Recitation 5 — More Graphs
Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

September 28, 2011

Today’s Agenda:
- Announcements
- Records in ML
- Random Walks
- Maze Generation
- SSSP
- HW2 handback

1 Announcements

- We have a survey for you to fill out about the course – it’s posted on the bboard. Please do it so we can make the course better for you!
- Assignment 4 is due tomorrow at 11:59pm. Same late day policy as last week: you have until Saturday at 11:59pm to hand in, at the cost of 2 late days.
- Assignment 5 will go out on Friday. It won’t be due until after Midterm 1 on Oct 6—but there will be test-prep questions that you should attempt before the test.
- Questions about homework, class, life, universe?

2 Records in ML

Here’s a useful programming technique that will prevent bugs in your code and help us read and grade it.

Records are tuples whose elements are named.

Instead of

type point = int * int
gval start : point = (5,8)

we can write

type point = (x:int, y:int)
gval start : point = {y = 8, x = 5} (* any order! *)

You even have similar pattern matching utilities.
fun (p:point) = 
  let
    val {x=xcoord, y=ycoord} = p
  in 
    ...
  end

Warning: Mind the distinction between variables and labels! x and y are not bound by the let statement; xcoord and ycoord are.

A useful trick with records is “punning”: in the above example, we can abbreviate

fun (p:point) = 
  let
    val (x, y) = p
  in 
    ...
  end

and now we can use x and y as variables in the let body. Note that these names must match the fields of p exactly.

3 Warmup: Random Walks Through Graphs

This should be a very simple programming exercise compared to what you’re doing in homework 4, but let’s do it just to get warmed up.

A random walk is a path through a graph decided randomly.

Input: a graph $G$, a starting vertex $v$ and a path length $l$

Output: a path of length $l$ following a random walk through $G$ starting at $v$

type path = vertex seq
fun randWalk i G = 
  let
    fun randWalk’ 0 G _ p = p
    | randWalk’ i G v p = 
      let
        val next = getRandom (neighbors v)
      in 
        randWalk’ (i-1) G next (hidel (Cons (v, p)))
      end 
  in 
    randWalk’ i G (anyVertex G) empty
  end

We could use this to solve a simpler version of one of the problems you did on homework 3: Babble generation with a $k$ of 1. (Q: why only a $k = 1$? A: our random traversal has no “memory”. It only knows where it is, not where it’s been.)

Q: How would you represent the document as a graph?
A: A weighted, directed graph where vertices are words, edge \((x, y)\) is “x precedes y”. Need to tweak the code slightly to weight getRandom by edge weight.

XXX code to turn a document into such a graph?

4 Maze Generation

Problem: generate a random maze on a grid graph.

In a grid graph, nodes are cells and edges are walls. E.g.

```
  o----o----o----o
 /         /     / |
o----o----o----o
 /     /         |
o----o----o----o
 /     /         |
o----o----o----o
```

We can generate a random maze by traversing this graph and randomly destroying walls.

BFS: (XXX is this really BFS?)

- Start anywhere
- Look at neighbors – if any are unvisited, destroy the edges to them with some probability (density can be a parameter to the maze function)
- Add unvisited neighbors to the queue
- Recur

DFS:

- Start anywhere, add self to visited
- Choose unvisited neighbor randomly, remove the edge and add it to the queue
- Recur

TA note: don’t write the code for this – it might go on the homework. I decided it was too large for recitation.

5 Single Source Shortest Path (SSSP) Problem

Problem: Single source shortest path (SSSP)

Instance: A graph \(G = (V, E)\) and a source vertex \(v \in V\)

Solution: For every vertex \(u \in V\), the shortest path distance from \(v\) to \(u\).

Why won’t BFS work?

Simple counterexample:
5.1 Dijkstra’s Algorithm

Guy wants to cover this formally in lecture tomorrow, so this will just be a sneak preview of the real thing.

At a high level (imperatively), we:

- maintain a current node, starting at source \( s \), and a set of guessed distances for nodes we’ve seen, starting with \((s, 0)\)
- add the current node’s guessed distance to a table
- guess all my neighbors’ distances to be the current distance plus their edge weights
- choose my closest neighbor as the new current node; goto 2

Here’s an example graph:

Supose unit weight edges. How would a DFS shortest path calculation from node \( a \) look?

We’ll maintain a current node \( v \), a frontier \( F \) with our “guesses” for unvisited node distances, and a distance table \( D \) that we’ll return at the end.

Start with \( F = \{(a, 0)\} \) and \( D = [] \)

Step 1:
\( v = a, D = [(a, 0)], F = \{(b, 1), (c, 1), (e, 1)\} \)

Step 2:
\( v = b, D = [(a, 0), (b, 1)], F = \{(c, 1), (e, 1), (a, 2), (c, 2)\} \)

Step 3:
\( v = c, D = [(a, 0), (b, 1), (c, 1)], F = \{(e, 1), a, c, (b, 2), (a, 2), (d, 2), (e, 2)\} \)

Step 4:
\( v = e, D = [(a, 0), (b, 1), (c, 1), (e, 1)], F = \{(a, 2), (c, 2), (b, 2), (a, 2), (d, 2), (e, 2), (a, 3), (c, 3)\} \)
Step 5 - 8:
\(v = a, c, b\) already in \(D\) with a lower cost;
\(F = \{(d, 2), (e, 2), (a, 3), (c, 3)\}\)

Step 6:
\(v = d, D = [(a, 0), (b, 1), (c, 1), (e, 1), (d, 2)]\),
\(F = \{(e, 2), (a, 3), (c, 3), (c, 3)\}\)

Step 7 - 10:
Already in \(D\) with a lower cost

So we return the \(D\) above.