4.5 Profiling

Before I Forget!

Facebook Programming Competition:

- 5th March
- 2.5 hours, 5 problems
- C, C++, Java
- Quiet Labs
- Facebook recruiters, food and swag
- See you there! http://io.gd/2F

Profiling - Memory Usage

Agenda

1. Memory profiling
2. Graphs representation
3. The dot language
4. Graphs Traversal
5. Topological Sorting

Profiling - Memory Usage

Ever thought about how much memory your program uses?

Profiling - Memory Usage

- You could use

```
long used = Runtime.getRuntime().freeMemory()
// create some objects
used -= Runtime.getRuntime().freeMemory()
```

- handle garbage collection, JVM non-determinism …. → Check article

Profiling - Memory Usage

- The article:

Object	Size (Bytes)
java.lang.Object	
java.lang.Integer	
java.lang.String	

Profiling - Memory Usage

- The article:

Object	Size (Bytes)
java.lang.Object	8
java.lang.Integer	
java.lang.String	

Profiling - Memory Usage

- The article:

Object	Size (Bytes)
java.lang.Object	8
java.lang.Integer	16
java.lang.String	

Profiling - Memory Usage

- The article:

Object	**Size (Bytes)**
java.lang.Object	8
java.lang.Integer	16
java.lang.String	40

Profiling - Memory Usage

- The article:

Object	Size (Bytes)
java.lang.Object	8
java.lang.Integer	16
java.lang.String	40

- Abstraction has some serious overheads!

Profiling - Use A Profiler!

- Sun JDK comes with a profiler installed

- Usually some place like
    - /usr/lib/jvm/jdk-1.x.y/bin/jvisualvm

Profiling - Memory Optimization

- Often have to deal with repeated strings (e.g. parsing a backend report)

- How can we reduce memory usage?

Profiling - Memory Optimization

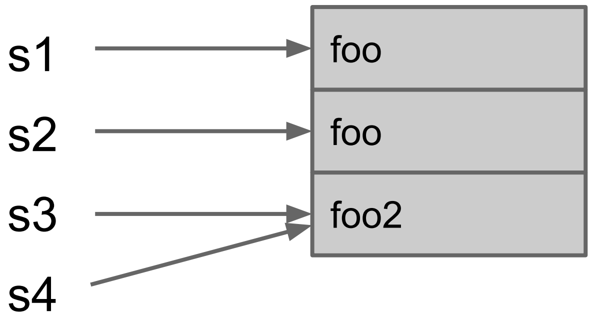
- Often have to deal with repeated strings (e.
  g. parsing a backend report)

- How can we reduce memory usage?

- String **interning!**

String Interning

```
String s1 = new String(“foo”);
String s2 = new String(“foo”);
String s3 = (new String(“foo2”)).intern();
String s3 = (new String(“foo2”)).intern();
```



5 Graphs

Representation

Adjacency Matrix

```
  0 1 0
  1 0 0
  0 1 0
int g[n][n];
O(n^2) memory
```

Adjacency List

0 -> 1

1 -> 0

2 -> 1

List<List<Integer>> g;

O(E) memory

Traversals

- BFS

```
public static void dfsIterative(int start) {
      Queue<Integer> queue = new Queue<>();
      queue.add(start); seen[start] = true;
      while (!queue.isEmpty()) {
          int node = queue.remove();
          for (int neighbour : nodes.get(node)) {
              if (seen[neighbour]) continue;
              seen[neighbour] = true;
              queue.add(neighbour);
}}}
```

Traversals

- Iterative DFS

```
public static void dfsIterative(int start) {
      Stack<Integer> stack = new Stack<>();
      stack.add(start); seen[start] = true;
      while (!stack.isEmpty()) {
          int node = stack.pop();
          for (int neighbour : nodes.get(node)) {
              if (seen[neighbour]) continue;
              seen[neighbour] = true;
              stack.push(neighbour);
}}}
```

Traversals

- Recursive DFS

```
public static void dfs(int start) throws Exception {
    seen[start] = true;
    for (int neighbour : nodes.get(start)) {
        if (!seen[neighbour]) {
            // do something useful and recurse
            dfs(neighbour);
        }
    }
}
```

Traversals

- Recursive DFS - don’t forget outer loop!

```
public static void dfs(int start) throws Exception  {...}

for (int i = 0; i < n; i++) {
    if (seen[i]) continue;
    dfs(i);
}
```

Traversals

- Recursive DFS - don’t forget outer loop!

```
public static void dfs(int start) throws Exception  {...}
```

```
for (int i = 0; i < n; i++) {
    if (seen[i]) continue;
    dfs(i);
}
```

Traverse all connected components!

