Algorithms

Final Project

28 April, 2015

Juan Vallejo

CS 420 Algorithms

Dr. Koehl

Who's Got Game?

Discussion

PROBLEM ANALYSIS

This problem lists directed node-pairs that represent actions or items in a game. We are tasked with determining whether a given set of relationships between such actions or items leads to an unfeasible game, linear gameplay, or allow for a non-linear game.

We interpret the set of relationships given as a graph, each item or action as a node, and the relationship between a pair of nodes as an edge. Non-linear gameplay translates to a disconnected, directed graph. Linear gameplay is a strongly connected graph. And an unfeasible game is given by any graph containing cycles.

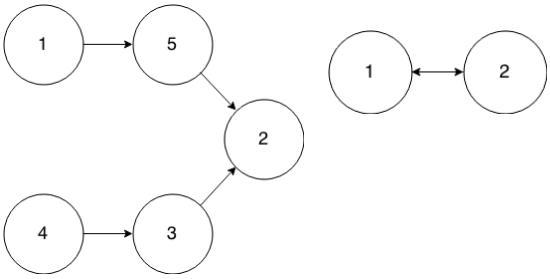

Figure 1.1 Disconnected graph (Input 1) and strongly-connected cyclic graph (Input 2).

In the sample inputs provided, we are given three directed graphs. They do not contain negative weight edges nor cycles. The first input is an acyclic, disconnected graph and leads to non-linear gameplay. The second input is a strongly connected, directed, acyclic graph and leads to linear gameplay. The graph has Eulerian and Hamiltonian paths. The third input is a strongly connected cyclic graph and leads to an unfeasible game.

APPROACH TO SOLVING

Cycle detection takes into account nodes that are strongly connected, as well as Eulerian and Hamiltonian cycles.

To solve this problem, we implement a depth-first search algorithm. This algorithm will use a stack to keep track of traversed nodes. We first address the issue where we might be faced with different sets of nodes in our graph, and the possibility of any of these nodes containing cycles, we use an adjacency list. Such list consists of key-pair values. The list index indicates our node's value, and the value from the key-value pair indicates, through a boolean, if that node has been visited yet. As we traverse all of the children of an initial node, we detect cycles by checking our adjacency list for any nodes we have already marked as visited. We do this for any remaining, disconnected nodes in our list as well. We detect linearity by making sure no nodes remain unvisited after traversing all of the children from our initial node. Finally, we find non-linear gameplay if no cycles are found, and unvisited nodes remain in our list.

Code

ASYMPTOTIC COMPLEXITY

Our programmatic implementation uses recursion, calling our `isCyclic()` function n times. Furthermore, each time we add a pair of nodes to the graph, we are iterating through each of the parent node's children another n times. Since our `isLinear()` function is called after our `isCyclic()` function, we are able to use memorization by using pre-calculated values, giving us linear performance. Our total runtime for this problem is $O(2n)$.

```
/**
 * Implementation of the problem above.
 * Note: to run this file, simply open a
 * new tab in Chrome, open the developer
 * console (F12 in Windows), and paste
 * this code into it. Press Enter. Magic.
 * Tip: For non-manual copy/pasting wizardry
 * find this code at the url below.
 *
 * @url https://gist.github.com/juanvallejo/4df80cefada564aa1de7
 * @file main.js
 * @author juanvallejo
 */

var total      = 5; // total number of nodes in sequence
var traversed = 0;
var visited   = []; // stack of nodes visited

function Node(value) {

	this.value          = value;
	this.children       = [];

	this.containsChild = function(childValue) {

		var childExists = false;
		for(var i = 0; i < this.children.length; i++) {
			if(childValue == this.children[i].value) {
				childExists = true;
				break;
			}
		}
		return childExists;

	}
}

function Graph() {

	this.root  = null;
	this.nodes = {};

	this.addNodes = function(pVal, cVal) {

		this.nodes[pVal] = this.nodes[pVal] || new Node(pVal);
		this.nodes[cVal]    = this.nodes[cVal] || new Node(cVal);

		if(!this.root) this.root = this.nodes[pVal];

		if(!this.nodes[pVal].containsChild(cVal)) {
			this.nodes[pVal].children.push(this.nodes[cVal]);
		}
	}
}

function isCyclic(node) {

	if(visited.indexOf(node) != -1) return false;
```

```
	traversed++;

	visited.push(node);

	if(!node.children.length) {
		return false;
	}

	var cyclic = false;

	for(var i = 0; i < node.children.length; i++) {
		if(visited.indexOf(node.children[i]) == -1) {
			cyclic =  isCyclic(node.children[i]);
		} else {
			cyclic = true;
		}
	}

	return cyclic;
}

function isLinear() {
	return total == traversed;
}

// add your test-case here
var graph = new Graph();
graph.addNodes(1, 5);
graph.addNodes(5, 2);
graph.addNodes(3, 2);
graph.addNodes(4, 3);

if(isCyclic(graph.root)) {
	console.log('Infeasible game.');
} else if(isLinear()) {
	console.log('Linear gameplay.');
} else {
	console.log('Nonlinear gameplay possible.');
}
```

Fun, Fun, Fun "Auf Der Autobahn"

Discussion

PROBLEM ANALYSIS

This problem lists a map with back-roads and autobahn segments. Our task is to find the distance between any two given locations on such map that takes the least amount of time, but also prefers to use autobahn segments whenever possible.

