Admin

- Reading
  - Chapter 9
  - Skipping 8 (optional reading if you want)
- Homework 4 out soon
  - Due April 5 (Almost 3 weeks from now)
- Project
  - Should be making some solid progress here
  - Don’t wait until the last minute!

What have we been talking about?

- What did we talk about last time?
- Graphs:
  - Overview
  - ADT
  - Representations:
    - Adjacency Matrix
    - Adjacency List
  - NP Completeness
  - TSP
  - Coloring

Today: Finding a path

- Given a graph, find a path from a start node to an end node
  - E.g., from A to F

- Given a graph, find a path from a start node to an end node
  - E.g., from A to F
  - Most obvious choice: A -> C -> F
  - How would we find this algorithmically though?
Today: Finding a path

• Graph search algorithm 1: Depth First Search (DFS)
  • From = A
  • To = F
  • A is adjacent to B, so if we can find a path from B to F, then easy
    • (That path + A on the front)

Today: Finding a path

• Graph search algorithm 1: Depth First Search (DFS)
  • From = B
  • To = F
  • B is adjacent to E, so maybe we can find a path from E to F?

Today: Finding a path

• Graph search algorithm 1: Depth First Search (DFS)
  • From = E
  • To = F
  • No, this looks like a dead end...
  • But we had other options at B

Today: Finding a path

• Graph search algorithm 1: Depth First Search (DFS)
  • From = D
  • To = F
  • D is adjacent to C...

Today: Finding a path

• Graph search algorithm 1: Depth First Search (DFS)
  • From = C
  • To = F
  • C is adjacent to F...
Today: Finding a path

Graph search algorithm 1: Depth First Search (DFS)
- From = F
- To = F
- Finding a path from F to F is trivial (same node)

DFS: Algorithm (v 1.0)
- Naturally recursive:
  - dfs(f,t)
    - If f is the same as t
      - Return the path [f]
    - Otherwise,
      - For each node N adjacent to f
        - X = dfs(N, t)
        - if X is a valid path then
          - The answer is the path with f on the front of X
        - If you don’t find any valid path from any adjacent node, then give back an answer of “no valid paths”

Let’s try this for G to E

Nodes adjacent to G:
A, C, F

Do A first (we’ll assume they are Ordered alphabetically to Be concrete)
DFS: Algorithm (v 1.0)

• DFS(A, E)
  • If A is the same as E
    • Return the path [A]
  • Otherwise,
    • For each node N adjacent to A
      • X = dfs(N, E)
        • if X is a valid path then
          • The answer is the path with A on the front of X
        • If you don’t find any valid path from any adjacent node, then give back an answer of “no valid paths”

ECE 590.01 (Hilton): Graphs II

Now recursively doing dfs(A, E)

Recursion stack:

DFS: Algorithm (v 1.0)

• DFS(A, E)
  • If A is the same as E
    • Return the path [A]
  • Otherwise,
    • For each node N in {B, C, D, G}
      • X = dfs(N, E)
        • if X is a valid path then
          • The answer is the path with A on the front of X
        • If you don’t find any valid path from any adjacent node, then give back an answer of “no valid paths”

ECE 590.01 (Hilton): Graphs II

Now recursively doing dfs(B, E)

Recursion stack:

DFS: Algorithm (v 1.0)

• DFS(B, E)
  • If B is the same as E
    • Return the path [B]
  • Otherwise,
    • For each node N adjacent to B
      • X = dfs(N, E)
        • If X is a valid path then
          • The answer is the path with B on the front of X
        • If you don’t find any valid path from any adjacent node, then give back an answer of “no valid paths”

ECE 590.01 (Hilton): Graphs II

Now recursively doing dfs(A, E)

Recursion stack:

DFS: Algorithm (v 1.0)

• DFS(A, E)
  • If A is the same as E
    • Return the path [A]
  • Otherwise,
    • For each node N in {B, C, D, G}
      • X = dfs(N, E)
        • If X is a valid path then
          • The answer is the path with B on the front of X
        • If you don’t find any valid path from any adjacent node, then give back an answer of “no valid paths”

ECE 590.01 (Hilton): Graphs II

Now recursively doing dfs(B, E)

Recursion stack:

DFS: Algorithm (v 1.0)

• DFS(B, E)
  • If B is the same as E
    • Return the path [B]
  • Otherwise,
    • For each node N in {A, D, E}
      • X = dfs(N, E)
        • If X is a valid path then
          • The answer is the path with B on the front of X
        • If you don’t find any valid path from any adjacent node, then give back an answer of “no valid paths”

ECE 590.01 (Hilton): Graphs II

Now recursively doing dfs(A, E)

Recursion stack:

DFS: Algorithm (v 1.0)

• DFS(A, E)
  • If A is the same as E
    • Return the path [A]
  • Otherwise,
    • For each node N in {B, C, D, G}
      • X = dfs(N, E)
        • If X is a valid path then
          • The answer is the path with A on the front of X
        • If you don’t find any valid path from any adjacent node, then give back an answer of “no valid paths”

ECE 590.01 (Hilton): Graphs II

Now recursively doing dfs(B, E)

Recursion stack:

DFS: Algorithm (v 1.0)

• DFS(A, E)
  • If A is the same as E
    • Return the path [A]
  • Otherwise,
    • For each node N in {B, C, D, G}
      • X = dfs(N, E)
        • If X is a valid path then
          • The answer is the path with A on the front of X
        • If you don’t find any valid path from any adjacent node, then give back an answer of “no valid paths”

ECE 590.01 (Hilton): Graphs II

Now recursively doing dfs(A, E)

Recursion stack:
Why now?

