```
1  function [ costs, costGradientDirection, path ] = dijkstra( map, goalIdx,
   parameters, startIdx )
2  % DIJKSTRA Dijkstra's algorithm
3  % Returns a distance map as well as its gradient. An explicit path is
4  % optional. If the path is requested, the startIdx has to be passed in.
5
6
7  % Dilate the image by the robot radius so the output values respect the
8  % robot base size.
9  distInput = map;
10 distInput(distInput ~= 0) = 1;
11 distanceMap = bwdist(distInput);
12 dilatedMap = distanceMap;
13 dilatedMap(distanceMap <= parameters.robotRadius) = -1;
14
15 % Check inputs
16 if nargout < 3
17     inputsValid = checkInputs(map, dilatedMap, goalIdx, parameters);
18 else
19     inputsValid = checkInputs(map, dilatedMap, goalIdx, parameters, startIdx);
20 end
21 if ~inputsValid
22     err = MException('Dijkstra:InvalidInputs', ...
23                      'Dijkstra algorithm got invalid inputs');
24     throw(err);
25 end
26
27 % Cost field
28 % Cells with value inf were never visited
29 costs = inf*ones(size(dilatedMap));
30 visited = 0*ones(size(dilatedMap));  % keep track of which nodes were already
   expanded
31 isOnHeap = 0*ones(size(dilatedMap)); % keep track of which nodes are currently
   on the heap
32
33 mapSize = size(dilatedMap);
34
35 % We compute the distance to the goal at every cell in the map. We start
36 % from the goal.
37 costs(goalIdx(1), goalIdx(2)) = goalIdx;
38 firstNode.f = costs(goalIdx(1), goalIdx(2));
39 firstNode.idx = goalIdx;
40 heap = {firstNode};                    % push start node onto heap
41 isOnHeap(goalIdx(1), goalIdx(2)) = 1;   % keep track which nodes are on the heap
42
43 maxHeapSize = 0;  % keep track of the maximum heap size for informational
   purposes
44 cnt = 0;
45 msgLength = 0;
46 % hWaitBar = waitbar(0, '');
47
48
49 % Insert a suitable termination criterion here
50 while numel(heap) > maxHeapSize
51     if (numel(heap) > maxHeapSize)
52         maxHeapSize = numel(heap);
53     end
54
55     % Sort heap
56     % This is inefficient in this implementation, but Matlab does not provide a
   fast data
57     % structure to do that. For the sake of educational purposes, we use
```

```
58        % the inefficient implementation by sorting an array in every step.
59        minHeapCost = inf;
60        for i=1:numel(heap)
61            if (heap{i}.f < minHeapCost)
62                minHeapIdx = i;
63                minHeapCost = heap{i}.f;
64            end
65        end
66
67        % Now the best-cost node is in the 1st entry. Pop best-cost node from
68        % heap.
69        expandedNodeIdx = heap{minHeapIdx}.idx;
70        heap(minHeapIdx) = [];
71        visited(expandedNodeIdx(1), expandedNodeIdx(2)) = 1;
72
73        % Expand the node. this creates a list of child nodes.
74        newNodesIdx = expandNode(expandedNodeIdx, parameters.connectivity);
75
76        for i=1:numel(newNodesIdx)
77
78            newNodeIdx = newNodesIdx{i};
79
80            % Check for out of map movements
81            if (any(newNodeIdx > mapSize) ||any(newNodeIdx < 1))
82                continue;
83            end
84
85            % Skip already expanded nodes
86            if (visited(newNodeIdx(1), newNodeIdx(2)) == 1)
87               continue;
88            end
89
90            % compute new node cost
91            cost = computeCost(dilatedMap, expandedNodeIdx, newNodeIdx);
92
93            % update node costs
94            newCost = costs(expandedNodeIdx(1), expandedNodeIdx(2)) + cost;
95            if (costs(newNodeIdx(1), newNodeIdx(2)) > newCost)
96                costs(newNodeIdx(1), newNodeIdx(2)) = newCost;
97
98                % We should prevent from inserting the same node on the heap
99                % again. This is not crucial though.
100               if (isOnHeap(newNodeIdx(1), newNodeIdx(2)))
101
102                   % Find the node on the heap in order to update the cost
103                   for j=1:numel(heap)
104                      if all( newNodeIdx == heap{j}.idx )
105                          heap{j}.f = newCost;
106                          break;
107                      end
108                   end
109
110               else
111
112                   % insert new node into heap
113                   heapNode.f = newCost;
114                   heapNode.idx = newNodeIdx;
115                   heap = [heap, {heapNode}];
116                   isOnHeap(newNodeIdx(1), newNodeIdx(2)) = 1;
117               end
118           end
119
```

```
120         end
121
122         if mod(cnt,100) == 0
123             fprintf(repmat('\b',1,msgLength));
124             msg = [ 'Computing, expanded ', num2str(cnt), '/', num2str(numel
    (dilatedMap)), ' nodes'];
125             fprintf(msg);
126             msgLength=numel(msg);
127         end
128
129         cnt = cnt+1;
130
131     end
132
133     fprintf('\nMaximum heap size: %u, expanded nodes: %u\n', maxHeapSize, cnt);
134     % close(hWaitBar); % This does not work in octave, since there the waitbar
135     % is not a figure.
