Programming Project 2 - Dijkstra's Algorithm

Please submit your solution via email to the instructor with CC to ly603@nyu.edu. The deadline for the project is May 18.

The goal of this project is to implement and (partially) verify Dijkstra's algorithm for computing the single-source shortest paths to all vertices in a directed graph.

Part 1 Implementing and Verifying a Heap-Based Priority Queue

Dijkstra's algorithm uses a priority queue for efficient exploration of the vertices in the graph. In the first part of this project you will implement and verify a heap-based priority queue.
Your priority queue should have the following signature:

```
class PriorityQueue<T(==)>
{
 constructor Init(N: nat);
 method insert(t: T, k: int);
 function method min(): T
 method deleteMin();
 method decreaseKey(t: T, k: int);
 function method isEmpty(): bool;
}
```

The priority queue should maintain a set of T values that are ordered by integer keys. There should always be at most one key/value pair for any T value in the queue. The constructor takes the initial capacity as argument. The `decreaseKey` operations sets the key for the given value `t` to `k`. It may assume that `k` is not larger than the old key associated with `t`. The semantics of the remaining operations is as expected.

(a) Implement the priority queue operations. Make sure that all operations have the expected worst-case running times of a heap-based implementation. In particular, `insert`, `deleteMin`, and `decreaseKey` should all run in time $\mathcal{O}(\log(n))$, where n is the size of the queue. The operations `min` and `isEmpty` should run in constant time.

(b) Add a dynamic frame to your priority queue implementations and add an invariant that ties it to the representation of the queue. Use Dafny to check that all queue operations satisfy there modifies clauses and preserve the representation invariant.

(c) Add contracts to all operations to specify their functional behavior. You may introduce additional ghost fields as you see fit. Use Dafny to verify that all contracts are satisfied by your implementation. Add additional representation invariants to your implementation if necessary.

Hint: The Boogie source code distribution contains a partial heap-based priority queue implementation. See `Test/dafny1/PriorityQueue.dfy`. This implementation only stores keys instead of key/value pairs but you can use it as a starting point for your own implementation.

Part 2 Implementing and Verifying Dijkstra's Algorithm

In the second part of this project you will implement Dijkstra's Algorithm in Dafny and verify it against your priority queue implementation.

(a) Use the class `Graph` provided on the course web site and your priority queue implementation from Part 1 to implement a method

```
method shortestPaths(G: Graph, source : int)
returns (prev: array<int>)
  requires G != null && G.Valid;
  requires G.hasVertex(source);
```

The method should use Dijkstra's algorithm to compute an array `prev`. The array `prev` encodes (in inverse order) the shortest paths from the vertex `source` in the graph `G` to all other vertices in the graph. More precisely, `prev` maps a vertex `i` to its immediate predecessor on the shortest path from `source` to `i` if the path exists, and `-1` otherwise.

An object `G` of class `Graph` represents a directed graph with vertices `0` to `G.size-1` and edges encoded as adjacency lists for each vertex (see field `neighbors`). In your implementation you may assume that all edges in `G` have weight 1, i.e., the shortest distance between two vertices u and v is the infimum of the lengths of all paths from u to v in `G`.

(b) Use Dafny to verify that your implementation always terminates and that it satisfies all the contracts of the priority queue operations. Add loop invariants and decrease expressions to your implementation as you see fit. You do not need to verify that your implementation satisfies the specification of Dijkstra's algorithm, i.e., that it actually computes the shortest paths.

Hint 1: To make your life easier, avoid adding additional state to the objects of class `Graph`, e.g., by extending the class with additional mutable fields such as arrays. Keep all additional state local to the method `shortestPaths`.

Hint 2: You can start with Part 2 before you have finished Part 1. To do this, you need to provide an axiomatic interface specification of the priority queue. For example, an axiomatic interface specification of a set data structure is shown in Figure 1. Note that in the interface specification, the method `isEmpty` itself is used to denote the return value in the postcondition that defines the behavior of `isEmpty`. To tie the knot, once you are done with Part 1 you need to make sure that your implementation of the priority queue actually satisfies the axiomatic specification that you used in Part 2.

```
class Set<T(==)>
{
  ghost var Repr: set<objects>;
  ghost var Contents: set<T>;

  predicate Valid { this in Repr };

  ...

  method insert(x: T)
    requires Valid;
    modifies Repr;
    ensures Valid;
    ensures fresh(Repr - old(Repr));
    ensures Contents = old(Contents) + {x};

  function method isEmpty(x: T): bool
    requires Valid;
    reads Repr;
    ensures isEmpty() <==> (Contents == {});
}
```

Figure 1: Axiomatic interface specification of a set ADT

Part 3 Stretch Goals

If you are up for a challenge you can try to achieve one or both of the following stretch goals.

(a) For better performance, Dijkstra's algorithm is usually implemented using a priority queue based on Fibonacci heaps. Implement and verify such a priority queue.

(b) Verify that your implementation of Dijkstra's algorithm really computes the shortest paths to all vertices in the given graph. You can look at the implementation of the Shorr-Waite graph marking algorithm provided with the Boogie distribution to get some inspiration how to do this. Though, verifying Dijkstra's algorithm is more difficult and you might hit the limitations of what you can prove with the automated theorem prover underlying Dafny.