- This problem: not in lists/trees, but in graphs
  - Why?
  - Lists and trees do not have cycles
    - Moving "down" a list naturally moves you closer to the end (in one direction). Can't go backwards
  - Graphs can have cycles:
    - No clear notion of "forward progress" by moving to an adjacent node (may have already been there).

- Before Question 2 (how do we fix it), let's talk about termination in general
  - Termination = ending: not going into infinite loop/recursion

Termination

- How can you be sure your algorithm terminates
  - For all possible inputs?

- How can you be sure your algorithm terminates
  - For all possible inputs?
  - Ask this guy?
  - No, that's a Terminator

- How can you be sure your algorithm terminates
  - For all possible inputs?
  - Ask this guy?
  - No, that's a Terminator

- How can you be sure your algorithm terminates
  - For all possible inputs?
  - Ask this guy?
  - No, that's a Terminator

- Establish a "measure" (sometimes called "energy") which
  - Is an ordinal number
  - For simplicity: think natural numbers
  - Decreases with every "step"
    - Loop iteration or recursive call
  - Must eventually terminate, since ordinals (and naturals) have lower bound: 0
  - For lists, trees, etc:
    - Length of (remaining) list, height of (remaining tree) etc work

- We aren't going to prove termination
  - We don't do proofs in this course

- ...but its good to know what the right tool to reason about your code is
  - Write some big complex algorithm and want to know "does this always terminate?" 
  - Now you know how to approach it
Termination: Aside

Why doesn't our compiler check if our code terminates?

• Would be a really nice feature:
  • Warning: you wrote an infinite loop in graph.cpp: 107-142
  • How would you write an algorithm to see if code terminates?

Let's think about what we would do

• First, lets imagine that we could write this function:
  
  ```c++
  bool halts(string code, string input) {
    //magic goes here
  }
  ```

  This function takes two inputs:
  • `code`: the C++ source code for a function as a string
  • `input`: the input we want to test

  And returns true or false, depending on if the function in
  "code" halts on the given input

  ```c++
  halts("void f(string x) { return x.length() > 4; }", "abc");
  ```

  • Returns true

Now, let us suppose I write this...

```c++
bool weird(string a) {
  if(halts(a,a)) {
    while(1) { ; }
  }
  return false;
}
```

If the function represented by the string a halts on input a,

Else return false.

Strange, but valid...

Now, let us suppose I write this...

```c++
string s = "bool weird(string a) {
  if(halts(a,a)) {
    while(l) { ; }
  }
  return false;
}"
```

What is `halts(s,s);`?

• Cup in front of me: it returns true
• Cup in front of you: it returns false

(The cup with the wrong answer is poisoned)

Clearly its not in the cup in front of me

```c++
string s = "bool weird(string a) {
  if(halts(a,a)) {
    while(l) { ; }
  }
  return false;
}"
```

What if `halts(s,s);` returns true?

That means that `weird(s)` halts...

but if that's the case, then `weird(s)` goes into an infinite loop... so it does not halt

Ok, so it must be in the cup in front of you?
Clearly its not in the cup in front of you

```cpp
string s = "bool weird(string a) {
    if(halts(a,a)) {
        while(1) {
            
        }
    return false;
}

What if halts(s,s); returns false?
That means that weird(s) does not halt...
but if that's the case, then weird(s) just returns false...
so it does halt
```

So where is the poison?

What is halts(s,s);?
Cup in front of me: it returns true
Cup in front of you: it returns false
(The cup with the wrong answer is poisoned)

So where is the poison?

What is halts(s,s);?
Both cups (Neither answer is correct)
Cup in front of me: it returns true
Cup in front of you: it returns false
(The cup with the wrong answer is poisoned)

It cannot possibly return true
...and it cannot possibly return false

So the only possibility is that halts(s,s) cannot give the right answer (either its wrong, or goes into an infinite loop)—it's impossible to write it correctly.

Ok, back to graphs

- Undecideability: some programs cannot be written
  - At least not with models of computation as we understand them
  - Maybe if you have a fundamentally different way to compute you could

- Those were a few important lessons
  - Back to graphs, and fixing our DFS

DFS: Algorithm (v 1.0)

- Now, let's see how to fix it
  - Problem: went back to A after we already went there
  - Do you all know the joke about the guy who goes to the Dr...
DFS: Algorithm (v 1.0)

Dr, Dr! My algorithm doesn’t terminate when I go back to nodes I already visited!

