

CSE 150: Problem Set #4

December 13, 2007

Problem 1

Write two recursive functions that operate on linked lists:

- `insert(L,x)`. If `L` is sorted, then `insert(L,x)` returns a sorted linked list containing all the elements of `L`, and `x`. For example, `insert([1,3,4], 2) = [1,2,3,4]` and `insert([1,2,3], 1) = [1,1,2,3]`.
- `sort(L)` returns a sorted version of `L`. For example, `sort([2,1,3,0]) = [0,1,2,3]` and `sort([])=[]`.

Recall that the basic functions you can use are `head()`, `tail()`, `cons()`, and `isempty()`.

What are the running times of your algorithms?

Solution

```
procedure insert(L,x)
  if isempty(L)
    return cons(x, [])
  else
    if x ≤ head(L)
      return cons(x, L)
    else
      return cons(head(L), insert(tail(L), x))
```

This algorithm will call itself at most n times before it reaches the base case, and each time it only does one comparison and one call, so we have the recurrence $T(n) = T(n - 1) + 1$ for the worst case, which is $O(n)$.

```
procedure sort(L)
  if isempty(L)
    return L
  else
    return insert(sort(tail(L)), head(L))
```

This algorithm will also call itself at most n times before it reaches the base case, but each time, it calls `insert()`, which we know to be $O(n)$. Therefore, this algorithm takes $n \cdot O(n) = O(n^2)$ time.

Grading

- 10 points for each algorithm:
 - 4 for correctness
 - 4 for the correct running time
 - 2 for the analysis

Problem 2

Let T be a full binary tree (i.e. a binary tree in which each node has 0 or 2 children) with n leaves. How many internal nodes are there? Prove your answer.

Solution

Theorem. *A full tree with n leaves has $n - 1$ internal nodes.*

Proof. We prove by induction on n .

Base case ($n = 1$):

If T has 1 leaf, it must have 0 internal nodes (if it has one internal node, that would only have 1 child).

Inductive step:

Assume T is a full binary tree with k leaves and $k - 1$ internal nodes. If we add a leaf ν to T , we must add it beneath one of the leaves that is already there (because there are no leaves without a sibling). However, to keep T full, we must add a second node next to ν (or move the old leaf down), which will be a leaf, forcing the old leaf to become an internal node. In doing so, we have removed 1 leaf, created 2 more, and added 1 internal node. Therefore, $\text{insert}(T, \nu)$ has $k + 1$ leaves and k internal nodes. This proves our inductive hypothesis.

Therefore, any tree with n leaves must have $n - 1$ internal nodes. □

Grading

- 6 points for getting $n - 1$
- 14 points for the proof:
 - 4 points for the base case
 - 8 points for the inductive step
 - 2 points for style

Problem 3

How many different undirected graphs are there on n nodes? Directed graphs?

Solution

To define an undirected edge (assuming (a, a) is not a legitimate edge), we must choose two nodes, so there are $\binom{n}{2}$ possible edges. A graph is a subset of these possible edges, and there are $2^{|E|}$ subsets of E (the set of edges), so we have $2^{\binom{n}{2}}$ possible graphs on n nodes. If we include (a, a) , this gives us n more edges, so we have $2^{\binom{n}{2} + n}$ graphs.

To define a directed edge, we must choose an origin node and a destination node (here we will assume (a, a) is allowed), so we have n choices for the origin and n choices for the destination, which gives us n^2 possible edges, and therefore $2^{n^2} = 4^n$ graphs. If we do not allow (a, a) , we have instead $n - 1$ choices for the destination, so we have $2^{n(n-1)}$ graphs.

Grading

- 10 points for the undirected version, 10 points for the directed version, both graded as follows:
 - 2 points for including/excluding (a, a)
 - 8 points for the proof (divided this way if you attempted the type of proof I gave, divided otherwise if you did something else):
 - * 6 points for counting the number of possible edges
 - * 2 points for getting $2^{|E|}$

Problem 4

Examine the Bridges of Königsberg problem in Figure 1, where white boxes are “islands” and grey boxes are “bridges”. Is it possible to walk around Königsberg and walk over each bridge exactly once? Why or why not?

An Eulerian path in a graph is one which crosses every edge exactly once. An Eulerian cycle is an Eulerian path which starts and ends on the same vertex. Is there an Eulerian path on the Königsberg graph (Figure 1)? Why not? **Prove that a connected graph G contains an Eulerian cycle if and only if every vertex of G has even degree.**

Solution

For the north, east, and south islands, as each one has only 3 bridges leading off, if you enter and leave one of them, when you return, you are stuck (because you have crossed all the bridges connecting to that island). Since we can think of the beginning and the end of the path as places where it's okay to get stuck, we can only get stuck on two islands and still be okay. Since this leaves one more that we cannot get stuck at, we can only enter and leave it (covering two bridges), which means there will be one bridge we missed. Therefore, we can not cross every bridge in Königsberg exactly once in a connected path.

