Sarah MacAdam
Algorithms
Written 5
4/3/15

1

Problem Description
Show that a decision tree for comparison sort has exactly $n!$ leaf nodes each with probability $\frac{1}{n!}$ of being reached on any particular input. If there are more than $n!$ leaf nodes, then these nodes have a probability 0 of being reached.
Inputs
a list L with n elements
Outputs
int y: the number of permutations for list L's n elements
Assumptions
Strategy Overview
Show that the number of permutations is $n!$
Algorithm Description

//build the decision tree by adding one element at a time to each possible solution
//from the root node, create a path for every possible first element in the sorted list
//there are n possibilities
//for each of these n possibilities there are $(n-1)$ possibilities for the second element in the sorted list
//this is a total of $n(n-1)$ possibilities for length 2
//we repeat this pattern for every level in the decision tree so that it begins to look like this
$n(n-1)(n-2)(n-3)(n-4)...(n-n)$
//The permutations end at $(n-n)$, when all n elements have been added
//This series can be written as $n!$ and represents the total number of list arrangements of length n, or all of the leaf nodes
//Therefore, if there were an additional leaf node, it could not be a valid permutation because the tree has exhausted all possibilities. The possibility of reaching an impossible solution is 0.

//The probability of reaching any 1 valid permutation is equivalent for all of the $n!$ leaf nodes because each is unique.
//The probability of reaching one if these leaf nodes is therefore $\frac{1}{n!}$.

2

Problem Description
G is a graph with n nodes, where n is even. Prove or disprove that if every node in G has a degree of at least $\frac{n}{2}$, then G must be connected.
Inputs
graph G with n nodes
Outputs
boolean B: true if G is connected
Assumptions
n is even
Strategy Overview
Show that even when the nodes have the minimum possible degree, the graph must be connected. We can infer from there that if the nodes had a higher degree, the graph would still be connected.
Algorithm Description

//Take one node, P, in G and make it degree $\frac{n}{2}$, the minimum degree. P is now connected to $\frac{n}{2}$ other nodes. Therefore, $\frac{n}{2}+1$ nodes in G must be connected so far. Now take a node that is still unconnected. There are $\frac{n}{2}-1$ unconnected nodes remaining. Call the selected node Q. Q's degree must be $\frac{n}{2}$. However, not including Q, there are only $\frac{n}{2}-2$ unconnected nodes left. Therefore, Q must connect to nodes that are already connected to P.

We could connect Q to nodes already in the connected graph, and attempt to keep some of the unconnected nodes unconnected, but from this point on, there will never be $\frac{n}{2}$ nodes that are not already a part of the connected graph. The number of unconnected nodes will only continue to decrease.

Therefore, if all nodes have a degree of at least $\frac{n}{2}$, then the graph must be connected

3

Problem Description
Adapt Depth-First Search to detect the existence of cross-edges and descendant-edges and distinguish between the two.
Inputs
Graph G: a directed graph
Outputs
Set C: the set of cross edges
Set D: the set of descendant edges
Assumptions
Graph G is directed
Strategy Overview
Build on the depth first algorithm by adding a field to the nodes that records their discovery time. Discovery time will be when the node is processed, or when it turns gray.
Algorithm Description

```
//Create a stack S

1. Choose one node $u$ to begin.
2. Add $u$ to the stack
3. Pop the top node in the stack, $u$, for processing
4. If $u$ is white, make $u$ gray and record the time in $u.disc$
If $u$ is gray, proceed
If $u$ is black, it has already been processed, pop it off of the stack and return to step 3
5. Look at $u$'s adjacent nodes
6. Push all nodes adjacent to $u$, that are not black, onto the stack
7. Consider an adjacent black node, $v$
8. Compare $v.disc$ to $u.disc$
9. If $u.disc < v.disc$, $u$ was discovered first, add $Edge(u, v)$ to the set of descendant edges $D$
10. If $v.disc < u.disc$, $v$ was discovered first, add $Edge(u, v)$ to the set of cross edges $C$
11. Pop the node on top of the stack for processing
12. Repeat steps 4 to 11 until you reach a node $u$ with no adjacent edges
13. Make $u$ black and return to step 3
```

14. Continue until the stack is empty
15. If there are any remaining white nodes (in the case of an unconnected graph) select one and begin at step one again

