Logical Model of Graphing Algorithms

We used the MIT logical modeling software *alloy* to model Prim's, Kruskal's, and Bellman-Ford algorithms.

Modeling a Graph Algorithm

We attempted several different graph models, the variation between them coming from how they represented edges. At first, our desire to show *weighted* edges seemed to necessitate that we make an `edge` sig with a `weight` attribute. However, after some trial we settled on `Node -> Int -> Node` which allows us to model connectedness and visualizes well.

Prim and Kruskal

Prim's forced us to model adjacency, connectedness, and how to create a `Source` node. Kruskal's forced us to further consider connectedness. We needed a way to test if a graph was cyclic, which initially seemed impossible for weighted edges. The solution turned out to be a function `getUnweightedEdges`, which taught us of the strange implicit return values in alloy (see the function declaration for more details). With this tool, along with the function `getWeight` allowed us to properly reason about cycles.

We compare the two in the file *prim-kruskal-test.als*. This comparison led us to create a shared `Node` definition between the two MST algorithms. However, we wanted them to have different `State` sigs so that we could see how each generated an MST starting from the same input. `nonUniqueMST` shows the conditions under which Prim's and Kruskal's algorithms might find different results. The crucial step is in finding some potential edge that can be added to an MST to make a cycle and swapped for another edge in the cycle. See the `nonUniqueMST` definition for more information.

We chose to implement Prim's and Kruskal's algorithms on connected graphs. Though they could have been implemented to make minimum spanning *forests* instead of trees, we were more focused on MSTs and this abstraction made reasoning about MSTs simpler.

Dijkstra and Bellman-Ford

To view these shortest path algorithms, we recommend that in addition to projecting over `State` and `Event` to also show `distance` and `infinite` sigs as attributes rather than arcs.

We decided to model Dijkstra's model with an `infinity` relation to shows which nodes have an infinite distance variable. Though this required our model to be slightly more complicated, it results in a more scalable model since the our representation of "inifinity" need not change with a greater integer bit-width.

In addition to the sigs and relations from Dijkstra, Bellman-Ford made us create `anotherStepPossible` predicate. After run for |N|^2 `States` , `anotherStepPossible` implies a negative cycle. When Dijkstra's is run on the same input it simply generates an incorrect tree instead of noting this. Unfortunately, the numerous steps required for Bellman-Ford means that it is extremely slow, and is best run on two nodes, but these are enough to see its functionality. Since this was our reach goal, we are proud of what we were able to accomplish with Bellman-Ford.