If there were an Eulerian path on the Königsberg graph, this would give us a path on which to walk that would visit each bridge exactly once. Since we just proved this is impossible, there cannot be an Eulerian path on the Königsberg graph.

If an Eulerian cycle exists on G , we know that every time we reach a vertex by some edge, we must leave it by another edge we have not crossed before. Therefore, every time we visit a vertex, we can “mark off” exactly two edges. As such, every vertex we visit in the cycle must have even degree. Since we can start and end the cycle at any vertex in the cycle, we can walk the cycle once starting at vertex v to prove that all other vertices have even degree, and then we can start the same cycle at another vertex to prove that v has even degree. Alternatively, we know that we must leave v at the beginning and return to v at the end, which gives us two marked edges, and we know that every other time we visit v , we can mark off two edges, so this gives us another proof that v has even degree.

If a graph G has even degree, we can start walking from some vertex v . If we mark edges as we go, every time we reach a vertex, since it has even degree, there must be another unmarked edge to leave by, until we get back to v , which, because we marked our exit edge at the beginning, will eventually leave us trapped at v . Note that we can never get trapped anywhere else. Now, if we have missed any edges in this walk, since G is connected, we know that we must have passed a vertex with some of those edges on it at some point during the walk. Furthermore, that vertex has even degree, so we can eliminate the edges we walked over originally, and start a new Eulerian cycle at that vertex, covering just the edges we missed originally. We can then connect the beginning of that cycle to the point in our original cycle when we got to that vertex, and the end of that cycle to the point when we originally left that vertex. In this way, we form a cycle that visits all of the edges in both cycles combined exactly once. Since there are a finite number of edges in a graph, and we can continue this splicing algorithm indefinitely, we must be able to cover all the edges in G with an Eulerian cycle.

Food for thought: What can you say about Eulerian paths/cycles on directed graphs?

Grading

- 4 points for saying why the original Bridges of Königsberg problem has no path
- 4 points for saying why there is no Eulerian path through Königsberg
- 6 points for one implication in the equivalence proof
- 6 points for the other implication in the equivalence proof

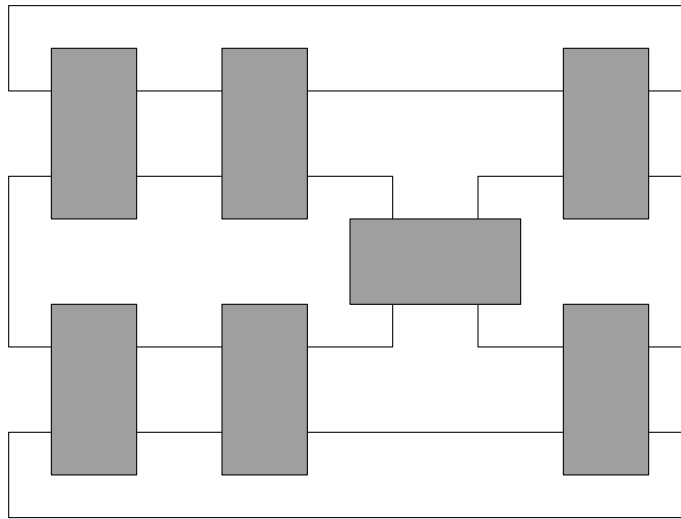


Figure 1: The Bridges of Königsberg

Problem 5

Prove that the following algorithm outputs some permutations more often than others:

```

procedure genPerm(n)
  A := [ 1, 2, 3, ..., n ]
  for i = 1 to n
    j := random number between 1 and n
    k := random number between 1 and n
    SWAP(A[j], A[k])
  return A

```

Solution

Each iteration of the algorithm has n^2 choices for the (i, j) pair that it swaps. Thus, overall, the algorithm can make n^{2n} choices for all the is and js . We can define a mapping from the set of (i, j) pairs to permutations

$$f : \{1, \dots, n\}^{2n} \rightarrow \text{Perms}(n)$$

The algorithm picks each combination of is and js with equal probability, so this mapping would have to be k -to-1 (for some k) in order for the algorithm to generate all permutations with equal probability. However, $n!$ does not divide n^{2n} (a proof of this was not necessary).

Therefore, some permutation must occur more often than another.

Grading

1. 4 points for stating that there are n^2 choices for an (i, j) pair
2. 6 points for stating that the algorithm makes n^{2n} choices altogether
3. 6 points for defining some sort of function between choices and permutations
4. 4 points for getting that there are $n!$ permutations and stating that $n! \nmid n^{2n}$