4

Problem Description
Given a list of potential flights F, determine if your airline can service all of the flights with k number of planes.
Inputs
List $F = \{f_1, f_2, ..., f_n\}$: the list of flights that must be serviced, a flight f_n is an object with fields departure time, arrival time, starting airport, and ending airport
int k: the number of planes available
Outputs
bool s: true means all flights are serviceable
Assumptions
1. Plane maintenance takes m minutes
2. No plane can depart less than m minutes after arriving
3. A plane can fly between any two cities in l time to service multiple flights in F
4. All flights take place between 6am and 12 midnight
5. A plane can end at any airport
Strategy Overview
Create a graph where each edge is a flight in F. If an end time for one flight leaves $m + l$ time before the start of another flight at another airport then the two are compatible. If an end time for one flight leaves m time before the start of another flight at the same airport then the two are compatible. All compatible flights will be connected by a splice edge. All of the required flights in F will have a lower bound of 1. A super source node will connect to all starting airports and a super sink will connect to all ending airports. The super source will only be able to supply k units of flow. The capacity for all flight edges, including splice flights, will be 1. An additional edge will connect the source and sink with capacity k. Any surplus planes can flow through this edge. Ford-Fulkerson will be run on the graph. If all lower bounds are met then the flights are serviceable.

Algorithm Description

```
//Construct nodes and edges for every flight in $F$
//make their capacity 1 and their lower bound 1
//Loop through $F$

    //For every flight $f_n$, determine the end time

    //Calculate $f_n endtime + m + l = et$ and $f_n endtime + m = sa$

    //For every flight $f_e$ departing from a different airport with a start time later than $et$ create a splice edge from $f_n$ to $f_e$

    //For every flight $f_e$ departing from the same airport with a start time later than $sa$ create a splice edge from $f_n$ to $f_e$

    //make the capacity for the splice edge 1 and the lower bound 0

//Create a super source node and a super sink
//Create edges with capacity 1 from the source node to every starting airport and from every ending airport to the sink also with capacity 1.
//Create a new super source $source'$ that connects to the old, make the edge's capacity $k$
//Create one last edge from the original source to the sink with capacity $k$ for any surplus planes to travel on
//Run Ford-Fulkerson on the graph
//Check if all lower bounds were met for the flights in $F$.
//If a lower bound was not met, return $s = 0$ the flights are not serviceable with k planes.
//If they are met, return $s = 1$ the flights are serviceable
```

5

Problem Description
Create the minimal schedule for two robots to reach their destination in a graph without ever colliding or being adjacent to each other.
Inputs

Graph $G = (V, E)$: a directed graph
Nodes s_1, s_2: the start nodes for the two robots, $s_1, s_2 \in V$
Nodes d_1, d_2: the destinations for the two robots, $d_1, d_2 \in V$

Outputs

Set R: The set of moves in the minimal schedule

Assumptions

The start and end nodes are compatible, i.e. not adjacent or the same

Strategy Overview

Create a new graph that holds all compatible moves on the graph, starting with the two start nodes. Each node will be a tuple that represents the current positions of the two robots. A breadth first search on this new graph will find the shortest root to the destinations of both robots.

Algorithm Description

```
//Build the new graph of compatible tuples
//The root node will be the tuple $(s_1, s_2)$ the two start nodes
1. Determine the children of this node, generalized as $(a, b)$
    2. The set of tuples $(a, b_c)$ where $b_c$ is any child of $b$ not adjacent to $a$ and not $a$ itself
    3. The set of tuples $(a_c, b)$ where $a_c$ is any child of $b$ not adjacent to $b$ and not $b$ itself

// For all children of $(a, b)$ repeat steps 1,2, and 3 until all nodes in the original graph have been exhausted and there are no more children
//Run breadth first search on the new graph
//Create a queue Q
//Begin at the root node, push the root onto the back of the queue, mark it as gray
//Pop the node off the top of the queue for processing, make it black
1. Examine all children of the current node
2. If a node's tuple is $(d_1, d_2)$, the shortest path has been found
    //push the node onto a stack
    //find a black parent of the node and add it to the stack
    //repeat until you push the root node onto the stack
    //The set of moves is now saved on the stack
3. Else, add the node to the back of the queue and make it gray, do this for all children of the current node
4. Pop the next node off of the queue for processing, make it black
```

5. Repeat, If the destination (d_1, d_2)node is never found, there is no solution

Runtime

The runtime for a breadth first search is $\theta(V + E)$. In the worst case, the maximum number of compatible nodes creates the maximum number of nodes for the new graph. $\frac{n}{2}$ nodes will be in the robot 1's path from its start to destination and the other half will be in the path for robot 2. The two paths are unconnected. This will give $\frac{n^2}{4}$ pairings and therefore $\frac{n^2}{4}$ nodes for the new graph. If the individual graphs for robots 1 and 2 connect the start node to every other node in the path and have a linear path between the start and end node, then we have the greatest number of edges. There will be $\frac{n^2}{4} * (\frac{n}{2} - 1) * 2$ edges in this graph.$\frac{n}{2} - 1$ is the remaining number of nodes to connect to and 2 represents the graphs for each robot.
We can generalize the runtime as $\theta(\frac{n^R}{R^R} * (\frac{n}{R} - 1) * R)$ where n is the number of nodes in G and R is the number of robots.
As robots are added, the runtime will decrease.