DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 202 NOTES
Spring 2010: Amber Settle

Week 6: Monday, May 3, 2010

Announcements

- The fourth assignment is due now
- The fifth assignment is due Monday, May 10th

Midterm

Statistics for both sections combined:

- High: 100
- Low: 28
- Average: 84.81
- 100 – 90: 9
- 89 – 80: 3
- 79 – 70: 1
- 69 and below: 3

Discuss the answers to the questions

Functions

We have seen equivalence and ordering relations, but there is a third type of relations that is fundamental to mathematics (and computer science) that we have not yet seen.

A binary relation R is called **single-valued** if $(\forall x)\ (\forall y)\ (\forall z)$ such that $R(x,y) \wedge R(x,z) \rightarrow y = z$.

A binary relation R is called **total** if $(\forall x)\ (\exists\ y)$ such that R(x,y).

A **function** is a single-valued, total, binary relation. Since by definition for every x there is a unique y such that R(x,y), we can write f(x) = y in what is called **function notation**.

Examples:

- R(x,y) = "x + y = 2" is single-valued and total so it defines a function, namely f(x) = y = 2 – x
- R(x,y) = "$x = y^2$" is not total nor single-valued over the reals so it is not a function. Over the complex numbers it is

total, but it is still not single-valued (since, for example, $3^2 = (-3)^2 = 9$)

Properties of functions

If $f \subseteq X \times Y$ is a function, we write it as $f: X \to Y$.

X is called the **domain** of f and Y is called the **range** of f.

$f(X) = \{f(x) : x \in X\}$ is called the **image** of f.

If $f(X) = Y$, we call f **onto** or **surjective**, which means that every element of the domain is taken on as a function value.

Examples:

- $f: \mathbb{N} \to \mathbb{N}$ that takes a number and removes its first digit, e.g. $f(1923) = 923$, is onto
- $f: \mathbb{N} \to \mathbb{N}$ that maps a number to the number of digits it has is nearly surjective. Why nearly? There is no number with no digits, so 0 is not in $f(\mathbb{N})$. So the function $f: \mathbb{N} \to \mathbb{N} - \{0\}$ is surjective.

If $f(x) = f(y) \to x = y$ for all $x,y \in X$, then f is **one-to-one** (sometimes written 1-1) or **injective**.

One-to-oneness expresses uniqueness.

Example: The function taking Americans with a social security number to their social security number.

Exercises: Which of the following functions are one-to-one and which are onto?

- $f: \mathbb{R} \to \mathbb{R}, x \mapsto x$
- $f: \mathbb{R} \to \mathbb{R}, x \mapsto x^2$
- $f: \mathbb{R} \to \mathbb{R}^{\geq 0}, x \mapsto x^2$ where $\mathbb{R}^{\geq 0}$ are the positive real numbers

Since every function is a relation, we can write f^{-1} for the **inverse relation to f**, that is, $\{(y,x) : y = f(x), x \in X\}$

If f is onto, then for every y there is an x such that $(y,x) \in f^{-1}$. If, in addition, f is one-to-one, then there is exactly one y such that $(y,x) \in f^{-1}$.

This means that f^{-1} is total and single-valued, so it is a function itself.

A function f which is one-to-one and onto is called **bijective**.

Lemma: If f: X → Y is bijective, then f^{-1} is a function from Y to X and f^{-1} is also bijective.

A bijection f: X → Y shows that X and Y are really the same set, up to a renaming of the elements of the set.

> **Example**: A bijection from {Amber, André, Erin} to {1, 2, 3} is f(Amber) = 1, f(Erin) = 2, and f(André) = 3.

The previous example highlights a shortcoming of bijections.

> While bijections show that two sets are the same up to a renaming of the elements, **they do not maintain any other sort of structure inherent in the set**.
>
> > For example, the set {Amber, André, Erin} doesn't have any natural order, but the set {1, 2, 3} does. So while the set {1, 2, 3} can be used to order {Amber, André, Erin}, the set {1, 2, 3} lost its order when associated with the set {Amber, André, Erin}.
>
> There are stronger types of bijections, called **isomorphisms**, that maintain the structure of the sets they are mapping. We will see them later when we discuss graph theory.

Exercises:

- How many different bijections are there from {Amber, André, Erin} to {1, 2, 3}?
- If there is a bijection between {Amber, André, Erin} and a set X, what can we say about the size of the set X?

Review: Databases and functions

We have actually seen functions in the context of databases, although we didn't call them that at the time.

