Priority Queues, Dijkstra's Shortest Path

The goal of this project is to specify, implement, and prove an algorithm, originally attributed to Edsger Dijkstra, for finding the shortest path between two nodes in a weighted graph. This is a famous graph algorithm and there is a lot of information available in books and on the web, including a detailed Wikipedia page. An auxiliary component needed is a module implementing priority queues, and thus the first part of the project is to specify, implement, and prove such a module.

You are expected to perform this project using Why3 version 0.71, using automated provers (typically Alt-Ergo, CVC3, Z3) and, if you like, Coq. The project must be done individually (no team work allowed). You must send by e-mail to `Claude.Marche@inria.fr`, `Guillaume.Melquiond@inria.fr`, no later than **Monday March 5, 2012 at 12:00 UTC**, an archive (`zip` or `tar.gz`) containing:

- A *`name`*`.mlw` source file containing your specification and implementation, where you replace *name* with your family name.
- The directory *name* generated by Why3, containing the proof session file `why3session.mlw` and Coq proof scripts `.v` if any.
- A document *`name`*`.pdf` in PDF format reporting on your work, where you should detail your approach to the problem, the choices you made to design the specifications, the implementation, etc. The structure of the document should follow the questions that are asked below.

A note about the evaluation of the project: there are several tasks to perform, and fully completing the given tasks is likely to be difficult. The evaluation will consider the quality of the work in priority, instead of the quantity. In other words, it is better not to answer every questions but doing it well, than answer everything but badly. Notice also that the quality of the PDF report counts at least for the half of the final evaluation.

1 Priority Queues

A priority queue for elements of some given type τ is a data structure offering these operations:

- creating an empty queue:

  create () : queue τ

- adding an element x in the queue, with some priority p (an integer):

  insert(q:queue τ, x:τ, p:int) : unit

- extracting from a non-empty queue an element of *minimal* priority

  extract (q:queue τ) : (τ,int)

A short program that illustrates the behavior of a priority queue is as follows:

```
let q = create () in
insert(q,a,12);
insert(q,b,23);
let x,p = extract(q) in
assert x=a ∧ p=12;
insert(q,c,7);
let x,p = extract(q) in
assert x=c ∧ p=7;
insert(q,d,42);
let x,p = extract(q) in
assert x=b ∧ p=23;
let x,p = extract(q) in
assert x=d ∧ p=42
```

The goal of this first section is to provide appropriate contracts for these operations, and then implement a procedure for sorting an array using queues.

1.1 Contracts for queue operations

1. *Specify a model for priority queues, under the form of a logic function* elements *denoting in some appropriate way the set of pairs (element,priority) stored in a queue.*

2. *Specify the three operations needed by Why3 functions with appropriate contracts (but without any body).*

3. *Implement the small test program above and prove the assertions.*

1.2 Sorting using Priority Queues

Using priority queues, there is a simple algorithm to sort an array a of integers, of size n. Here is a pseudo-code:

```
q := create ();
for i = 0 to n - 1 do insert(q,a[i]) done;
for i = 0 to n - 1 do a[i] := extract q done;
```

4. *Specify, implement, and prove a Why3 function for sorting an array, following this algorithm.*

2 Implementation of Priority Queues by Skew Heaps

The goal of this section is to implement priority queues using a data structure known as *skew heaps*, which are binary trees of some particular form.

Binary trees, holding at each node a data of type τ and a priority, could be defined as a sum type

type tree τ = Leaf | Node(tree τ, τ, int, tree τ)

A *heap* is by definition a binary tree where the value of the priority at any node is smaller or equal to the values in the sub-trees. A *skew heap* is a heap where insertion and removal of elements are done using the following pseudo-code, using an auxiliary function merge:

merge(a,Leaf) = a
merge(Leaf,b) = b
merge(Node(l_a,x_a,p_a,r_a),Node(l_b,x_b,p_b,r_b)) =
  if p_a <= p_b then Node(r_a,x_a,p_a,merge(l_a,b)) else Node(r_b,x_b,p_b,merge(l_b,a))

insert(a,x,p): a := merge(a,Node(Leaf,x,p,Leaf))

extract(a):
  case a = Leaf : error
  case a = Node(l,x,p,r) : a := merge(l,r); return (x,p)

It can be shown that skew heaps have good complexity properties. However only the soundness is our concern in this project.

5. *Provide Why3 implementations for each of the 3 expected operations for queues.*

6. *Prove that the implementations satisfy the contracts given in previous section.*

3 Shortest Path between two Nodes in a Graph

3.1 Weighted Graphs

A weighted graph is defined by

- A finite set of objects called *nodes* ;
- A set of directed *arcs*. Each arc relates two nodes and is labeled with a positive integer called its *weight*.

We assume there is at most one arc relating two given nodes, and that no arcs relate a node to itself.

A *path* from a node n to a node n' in the graph is a finite sequence of nodes $n_0, \ldots, n_k$ such that $n_0 = n$, $n_k = n'$ and for each i ($0 \leq i < k$) there is an arc from n_i to n_{i+1} of some weight w_i. The weight of the path is then $w_0 + \cdots + w_{k-1}$. A *shortest path* from n to n' is a path of minimal weight: there are no path of smaller weight between these nodes.

7. *Design a formal model of weighted graphs under the form of a theory in Why3. Discuss the various possible representations of arcs and explain the advantages of the one you choose.*

8. *Specify a predicate $path(n, n', w)$ meaning that there is a path from n to n' of weight w.*

9. *Specify a predicate $spath(n, n', w)$ meaning that the weight of the* shortest *path from n to n', if any, is w.*

3.2 Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path in a graph from a $source$ node to a $target$ node. A pseudo-code for a variant of this algorithm is as follows, where the returned value is the weight of the path. It also raises an exception "no path" if there are no path from $source$ to $target$.

```
dijkstra (source:node) (target:node) : int
  visited = empty set of node
  tovisit = empty priority queue
  insert source into tovisit with priority 0
  while tovisit is not empty do:
    extract (n, d) from tovisit
    if n = target then return d
    add n to visited;
    for each arc from node n to some node n' with some weight w:
      if n' not in visited then add n' in tovisit with priority d + w
  done;
  raise "no path"
```

10. *Implement the algorithm in Why3.*

11. *Specify a postcondition stating that if it returns some value w, then there is at least one path from source to target of weight w. Prove that the postcondition is verified by the implementation. A detailed explanation about the loop invariant used is expected.*

12. *Specify a postcondition stating that if it raises the exception, then there are no path at all from the source to the target. Prove that the postcondition is verified by the implementation. A detailed explanation about the loop invariant used is expected.*

13. *Specify a postcondition stating that if it returns some value w, then w is indeed the weight of the shortest path. Prove that the postcondition is verified by the implementation. A detailed explanation about the loop invariant used is expected.*