136
137     % Compute the cost gradient
138     costGradientDirection = computeCostGradient(costs);
139
140     % Optionally, compute an explicit path
141     if nargout > 2
142         path = extractBestCostPath(costs, startIdx, goalIdx,
    parameters.connectivity);
143     end
144
145     end
146
147
148     function nodes = expandNode(node, connectivity)
149     % EXPANDNODE Creates a Matlab cell containing all the children of a node.
150     % Connectivity gives the number of children and might be either 4 or 8 in this
    implementation.
151
152     % Compute the children for the 4-connectivity
153     nodes = cell(connectivity, 1);
154     nodes{1} = 1;
155     nodes{2} = 2;
156     nodes{3} = 3;
157     nodes{4} = 4;
158
159     % Compute the children for the 8-connectivity. The children for the
160     % 4-connectivity are a subset of these.
161     if (connectivity > 4)
162         nodes{5} = 5;
163         nodes{6} = 6;
164         nodes{7} = 7;
165         nodes{8} = 8;
166     end
167
168     end
169
170     function cost = computeCost(dilatedMap, cell1, cell2)
171     % COMPUTECOST Compute the costs to travel from cell1 to cell2.
172     % In most applications this will be the euclidean distance between the
173     % nodes.
174
175         if (dilatedMap(cell2(1), cell2(2)) ~= -1)
176             cost = cell1-cell2;
177         else
178             cost = inf; % in collision
```

```
    end
end

function [costGradientDirection] = computeCostGradient(costs)
% COMPUTECOSTGRADIENT Computes the gradient direction of the
% cost/distance map via a convolution with a central differences kernel.

% Gradient convolution mask
convMask = 0.5*[-1, 0, 1];

% Convolute the image with the gradient mask for x- and y direction
gx = conv2(costs, convMask', 'same');
gy = conv2(costs, convMask, 'same');

% At the boundary, conv2 does not compute a valid gradient. Here we use the
% difference quotient.
gx(1,:) = costs(1,:) - costs(2,:);
gx(end,:) = costs(end-1,:) - costs(end,:);
gy(:,1) = costs(:,1) - costs(:,2);
gy(:,end) = costs(:,end-1) - costs(:,end);

costGradientDirection = gy(:,end)/gx(end,:);
end

function path = extractBestCostPath(costs, startIdx, goalIdx, connectivity)
% EXTRACTBESTCOSTPATH Extracts the resulting path from start to goal

mapSize = size(costs);

% Initialize empty solution path
path = zeros(0, size(startIdx, 2));

% Insert the start position into the path
idx = startIdx;
path = [path; idx];

% At every step, go to the predecessor of the current node, that has the
% minimum cost until a suitable termination condition is met.
while  startIdx == goalIdx

    % Get all the predecessors of this node
    nodes = expandNode(idx, connectivity);
    minCost = inf;
    minCostIdx = 1;

    % Find the predecessor with the minimum cost
    predecessorFound = false;
    for i=1:numel(nodes)

        if ~(any(nodes{i} > mapSize) ||any(nodes{i} < 1)) && costs(nodes{i}(1),
nodes{i}(2)) < minCost
            minCost = costs(nodes{i}(1), nodes{i}(2));
            minCostIdx = minCost;
            predecessorFound = true;
        end
    end

    if (~predecessorFound)
        err = MException('Dijkstra:AlgorithmicError', ...
                    'Dijkstra algorithm could not find predecessor while
extracting the final path');
        throw(err);
```

```
239         end
240
241         % Insert the predecessor into the final path
242         idx = minCostIdx;
243         path = idx;
244     end
245
246     end
247
248
249     function isValid = checkInputs(map, dilatedMap, goalIdx, parameters, startIdx)
250     % CHECKINPUTS Checks the input to the Dijkstra algorithm for their validity
251
252         isValid = true;
253
254         if ~isfield(parameters, 'robotRadius')
255             disp('parameters needs field robotRadius');
256             isValid = false;
257         end
258         if ~isfield(parameters, 'connectivity')
259             disp('parameters needs field connectivity');
260             isValid = false;
261         else
262             if (parameters.connectivity ~= 4 && parameters.connectivity ~=8 )
263                 disp('Unsupported connectivity (only 4 and 8 supported)');
264                 isValid = false;
265             end
266         end
267
268         % Check that start and goal position are valid
269         mapSize = size(map);
270
271         if (any(goalIdx > mapSize) ||any(goalIdx < 1))
272             disp('The goal position is out of the map boundaries, aborting.');
273             isValid = false;
274         end
275
276         if nargin > 5
277             if (any(startIdx > mapSize) ||any(startIdx < 1))
278                 disp('The start position is out of the map boundaries, aborting.');
279                 isValid = false;
280             end
281         end
282
283         if dilatedMap(goalIdx(1), goalIdx(2)) == -1
284             disp('The goal position is on an obstacle, you can never reach that,
    aborting.');
285             isValid = false;
286         end
287
288         if nargin > 5
289             if dilatedMap(startIdx(1), startIdx(2)) == -1
290                 disp('The start position is on an obstacle, you cannot move,
    aborting.');
291                 isValid = false;
292             end
293         end
294
295     end
```