There are two categories of functions in databases:

1. Aggregate functions that accumulate information over multiple records, such as count
2. Functions used on field values such as addition and multiplication

Examples:

- Average year that students started:
  SELECT avg(started)
  FROM Student;
- First and last year that 'Theory of Computation' has been offered:

```
SELECT min(year), max(year)
FROM Course, Enrolled
WHERE CI = CourseID AND CourseName = 'Theory of
Computation'
```

Different database systems vary wildly on the names of built-in functions and the details of how they are used, so we will avoid many detailed examples here.

Using functions: diagonalization

The following is a beautiful result (and one of my favorites ever since I used it correctly on one of my qualifying exams in graduate school).

Theorem (Cantor): There is no onto function from a set to its power set.
Proof: Let the set in question be X and suppose there is an onto function f: $X \rightarrow P(X)$.
Define a set A as follows: $A = \{x \in X: x \notin f(x)\}$.
A is a subset of X, i.e. $A \subseteq X$.
Further, the way we defined it, $A \neq f(x)$ for all x.
Why? Suppose that $A = f(x)$ for some x. But $x \in f(x)$ if and only if $x \notin A$, so the two sets differ on x.
This means that f is not onto.

This proof is known as **Cantor's diagonal argument**.

Visualize the proof as an infinite matrix.
The rows of the matrix are labeled with elements of X, and the columns with elements of P(X) listed as f(x) for $x \in X$.
The proof constructs a new set A by letting x be in A if and only if $x \notin f(x)$, so the set is constructed using the diagonal of the matrix.

Corollary: There is no one-to-one function from P(X) to X.

These two results show that the size of any set A is less than the size of its power set P(A). The results hold for finite and infinite sets!

Games and graphs

Nearly all of the rest of the course will be motivated by games and puzzles.

Modeling a maze

Consider a simple maze. See **Figure 1** on the separate sheet.

To search through a maze there are several ways to proceed:

1. **Locally**: Using only information from the part of the maze in which you find yourself immediately
    a. Right-hand rule: Place your hand on the right wall and keep it there while walking
    b. Left-hand rule
    c. Random decisions at intersections, which works surprisingly well
2. **Globally**: Using a layout of the maze in front of you which can be analyzed
3. **Intermediate**: Using a combination of the two
    a. Marking intersections
    b. Drawing a plan of the maze

Let's concentrate on the **global perspective**.

The drawing of the maze contains a lot of extraneous information.

We can simplify it a great deal if we concentrate on **the important parts**:

- Entrance
- Exits
- Forks
- Dead ends

We can place nodes at these locations and connect them by lines to model connections between those locations.

See **Figure 2** on the separate sheet.

We can also visualize the connections without the "walls" around the maze.

It still contains all of the information we need to solve the maze.

See **Figure 3** on the separate sheet.

Without the "walls" there is no reason to retain the geometry.

The maze can be redrawn into what is called a graph.

See the **last diagram** on the separate sheet.

At this point the solution to this maze has become trivial to determine.

And because in constructing the graph we retained the order in which the connections leave the nodes, it is easy to

translate a walk through the graph back into a walk through the maze.

Graph definitions

A graph is a collection of vertices (also known as nodes) which can be connected by edges.

More formally, a **graph** G is a pair G = (V,E) of vertices V and edges E, where an edge in E is a set of two vertices.

For **vertices** we typically use letters like u, v, s, t, w, and for **edges** we use e, f, g.

If **e is the edge from u to v**, then we formally write e = {u, v} but we can write e = uv or e = (u,v) for simplicity.

The vertices u and v are the ends of edge uv, and we say that u and v are **incident** to the edge uv.

Since e is the set {u,v} = {v, u}, we do not (yet) distinguish between uv and vu. This is true even if we write e = (u,v) until we consider directed graphs later in the course.

Note: By definition of an edge, we do not allow an edge from a vertex to itself or multiple edges between the same pair of vertices.

Those features are allowed in multigraphs, in which case what we call graphs are called simple graphs.

In the motivating example, we translated a walk through a maze into a walk through a graph.

A **walk** in a graph is a sequence $u_1, e_1, u_2, e_2, \ldots e_{n-1}, u_n$ of vertices and edges that traverses the graph.

This means that you start at vertex u_1, then take edge e_1 to vertex u_2 and so on. This means that $e_1 = u_1u_2$, $e_2 = u_2u_3$, etc.

You can also write this as: $e_1e_2e_3 \ldots e_{n-1}$ if the edges e_i are known.

There are some special walks that are of interest:

1. A **path** is a walk in which every vertex occurs at most once.
2. A **cycle** is a walk with $n \geq 3$ whose first and last vertex are the same, $u_1 = u_n$, but there are no other vertices that repeat.
3. A **trail** is a walk which contains every edge at most once.