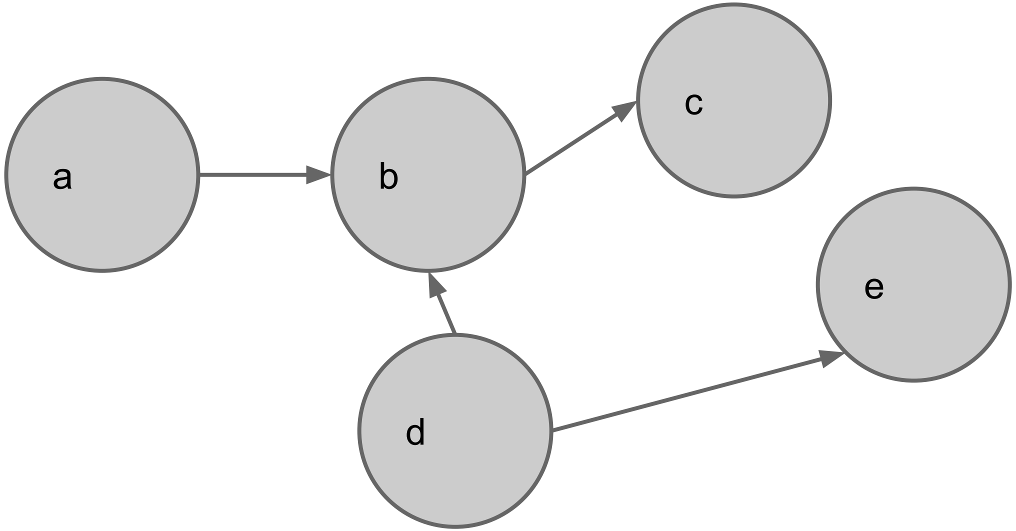
Topological Sorting

- If there is an edge a → b, then a is before b in the order

- “execution schedule” for dependent jobs

Topological Sorting

- "execution schedule" for dependent jobs



Topological Ordering:
a d b c e

Topological Sorting

1. Do a DFS from every unseen node

2. After a node has visited all neighbours:
   add it to the front of the order

3. Print the order

Topological Sorting

```
public static void dfs(int start) throws Exception {
     seen[start] = true;
     for (int neighbour : nodes.get(start)) {
         if (!seen[neighbour]) {
             // do something useful and recurse
             dfs(neighbour);
         }
     }
 }
```

Topological Sorting

```
public static void dfs(int start) throws Exception {
     seen[start] = true;
     for (int neighbour : nodes.get(start)) {
          if (!seen[neighbour]) {
               // do something useful and recurse
               dfs(neighbour);
          }
     }
     order.addFirst(start);
}
```

Minimum Spanning Tree

Given an undirected, connected graph (G), an MST is a minimum weight subgraph that contains all nodes of G.

MST - Kruskal

1. Start with each vertex as a separate tree

2. Select minimum weight edge

3. If it connects nodes from diff. trees **merge** them

4. Repeat until no more edges left

MST - Kruskal

1. Start with each vertex as a separate tree
2. Select minimum weight edge
3. If it connects nodes from diff. trees **merge** them
4. Repeat until no more edges left

Need something that can do **merging** and **finding** of/in sets fast: http://en.wikipedia.org/wiki/Disjoint-set_data_structure

MST - Kruskal

1. Start with each vertex as a separate tree
2. Select minimum weight edge
3. If it connects nodes from diff. trees **merge** them
4. Repeat until no more edges left

Need something that can do **merging** and **finding** of/in sets fast: http://en.wikipedia.org/wiki/Disjoint-set_data_structure

Not in Collections API :(

MST - Prim

1. Start with one node, add it to V

2. E = closest edge with exactly one endpoint in V

3. add E to MST

4. Repeat 2 until V contains all nodes of G

Single Source Shortest Path

Find shortest path from one node to all others.

Single Source Shortest Path

Dijkstra’s Algorithm:

```
1.  Start with source node (S)

2.  Select closest node to S (N)

3.  For each neighbour E of N:

    d(S, E) = min(d(S, E), d(S, N) + d(N, E))

4.  Repeat from 2 until no nodes left to check
```

Single Source Shortest Path

Dijkstra’s Algorithm:

```
1.  Start with source node (S)

2.  Select closest node to S (N)

3.  For each neighbour E of N:

    d(S, E) = min(d(S, E), d(S, N) + d(N, E))

4.  Repeat from 2 until no nodes left to check
```

Need something to give us min() fast!

Single Source Shortest Path

Dijkstra’s Algorithm:

```
1.  Start with source node (S)

2.  Select closest node to S (N)

3.  For each neighbour E of N:

    d(S, E) = min(d(S, E), d(S, N) + d(N, E))

4.  Repeat from 2 until no nodes left to check
```

Need something to give us min() fast!

Need fast updates!

Single Source Shortest Path

Dijkstra's Algorithm:

- Fast min ⇒ PriorityQueue!
- Fast Updates? Could technically do in O(log(n))

Single Source Shortest Path

Dijkstra's Algorithm:

- Fast min ⇒ PriorityQueue!
- Fast Updates? Could technically do in O(log(n))
    - JavaDoc: *this implementation provides O(log(n)) time for the enqueing and dequeing methods (offer, poll, remove() and add); l**inear time for the remove(Object) and contains(Object) methods**; and constant time for the retrieval methods (peek,element, and size).*

Single Source Shortest Path

Dijkstra's Algorithm:

- Obvious solution:
    - roll out our own

Single Source Shortest Path

Dijkstra's Algorithm:

- Obvious solution:
  - roll out our own
- Quick **hack** (generally good enough):
  - just leave nodes in the priority queue
  - keep track of what we've seen