We interpret the given map as a graph and each location as a node. Roads represent edges between nodes and are classified as either a back-road or an autobahn segment ("b" or "a"). Since edge length as well as speed are factors in determining the fastest route to a destination node, we compare the "length" of each edge as a ratio between its kilometer length and speed at which it can be traveled through.

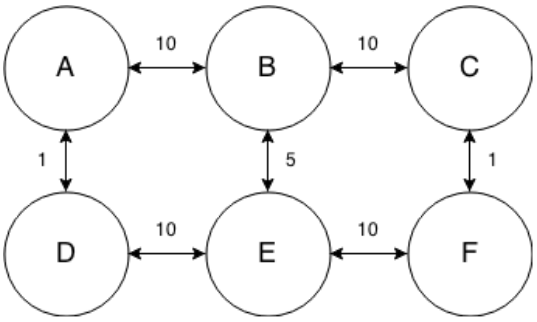

Figure 2.1 Strongly connected nodes in a graph.

In the sample input provided, we are given a cyclic, connected graph. It does not contain negative weight edges nor cycles. Because we are given the assumption of bi-directional streets, the graph is undirected. The graph contains a Hamiltonian path and cycle as we can visit each vertex in the graph exactly once while ending at the starting node. It also contains a Euler path, but not cycle, as we are able to traverse each edge once, but not end at the starting edge.

APPROACH TO SOLVING

All streets on the map are bidirectional. This is addressed in our implementation as a graph consisting of strongly connected nodes. Because only one edge can exist between any two nodes on our graph, Dijkstra's algorithm can be feasibly implemented. Since shortest path algorithms cannot have cycles, we treat our graph as "directed" and "acyclic" by always going from root node to child node, and ignoring already visited nodes.

This problem implements Dijkstra's algorithm to find the shortest path. While Bellman-Ford is also a viable option, we do not care for negative-weight cycle detection, and are fine running through our graph only once. Dijkstra's algorithm could be seen as a form of breadth-first search that also takes into account edge-weights. Because of this, a non-recursive method is used. A queue holds all nodes "pending" to be traversed, and a hash map holds all distances between each node and the root node. Dijkstra's algorithm is "greedy"; to address this, our local optimizations consist of looking for a child node with the shortest tentative distance to the current node.

Code

ASYMPTOTIC COMPLEXITY

Our programmatic implementation is queue-based. It has a main loop that runs n times, while there are items in the queue. Inside this loop, we have one more that iterates through the children of the current queued node, relaxing their tentative distances. We also have another one that looks through the node's