Exercises:

- Show that in a path every edge can occur at most once (which shows that every path is a trail)
- Construct a graph and a trail in that graph in which every edge in the trail occurs at most once, but there are vertices that occur multiple times (which shows that not every trail is a path)

Proposition: If G contains a walk from s to t, then it contains a path from s to t.
Proof: If G contains a walk from s to t, let $u_1, e_1, u_2, e_2, \ldots e_{n-1}, u_n$ be shortest such walk.

Suppose that some vertex occurs twice on that walk.
This means $u_i = u_j$ for some $1 \leq i < j \leq n$.

Then $u_1, e_1, u_2, e_2, \ldots, e_{i-1}, u_j, e_j, u_{j+1}, \ldots e_{n-1}, u_n$ is also a walk from s to t contained in G and is shorter than the original walk.

But this is a contradiction to the assumption that the original walk was a shortest walk.

So a shortest walk in the graph G must not have any repeated vertex and must in fact be a path.

A **graph can also be seen as a relation** on its vertices, where two nodes are related if they share an edge.

For example, consider the graph G = {(a,b), (a,c), (b,d), (c,d), (d,e), (d,f), (e,g), (f,g)}.

This is a relation on the vertices a, b, c, d, e, f, and g.
Draw the graph.

This is the notation I will often use on assignments.

Paths and cycles in graphs

Consider G in the previous example.

In this graph, a, ab, b, bd, d, de, e, eg, g, gf, f, fd, d, dc, c is a walk from a to c. **Highlight the walk in the graph**.

It is not a path since the vertex d is repeated.

However, the part of the walk d, de, eg, g, fg, f, fd, d is unnecessary to get from a to c, and can be removed from the walk. Highlight the removed part of the walk.

When we remove this, we get a path a, ab, b, bd, d, dc, c from a to c. **Highlight the path in the graph**.

This is not the shortest path since we could simply walk a, ac, c.

Note that **the excursion we cut out of the walk is a cycle**.

There is another cycle in the graph: a, ab, b, bd, d, dc, c, ac, a.

We consider this to be the same cycle as b, bd, d, dc, c, ac, a, ab, b since they are two ways of writing the same four edges and vertices in the graph, just starting at a different place.

With this convention there are only two cycles in the graph, since the joining of the two cycles causes the vertex d to be repeated.

Finding paths

Solving a maze corresponds to finding **a path between two given vertices s and t in a graph**.

How do we do this? In small graphs (like the one we saw in the motivating example), trial and error works fine.

What about when the graph is large? Finding a path between two corners in a graph representing streets in the United States would not lend itself well to trial and error.

How do we take a systematic approach to determining how to get from a vertex s to a vertex t in a graph?

The approach we will use is called breadth-first search.

Breadth-first search

Breadth-first search is a conservative approach to the problem.

It starts by seeing what vertices can be reached from s directly. And then expands the search from there.

The set of vertices that can be reached from s directly is called the neighborhood of the vertex.

The **neighborhood N(u)** of a vertex u is defined by $N(u) := \{v : uv \in E\}$

If t is among the vertices in N(s), then we are done.

If not, then we check the neighborhoods of the vertices in N(s), that is the vertices that can be reached in two steps from s, or in other words, the neighborhoods of the neighborhood of s.

There is no reason to stop at 2 (unless we happen to reach t).

$N_k(u) := \{v : v$ can be reached in at most k steps from u$\}$ is the k-neighborhood of u.

This means that $N_0(u) = \{u\}$ and $N_1(u) = N(u) \cup \{u\}$.

The **k-neighborhood** arises very naturally step by step (or inductively or recursively) as follows:

- $N_0(u) = \{u\}$
- For $k > 0$, $N_k(u) := N_{k-1}(u) \cup \{w: vw \in E \wedge v \in N_{k-1}\}$

Breadth-first search: To find a path from s to t, find the smallest k such that $t \in N_k(s)$

Consider an example. Let **G be the graph obtained from the maze** with the nodes labeled a through t, where the entrance is s and the exit is t.

Specifically G = (V,E) where V = {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t} and E = {{s,a}, {a,b}, {b,c}, {b,d}, {d,e}, {d,f}, {a,h}, {h,g}, {h,i}, {i,j}, {i,k}, {k,l}, {k,n}, {n,t}, {n,m}, {n,p}, {p,q}, {p,r}}

Draw the graph.

We begin at s, and $N_0(s) = \{s\}$.

In at most one step we reach $N_1(s) = \{s, a\}$
In at most two steps we reach $N_2(s) = \{s, a, b, h\}$.

