in Homework 6, the performance of fixed-size hash tables for large programming tasks is highly dependent upon choosing a correct table size for the problem at hand. We can mitigate this dependence by extending our fixed-size implementation to an *adaptive* one.

The key idea behind adaptive hash tables is the same as that behind unbounded arrays. When the array backing an unbounded array becomes full, we double its size and copy over all of the elements. Similarly, when an adaptive hash table's load factor becomes too large, we double its size and reinsert all of the elements, rehashing their keys along the way to determine their position in the new table.

To implement adaptivity concretely, we rename the `table_insert` function to `table_really_insert` and we create a new `table_insert` function that optionally resizes before calling `table_really_insert` if the current load factor of the table is too large.

```
ht_elem table_insert(table H, ht_elem e)
//@requires is_table(H);
//@ensures is_table(H);
//@ensures table_search(H, (*H->elem_key)(e)) == e;
{
  REQUIRES(is_table(H));
  if (H->num_elems / H->size == RESIZE_LOADFACTOR)
    table_double_size(H);
  ht_elem result = table_really_insert(H, e);
  ENSURES(is_table(H));
  ENSURES(table_search(H, (*H->elem_key)(e)) == e);
  return result;
}
```

The threshold load factor is given by a macro `RESIZE_LOADFACTOR`. For example, if we want to resize when chains have a length of about two, then we would write:

```
#define RESIZE_LOADFACTOR 2
```

The heart of adaptivity is the `table_double_size` function, which allocates a new backing array twice as large as the hash table's old backing array, updates the hash table's fields appropriately, and reinserts all of the elements in the old array into the new one. Note that the elements cannot simply be copied directly since the change to the modulus m will almost certainly change the hash value of many of the keys currently stored.

Task 2 (15 pts)**.** Complete the C implementation of adaptive hashtables by implementing the resizing function `table_double_size` in the file `hashtable-double.c`. (You need not submit the original files, just your `hashtable-double.c`.[1]) Remember to free any memory that's no longer referenced after resizing.

Don't forget to include annotations using the macros in `contracts.h`, and be sure to test your code using `valgrind`!

[1]The split between `hashtable.c` and `hashtable-double.c` is achieved through a slightly idiosyncratic use of `#include`. Although convenient for the purposes of submission and grading, this use should probably not be emulated.