Well, don’t go back to nodes you already visited!

How do we detect this?

Keep a Set of nodes we visited, check if it contains "from" already

I told you Sets were great: right?

DFS: Algorithm (v 2.0)

- dfs(f,t,visited)
  - If f is the same as t
    - Return the path [f]
  - If visited contains f
    - Give back an answer of no valid path
  - Put f into visited
  - Otherwise,
    - For each node N adjacent to f
      - X = dfs (N, t, visited)
        - If X is a valid path then
          - The answer is the path with f on the front of X
          - If you don’t find any valid path from any adjacent node, then
            give back an answer of “no valid paths”

DFS: Algorithm (v 2.0b)

- dfs(F,T)
  - S is an empty stack (of Paths)
  - V is an singleton set (of Nodes) with F in it
  - Push the path [F] onto S
  - As long as S is not empty
    - P is the result of popping the top of S
    - N is the last node on the path P
    - If N is equal to T
      - Then return P
    - For each X adjacent to N
      - If V does not contain X then
        - Add X to V
        - Push the path (P with X added to the end) onto S
  - When the stack is empty, there are no possible paths

DFS vs BFS

- Depth first search vs Breadth first search

  Conceptually
  - DFS = exploring a maze with yarn to lead you back the way you came
  - BFS = pouring water in at “from”, seeing how it gets to “to”
BFS

• First explore A -> B
  • Same start as before
  • But then...

ECE 590.01 (Hilton): Graphs II

DFS: Algorithm (v 2.0b)

• dfs(F,T) queue
  • S is an empty stack (of Paths)
  • V is a singleton set (of Nodes) with F in it
  • Push the path (F) onto S
  • As long as S is not empty
dequeue from
  • P is the result of popping the top of S
  • N is the last node on the path P
  • If N is equal to T
  • Then return P
  • For each X adjacent to N
  • If V does not contain X then
  • Add X to V
  • Enqueue P with the path (P with X added to the end) onto S
• When the stack is empty, there are no possible paths

ECE 590.01 (Hilton): Graphs II

BFS

• Let’s see this in action
  • Queue [A, B], [A, C], [A, D], [A, G]
  • Visited A, B, C, D, G

ECE 590.01 (Hilton): Graphs II
BFS

Let's see this in action

Queue [A,D], [A,G], [A,B,E], [A,C,F],
Visited A,B,C,D,E,F,G

ECE 590.01 (Hilton): Graphs II

55

ECE 590.01 (Hilton): Graphs II

56

Let's see this in action

Queue [A,G], [A,B,E], [A,C,F],
Visited A,B,C,D,E,F,G

ECE 590.01 (Hilton): Graphs II

57

ECE 590.01 (Hilton): Graphs II

58

Shortest (weighted) Path

BFS finds a path with the fewest number of "hops"
But what if we want the shortest by total edge weight
May not be fewest hops

ECE 590.01 (Hilton): Graphs II

59

Dijkstra's Algorithm (w/ a PQ)

dijkstra(F,T)

Q is an empty priority queue (of Paths, ordered by total weight)
V is the empty set (of Nodes)
Push the path [F] onto Q
As long as Q is not empty
  P is the result of dequeuing from Q
  N is the last node on the path P
  If N is equal to T
    Then return P
  If N is not in V, then
    Add N to V
    For each X adjacent to N
      Enqueue the path (P with X added to the end) onto S
  When the stack is empty, there are no possible paths

ECE 590.01 (Hilton): Graphs II

60
Two notes on this

1: You may see Dijkstra’s algorithm presented many ways. BFS with a priority queue instead of a regular queue is easiest to understand/implement.

2: I re-arranged visited set operations occur
   • Either of these ways is valid for a BFS
   • The earlier way does not work for Dijkstra’s
   • It may omit a shorter (weighted) path found later in the process
   • We now “visit” something only when we are processing its last node, at which point we have found the shortest path to it

Any graph search algorithm you want

1. dijkstra(F,T)
   • Q is an empty priority queue (of Paths, ordered by total weight)
   • V is the empty set (of Nodes)
   • Push the path [F] onto Q
   • As long as Q not empty
   • P is the result of dequeuing from Q
   • N is the last node on the path P
   • If N is equal to T then
     • Return P
   • If N is not in V then
     • Add N to V
     • For each X adjacent to N
     • Enqueue the path (P with X added to the end) onto S
   • When the stack is empty, there are no possible paths

Generic Graph Searching

• Could template over DS type and write once
• Could make an abstract class with 3 sub classes
• Both of these require making the function names uniform
  • Push vs enqueue
  • Pop vs dequeue

Next time: More graphs!

• Next time, we’ll continue with graphs