We quickly see that it's more interesting to consider the new vertices we obtain.

Let **the k-boundary $B_k(u)$** be the vertices that can be reached in k steps from vertex u but not by any shorter path.
Formally $B_k(u) = N_k(u) - N_{k-1}(u)$.

Going back to our example graph, we find that:

$B_0(s) = \{s\}$
$B_1(s) = \{a\}$
$B_2(s) = \{b, h\}$
$B_3(s) = \{c, d, g, i\}$
$B_4(s) = \{e, f, j, k\}$
$B_5(s) = \{l, n\}$
$B_6(s) = \{m, t, p\}$
$B_7(s) = \{q, r\}$
$B_8(s) = \emptyset$

At this point we have seen the entire graph.

Breadth-first search in pseudocode

Using pseudocode we can describe the strategy of breadth-first search more completely.

This will use a queue. A **queue** is an ordered collection in which elements are added to the end and taken out from the beginning (like the line at a movie theater or post office, etc.)

The **queue operations** we need:

- **append** (q, x): Add element x to the (end of) the queue
- **pop**(q): Remove the first element from the (beginning of) the queue

In order to keep track of the progress of the breadth-first search, we will keep an array **distance that has an entry for each vertex in the graph**. It represents the distance from the source vertex (or starting place) to the vertex specified.

Initially the distance of each vertex is infinity, which we will represent by a -1 in our algorithm

As the algorithm works, if we visit a previously unvisited node v which is a neighbor of a node c we know that v has distance one greater than c (since otherwise we would have visited it previously).

The algorithm is as follows:

```
breadth_first_search(G, s, t)
    for each v in G          // initially very vertex has distance infinity
        distance[v] = -1
```

```
reached := {s}          // the current boundary
distance[s] := 0        // s is 0 distance from itself

while (reached not empty)     // vertices whose neighbors need
                              // to be explored
        c := pop(reached)     // take the first element of the queue
        if (c == t)           // if we are at the goal, stop
                return true   // This is the exit condition
        for each v in N(c)    // for every node in c's neighborhood
                if (distance[v] == -1)          // if not visited,
                        append(reached, v)      // append to queue
                        distance[v] := distance[c] + 1
return false    // we did not find the exit while exploring the graph
```

This algorithm tells us whether there is a path from s to t in G.

It is solving the **s-t-connectivity problem**.

Example: Let's re-do the example we did earlier using the algorithm.

reached = {s}
Pop s off the queue
    distance[s] = 0
    Add N(s) = {a} to reached

reached = {a}
Pop a off the queue
    distance[a] = distance[s] + 1 = 1
    Add unreached neighbors of a, i.e. {b, h}, to reached

reached = {b, h}
Pop b off the queue
    distance[b] = distance[a] + 1 = 1 + 1 = 2
    Add unreached neighbors of b, i.e. {c, d}, to reached

reached = {h, c, d}
Pop h off the queue
    distance[h] = 2
    Add unreached neighbors of h, i.e. {g, i}, to reached

reached = {c, d, g, i}
Pop c off the queue
    distance[c] = 3
    c has no unreached neighbors

reached = {d, g, i}

Pop d off the queue
    distance[d] = 3
    Add unreached neighbors of d, i.e. {e, f}, to reached

reached = {g, i, e, f}
Pop g off the queue
    distance[g] = 3
    g has no unreached neighbors

reached = {i, e, f}
Pop i off the queue
    distance[i] = 3
    Add unreached neighbors of i, i.e. {j, k}, to reached

reached = {e, f, j, k}
Pop e off the queue
    distance[e] = 4
    e has no unreached neighbors

reached = {f, j, k}
Pop f off the queue
    distance[f] = 4
    f has no unreached neighbors

reached = {j, k}
Pop j off the queue
    distance[j] = 4
    j has no unreached neighbors

reached = {k}
Pop k off the queue
    distance[k] = 4
    Add unreached neighbors of k, i.e. {l, n}, to reached

reached = {l, n}
Pop l off the queue
    distance[l] = 5
    l has no unreached neighbors

reached = {n}
Pop n off the queue
    distance[n] = 5
    Add unreached neighbors of n, i.e. {m, t}, to reached

reached = {m, t}
Pop m off the queue

distance[m] = 6
m has no unreached neighbors

reached = {t}
Pop t off the queue
distance[t] = 6
exit, returning true

Breadth-first search correctly finds the shortest distance between the starting vertex and the goal vertex because it proceeds conservatively, always considering shorter paths before longer paths.

On the other hand, it sometimes branches into directions that aren't promising before moving toward what is obviously the goal of the search.

