Notes

Prerequisites:

- **C++**
  - structs, classes, templates, iterators, managing pointers
- **ieng6**
  - If ieng6 still does not work, please stop by after class or find the ACMS thread on Piazza
  - you can also visit ACMS in-person on the first floor of APM
- **Starter code is distributed via GitHub**
  - [https://github.com/UCSD-CSE101-W16/PA1](https://github.com/UCSD-CSE101-W16/PA1)

Miscellaneous:

- Piazza - please set questions to public to avoid duplicate private questions
- Review turn-in instructions issued on Piazza
- Testing suite has been provided. Review testsrc folder and the Makefile
Templates<T>

- Templates help make algorithms code **useful** by allowing a programmer to reuse routines written for any object that “fits”.
  - We will see DFS reused for several routines containing different objects in PA1!
- Routines may require objects to implement certain functionalities (e.g. `==`)
  - Operator defining / overloading to satisfy criterium
  - Comparators are already implemented if you use standard types like int
- Using templates will save you time in the long run!
- Also note that you may not modify the method signatures. If needed, you can create a helper method and call that from your method.


Graph.hpp

- All questions will deal with directed graphs
- Adjacency list format
- id is templated for your convenience
- also included are pre, post and visited in case you would like to use them in your algorithms
  - NOTE: if you do use pre/post/visited, make sure that you have properly “sanitized” them before use, i.e. before starting a traversal, set all vertices’ visited to false.
- review the C++11 Standard Template Library prior to coding!!
  - this will help tremendously with map, list, iterator, etc.
Problem 1: DFS

Given directed graph G and vertex V, determine all vertices reachable from V in G.

- Standard DFS search reviewed in lecture.
  - pseudocode can be found in lecture slides
- std::set<T> is an ordered data structure -- don’t worry about insertion order.

Format for file input:
```
1  The first line is always the ID of the starting vertex
1 2  Each subsequent line defines a directed graph edge where
2 3  the order is source vertex -> destination vertex.
2 4
3 4
4 5
6 5
```

Sample Output:
```
$ ./TestDFS.out input_file
Result of DFS: [1 2 3 4 5]
```

Notice: vertex 6 is not in output since it is not reachable from 1.
Problem 2: Topological Ordering

● Given directed graph, find a topological ordering if one exists
  ○ note: the graph must be acyclic in order for a topological ordering to exist
● Refer to lecture slides for two viable algorithms (Lecture 3, slide 3)
● Kahn’s Algorithm (“Inductive” thinking):
  ○ compute the indegree of each vertex
  ○ use the fact that a source vertex must have indegree 0
  ○ after finding and removing a source vertex, its neighbors may then become source vertices
  ○ repeat for the whole graph
  ○ things to think about: what happens when there is a cycle? what about a self-loop?
● Tarjan’s Algorithm (DFS):
  ○ perform DFS marking post numbers accordingly
  ○ does ordering vertices by largest post number produce a topological ordering?
    ■ yes, but only if input graph is a DAG
  ○ things to think about: how can you detect if the graph is not a DAG?
Problem 3: Strongly Connected Components

Several ways to solve this problem

- **Kosaraju’s Algorithm - covered in class**
  - Find sink SCC by finding source of $G^R$, run DFS in descending ‘post’ order

- **Tarjan’s Algorithm - not covered in class**
  - A modified DFS that takes advantage of the fact that there are no back edges between vertices of unique SCC’s.
Problem 3: Strongly Connected Components

Several ways to solve this problem

- **Kosaraju’s Algorithm** - covered in class
  - Find sink SCC by finding source of $G^R$, run DFS in descending ‘post’ order

- **Tarjan’s Algorithm** - not covered in class
  - A modified DFS that takes advantage of the fact that there are no back edges between vertices of unique SCC’s.
Problem 3: Strongly Connected Components

Algorithm
Run DFS on $G^R$
for $v \in V$ in decreasing order of post numbers
    if not visited[$v$]
        explore($G,v$)
        output nodes seen as a SCC

~5 lines pseudocode!
Problem 3: Strongly Connected Components
Problem 3: Strongly Connected Components

Algorithm
Run DFS on $G^R$
for $v \in V$ in decreasing order of post numbers
  if not visited[$v$]
    explore($G,v$)
  output nodes seen as a SCC
Problem 3: Strongly Connected Components

**Algorithm**

Run DFS on $G^R$

for $v \in V$ in decreasing order of post numbers
  if not visited[v]
    explore(G,v)
  output nodes seen as a SCC
Problem 3: Strongly Connected Components

2

Run \textbf{DFS} on $G^R$

for $v \in V$ in decreasing order of post numbers

if not visited[v]

explore(G,v)

output nodes seen as a SCC
Problem 3: Strongly Connected Components

Run DFS on $G^R$

for $v \in V$ in decreasing order of post numbers

if not visited[v]

explore(G,v)

output nodes seen as a SCC
Problem 4: Worm

Key:
- Green - Worm Location
- Red - Food Obstacle
- Blue - Target End Location

Note: there can be many food obstacles
Problem 4: Worm

Key:

Green - Worm Location
Red - Food Obstacle
Blue - Target End Location

Note: there can be many food obstacles
Problem 4: Worm

Key:
Green - Worm Location
Red - Food Obstacle
Blue - Target End Location

Note: there can be many food obstacles
Problem 4: Worm

Key:
Green - Worm Location
Red - Food Obstacle
Blue - Target End Location

Note: there can be many food obstacles
Problem 4: Worm

Key:
- Green - Worm Location
- Red - Food Obstacle
- Blue - Target End Location

Note: there can be many food obstacles
Problem 4: Worm

Key:

Green - Worm Location
Red - Food Obstacle
Blue - Target End Location

Note: there can be many food obstacles
Problem 4: Worm

Key:
Green - Worm Location
Red - Food Obstacle
Blue - Target End Location

Note: there can be many food obstacles
Problem 4: Worm

Key:
- Green - Worm Location
- Red - Food Obstacle
- Blue - Target End Location

Note: there can be many food obstacles
Problem 4: Worm

What about this case?

Key:
- Green - Worm Location
- Red - Food Obstacle
- Blue - Target End Location

Note: there can be many food obstacles
Problem 4: Worm

- The worm cannot move to the end location
Problem 4: Worm

- We’ve looked many problems dealing with investigating properties of a graph
- Key to solving this problem efficiently is learning how to build the graph itself!
Problem 4: Worm

Things to keep in mind:

- Algorithm **must** run in linear time with respect to the area of the graph.
- If you choose to use classes or structs in your graph definitions, defining operators such as ‘==’ or ‘<’ will be useful to make them compatible with STL data structures such as ‘std::map’.
- You may reuse code from previous problems!
Problem 5: Rotated Sorted Arrays

- Find an element in a sorted array that has been rotated
- Naive search checks every element -- runtime: $O(n)$
- Apply Divide-and-Conquer to find the element -- runtime: $O(\log n)$
  - similar approach as binary search
  - what cases can you think of for the location of the element relative to the rotation point?
    - start with a rotation of 0 (still in original sorted order) and work from there
    - verify that your algorithm is in fact finding the correct element
- Timing code has been provided for you in the tester
- Remember that you must include a PDF writeup with your timing results and description of your DQ algorithm.
Questions?