abridgedlab — MkAStarCore.sml

```
functor MkAStarCore(structure Table : TABLE
                    structure PQ : PRIORITY_QUEUE
                      where type Key.t = real) : ASTAR =
struct
  structure Set = Table.Set
  structure Seq = Set.Seq

  type weight = real
  type vertex = Set.key
  type edge = vertex * vertex * weight
  type heuristic = vertex -> real

  type graph = weight Table.table Table.table

  fun makeGraph (edges : edge Seq.seq) : graph =
      let
        open Table
        (* Makes sure to include vertices without outgoing edges *)
        val forward = collect (Seq.map (fn (u,v,w) => (u,(v,w))) edges)
        val backward = fromSeq (Seq.map (fn (_,v,_) => (v, Seq.empty ()))
            ↪ edges)
        val seqTable = merge #1 (forward, backward)
      in
        map fromSeq seqTable
      end

  fun findPath h G (S, T) =
      let
        (* Q is a PQ of (dist(v)+h(v), (v, dist(v))) *)
        fun findPath' X Q =
            case PQ.deleteMin Q
              of (NONE, _) => (NONE, Set.size X)
               | (SOME (_, (v, dist)), Q') =>
                  if Set.find T v then
                    (SOME (v, dist), Set.size X + 1)
                  else if Set.find X v then
                    findPath' X Q'
                  else let
                    val X' = Set.insert v X
                    fun relax (q, (u, w)) =
                        PQ.insert (dist + w + h(u), (u, dist + w)) q
                    val Q'' = Table.iter relax Q' (valOf (Table.find G v))
                  in
                    findPath' X' Q''
                  end

        (* Build initial queue from sources *)
        val init = Seq.map (fn v => PQ.singleton (h(v), (v, 0.0)))
        val Q = Seq.reduce PQ.meld (PQ.empty ()) (init (Set.toSeq S))
      in
        findPath' (Set.empty ()) Q
      end
end
```

abridgedlab — MkBridges.sml

```
functor MkBridges(structure STSeq : ST_SEQUENCE) : BRIDGES =
struct
  structure Seq = STSeq.Seq
  open Seq

  type vertex = int
  type edge = vertex * vertex
  type edges = edge seq

  type ugraph = vertex seq seq

  fun makeGraph (e : edges) : ugraph =
      let
        (* Max label is |V|-1 *)
        val n = 1 + reduce Int.max 0 (map Int.max e)

        (* Duplicate edges in both directions *)
        val dup = map (fn (u,v) => %[(u,v),(v,u)]) e

        val updates = collect Int.compare (flatten dup)
      in inject updates (tabulate (fn _ => empty ()) n)
      end

  fun findBridges (g : ugraph) : edges =
      let
        val n = length g
        fun N(u) = nth g u
        fun visited X v = isSome (STSeq.nth X v)

        (* dfs p ((B, X, c, m), u)
         *
         *  p : vertex - parent of current vertex in dfs search tree
         *  u : vertex - current vertex being searched
         *
         *  -----STATE-----
         *  B : edge list - accumulate bridges
         *  X : int option stseq - stores dfs search order
         *  c : int - incrementing counter for dfs search order
         *  m : int - minimum vertex touched in dfs subtree
         *)
        fun dfs p ((B, X, c, m), u) =
            if visited X u then
              (B, X, c, Int.min (m, valOf (STSeq.nth X u)))
            else let
              val X' = STSeq.update (u, SOME c) X

              (* don't touch the parent vertex p *)
              val toVisit = filter (fn v => v <> p) (N(u))
              val (B', X'', c', m') = iter (dfs u) (B, X', c+1, n) toVisit

              (* if the lowest numbered vertex reachable from the dfs search
               * tree rooted at u is >= u, then (p, u) is a bridge.
               *)
              val B'' = if p <> u andalso m' >= c then (p, u)::B' else B'
            in (B'', X'', c', Int.min (m, m'))
            end

        val V = tabulate (fn i => i) n
        val X = STSeq.fromSeq (tabulate (fn _ => NONE) n)
        val (B, _, _, _) = iter (fn (S, v) => dfs v (S, v)) ([], X, 0, 0) V
      in fromList B
      end
end
```