Graph connectivity

Is it possible that there would be no path between two vertices s and t in a graph?

In that case, s and t must be in different parts of the graph.

This means that the graph is **disconnected**, that is, there are two vertices in the graph which are not connected to each other by a path.

A graph is **connected** if there is a path from every vertex to every other vertex in the graph.

Exercises:

- Draw a graph that has two vertices that cannot be reached from each other
- What is the smallest graph (in terms of number of vertices) in which two vertices may not be able to reach each other? What does it look like?
- What is the smallest graph (in terms of number of edges) in which two vertices may not be able to reach each other? What does it look like?

How can we use breadth-first search to determine if a graph is connected?

Run the algorithm on an arbitrary vertex and count how many vertices are visited in the search. (Assuming that we remove the condition to exit from breadth-first search). If the number of vertices reached is the same as the number of vertices in the graph, then the graph is connected. Otherwise it is not.

Finding spanning trees

Consider the following graph G = (V, E):

V = {a, b, c, d, e, f, g, h, s, t}
E = {{s,b}, {s,a}, {a,d}, {d,f}, {b,f}, {b,d}, {a,c}, {c,g}, {d,e}, {c,e}, {e,h}, {f,h}, {g,t}, {h,t}}

Draw the graph G.

Let's run breath-first search (without the exit condition) on this graph, starting from the vertex s.

reached = {s}
Pop s off the queue
    distance[s] = 0
    Add N(s) = {a, b} to reached

reached = {a,b}
Pop a off the queue
    distance[a] = 1
    Add unreached neighbors of a, i.e. {c,d}, to reached

reached = {b, c, d}
Pop b off the queue
    distance[b] = 1
    Add unreached neighbors of b, i.e. {f}, to reached

reached = {c, d, f}
Pop c off the queue
    distance[c] = 2
    Add unreached neighbors of c, i.e. {e, g} to reached

reached = {d, f, e, g}
Pop d off the queue
    distance[d] = 2
    There are no unreached neighbors of d

reached = {f, e, g}
Pop f off the queue
    distance[f] = 2
    Add unreached neighbors of f, i.e. {h} to reached

reached = {e, g, h}
Pop e off the queue
    distance[e] = 3
    There are no unreached neighbors of e

reached = {g,h}
Pop g off the queue
distance[g] = 4
Add unreached neighbors of g, i.e. {t} to reached

reached = {h, t}
Pop h off the queue
distance[h] = 3
There are no unreached neighbors of h

reached = {t}
Pop t off the queue
distance[t] = 4
There are no unreached neighbors of t

reached = ∅

First note that all the vertices in the graph have been visited.

What does that mean about the graph?
It is connected.

Which edges are traversed in this breadth-first search?

{s,a}, {s,b}, {a,d}, {b,f}, {f,h}, {a,c}, {c,e}, {c,g}, and {g,t}

Consider a subgraph T of G which consists of all the vertices and only the edges above. Does T have a cycle? Why not?

A **tree** is a connected graph that does not contain a cycle.

Our search algorithms show that every connected graph contains a tree on all the vertices of the graph. Such a tree is called a **spanning tree**.

Theorem: Every connected graph contains a spanning tree.
Proof:
Let G be a connected graph.
Run breadth-first search (without the exit condition) on G.
During the search, if some vertex v is visited from some vertex u, then draw an arrow from u to v.
Consider the graph T that consists of all the arrowed edges.
Our claim is that T contains the same vertices as G.

Every vertex is visited at most once by definition of either algorithm, so there is at most one arrow pointing toward each vertex.

Suppose T contains a cycle.
Let v be the vertex on C that was visited first in the search.
Then the two edges of C adjacent to v point away from v (since it was the first vertex on C).
But then C must contain two edges that point toward the same vertex.

To see this, follow the arrows starting at v until you run into an edge pointing the other way. That must happen, since the two edges at v point in different directions along C. (In the worst case it will happen when the two arrows "meet" on the other side of the cycle).

But this is not possible. Every vertex has at most one arrow pointing at it. So T cannot contain a cycle.

Summary

We have seen how to use breadth-first search on graphs to solve three problems:

1. **st-connectivity problem**: Given a graph G and two vertices s and t, can t be reached from s?
2. **Shortest path problem**: Given a graph G and two vertices s and t, find a shortest path from s to t, if one exists.
3. **Spanning-tree problem**: Given a graph G, find a tree that connects all the vertices of G.

The solutions to these problems are efficient.

Each vertex in the graph is visited only once by both breadth-first search, so that each algorithm runs in time proportional to the size of the graph.