children for unvisited, un-enqueued nodes, and one more that pushes these singled-out nodes to the queue. The complexity of our algorithm is now $O(4n^2)$.

```
/**
 * Implementation of the problem above.
 * Note: to run this file, simply open a
 * new tab in Chrome, open the developer
 * console (F12 in Windows), and paste
 * this code into it. Press Enter. Magic.
 * Tip: For non-manual copy/pasting wizardry
 * find this code at the url below.
 *
 * @url https://gist.github.com/juanvallejo/1fe342b02d1460d5e1bd
 * @file main.js
 * @author juanvallejo
 */

function sortArray(array) {

	var copy     = [];

	var arrays = {
		order:[],
		sort : null
	}

	for(var i = 0; i < array.length; i++) {
		copy[i] = array[i];
	}

	arrays.sort  = _sortArray(copy, []);
	arrays.order = [];

	for(var i = 0; i < arrays.sort.length; i++) {
		arrays.order.push(arrays.sort.indexOf(array[i]));
	}

	return arrays;

}

function _sortArray(array, sortedArray) {

	if(!array.length) {
		return sortedArray;
	}

	var smallest = array[0];
	var smallestIndex = 0;

	for(var i = 0; i < array.length; i++) {
		if(array[i] < smallest) {
			smallest = array[i];
			smallestIndex = i;
```

```
                }
        }

        array.splice(smallestIndex, 1);
        sortedArray.push(smallest);

        return _sortArray(array, sortedArray);
}

function Node(value) {

        this.value          = value;
        this.visited  = false;
        this.rootDistance = null;

        this.children       = [];
        this.distances      = [0, 0];

        this.containsChild = function(childValue) {

                var childExists = false;
                for(var i = 0; i < this.children.length; i++) {
                        if(childValue == this.children[i].value) {
                                childExists = true;
                                break;
                        }
                }
                return childExists;

        }
}

function Edge(nodeA, nodeB, length, type) {

        this.nodeA          = nodeA;
        this.nodeB          = nodeB;
        this.length         = length;
        this.type           = type;

}

function Graph() {

        this.root  = null;
        this.nodes = {};
        this.edges = {};

        this.getEdge = function(pVal, cVal) {
                return this.edges[pVal + cVal] || this.edges[cVal + pVal];
        }

        this.addEdge = function(pVal, cVal, edgeLength, type) {

                this.nodes[pVal] = this.nodes[pVal] || new Node(pVal);
                this.nodes[cVal] = this.nodes[cVal] || new Node(cVal);
                this.edges[pVal + cVal] = this.edges[pVal + cVal] ||
                        new Edge(this.nodes[pVal], this.nodes[cVal], edgeLength, type);
```

```
		if(!this.nodes[pVal].containsChild(cVal)) {
			this.nodes[pVal].children.push(this.nodes[cVal]);
		}

		if(!this.nodes[cVal].containsChild(pVal)) {
			this.nodes[cVal].children.push(this.nodes[pVal]);
		}

	}

	this.getSmallest = function(list) {

		var small = this.nodes[list[0]];
		for(var i = 0; i < list.length; i++) {
			if(this.nodes[list[i]].rootDistance < small.rootDistance) {
				small = this.nodes[list[i]];
			}
		}
		return small;

	}

	this.getFastestRoute = function(nodeAValue, nodeBValue) {

		var source = this.nodes[nodeAValue];
		var target = this.nodes[nodeBValue];

		var queue    = [];
		var visited = {};
		var targetFound = false;

		queue.push(source.value);
		this.nodes[queue[0]].visited            = true;
		this.nodes[queue[0]].rootDistance       = 0;

		while(queue.length) {

			var current = this.getSmallest(queue).value || queue[0];

			visited[current]                  = true;
			this.nodes[current].visited       = true;

			var distances = [];
			var sortOrder = [];

			for(var i = 0; i < this.nodes[current].children.length; i++) {

				if(!visited[this.nodes[current].children[i].value]) {

					var edgeLength = this.getEdge(current,
this.nodes[current].children[i].value).length;

					if(this.getEdge(current,
this.nodes[current].children[i].value).type == 'a') {
						edgeLength /= 160;
					} else {
						edgeLength /= 80;
					}
```

```
                                    if(this.nodes[current].children[i].rootDistance ==
null || this.nodes[current].rootDistance + edgeLength <
this.nodes[current].children[i].rootDistance) {

                                          this.nodes[current].children[i].rootDistance =
this.nodes[current].rootDistance + edgeLength;
                                          this.nodes[current].children[i].distances[0] =
this.nodes[current].distances[0] + this.getEdge(current,
this.nodes[current].children[i].value).length;

                                          if(this.getEdge(current,
this.nodes[current].children[i].value).type == 'a') {
                                                // console.log('updating autobahn
distance for ' + this.nodes[current].children[i].value + ' to ' +
(this.nodes[current].distances[1] + this.getEdge(current,
this.nodes[current].children[i].value).length));

this.nodes[current].children[i].distances[1] = this.nodes[current].distances[1] +
this.getEdge(current, this.nodes[current].children[i].value).length;
                                          } else {
                                                if(this.nodes[current].distances[1] >
this.nodes[current].children[i].distances[1]) {

this.nodes[current].children[i].distances[1] = this.nodes[current].distances[1];
                                                }
                                          }

                                    }


if(queue.indexOf(this.nodes[current].children[i].value) == -1) {

distances.push(this.nodes[current].children[i].rootDistance);
                                    }

                              }

                        }

                        sortOrder = sortArray(distances).order;

                        // pointers to unvisited & un-added nodes
                        var unvisited = [];

                        for(var i = 0; i < this.nodes[current].children.length; i++) {
                              if(!visited[this.nodes[current].children[i].value] &&
queue.indexOf(this.nodes[current].children[i].value) == -1) {
                                    unvisited.push(i);
                              }
                        }

                        for(var i = 0; i < unvisited.length; i++) {

queue.push(this.nodes[current].children[unvisited[sortOrder[i]]].value);
                        }

                        // remove current item from array
```

```
                        queue.splice(queue.indexOf(current), 1);

                }

                return {
                        source              : this.nodes[nodeAValue],
                        target              : this.nodes[nodeBValue],
                        autobahn     : this.nodes[nodeBValue].distances[1],
                        distance     : this.nodes[nodeBValue].distances[0],
                };

        }

}

// add your test-case here
var graph1 = new Graph();
graph1.addEdge('A', 'B', 10, 'a');
graph1.addEdge('B', 'C', 10, 'a');
graph1.addEdge('D', 'A', 1, 'b');
graph1.addEdge('E', 'B', 5, 'b');
graph1.addEdge('F', 'C', 1, 'b');
graph1.addEdge('D', 'E', 10, 'b');
graph1.addEdge('E', 'F', 10, 'b');

var route = graph1.getFastestRoute('F', 'D');
console.log(route.source.value + ' ' + route.target.value + ' ' + route.distance + ' '
+ route.autobahn);
```