Elizabeth Kukla

3 April 2015

# Written 5

**1)** The decision tree corresponds to each possible path taken by a given input size, n, to sort the list which means that for a given input input size, n, there are n! possible orderings of the set indices, and therefore n! sorting possibilities (leaf nodes). For any ordering of the input, it will follow exactly 1 path in the decision tree to a sorted leaf node. Therefore any input, n, has exactly $\frac{1}{n!}$ chance of reaching a particular leaf node. If a decision tree had more than n! leaf nodes they would either have to be duplicate paths or disconnected from the graph in which case there would be a zero probability of reaching the node.

**2)** G- graph with n nodes, assume n is even

If every node of G has a degree of at least $\frac{n}{2}$, G must be connected.

Connected graph: undirected graph, each vertex reachable from all others.

If every node has degree of at least $\frac{n}{2}$, the graph must be connected, because there is no way to create two disjoint subgraphs where each node is at least degree $\frac{n}{2}$. To achieve at least degree for each node in the graph, it must be connected, and there must be at least one node that has a degree greater than $\frac{n}{2}$.

**3)** Adapt depth-first search to register whenever it "finds" a black (completely processed) node. To distinguish between back edges and cross edges, the algorithm will need to note the start/finish time stamps for the black nodes found. If the time stamp for when the node was finished being processed is less than/before the start (discovery) time stamp of the current node, it is a cross edge. If the finished processing time stamp is after the discovery time of the current node, it is a descendant edge.

**4)**

**Input:**

set of flights F, integer K, integer m.

Each flight has the following attributes: arrival time, departure time, start city, end city.

K- maximum number of planes that can be used. m - mandatory maintenance time at each city.

**Mappings:**

Nodes=cities, edges = flight from city to city, Flow= flow of planes, Required edges= flights specified in the set F, Splice Edges = flights to move planes to required cities, but don't have to be used.

Required Edges have a lower flow bound of 1 and an upper flow bound of 1 (one and exactly one plane per edge)

Splice Edges have a lower bound of zero and an upper bound of 1 (you don't have to send a plane across the edge, but you can)

**Algorithm Description:**

Build basic graph with the required edges. Add all of the valid splice edges (validity based on constrains with departure, arrival, and maintenance time). From there you have a circulation problem (multiple starts/sinks), which you reduce to a max flow problem with a super-start and super-sink node.

**Algorithm Steps:**

**1.** Build basic graph From cities in Flight set. Fill in required edges.

**2.** Fill in valid splice edges

- splice edge from node v to u is valid if [v(departure) - u(arrival)] > [l + 2m] where l is the flight time from v to u and m is the required maintenance time on the ground at each node.
- once splice edges have been filled in, you have accounted for flight timing and don't need to worry about it anymore.

**3.** Reduce to a max flow problem by creating super-start S* and super-sink T*.

- your set of start nodes has k nodes and are the starting cities of K flights in F. connect them with S*. Do the same with your terminal nodes in T*.
- now you have a max flow problem.

**4.** See if you can satisfy all of the required edges with K items in your total flow (flow is movement of planes from city to city) and at most one plane should travel across ant required edge. You have at most K planes available to you. If you can, it is feasible to make the scheduling of routes work for the day.

**5)**

**Input:**

Graph with robots at start nodes. S= set of start nodes, D= set of destination nodes

**Algorithm Description:**

This is a variation of the state-space search problem where the initial state is the graph with robots at each of their start nodes and the goal state is the graph with each robot at their respective end node. The search for open states is governed by the constraints put upon the proximity of the robots. The scores for the best-search choice for choosing the best of the next open states are based on the number of moves required to reach the goal state.

Each robot has a list that keeps track of their route to their destination.

*** My algorithm assumes that multiple robots can move at the same time, i.e. one state change can result in one or all of the robots moving.

**Algorithm Steps:**

From start state described above

**WHILE** (open states)

**1.** current state = priority queue.next()

add current state to closed list

if a robot moves, add move to their route.

**2.** If state would make robots adjacent or sharing a node, ignore the state (it is not open)

**3.** If the state is in the closed list (already visited), ignore the state (it is not open)

**4.** Calculate score for open state (higher score means fewer total moves (for all robots) required to reach goal state). score = f(s) where f is the function that determines score and s is state.

**5.** add (state, score) to priority queue

**END WHILE**

By the end, if there is a solution (goal state is reachable with given constraints), you will be in your goal state and each robot will have a list of moves that comprise their complete route from their start node to their destination node.