11.5 Dijkstra's algorithm

A property called the *weight* can be added to the edges of the graph.

- the weigth can represent the length of the route or the cost of the state transition

This makes finding the shortest route significantly more complicated.

- the shortest route from the source to the vertex is the one whose combined weight of the edges is as small as possible
- if the weight of each edge is 1, the problem can be solved with scanning through the part of the graph reachable from the source with breadth-first search
- if the weights can be <0, it is possible that there is no solution to the problem although there are possible paths in the graph
  - if there is a loop in the graph for which the sum of the weights if the edges is negative, an arbitrarily small sum of the weights can be created by travelling the loop as many times as necessary

This chapter covers finding the shortest path on a weighted, directed graph where the weigths of the edges are positive and can be other than one.

- more complicated algorithms are needed for managing negative edge weights, e.g. the Bellman-Ford algorithm

The Dijkstra's algorithm is used for finding the shortest weighted paths.

- finds the shortest paths from the source s to all vertices reachable from s by weighing the lengths of the edges with w
- chooses in each situation the shortest path it hasn't investigated yet
  $\Rightarrow$ it is a greedy algorithm

The algorithm uses a data structure Q, which is a priority queue (the chapter 3.2 in the lecture notes)

w contains the weights of all the edges

Dijkstra's algorithm uses the algorithm RELAX

- the relaxation of the edge (u, v) tests could the shortest path found to vertex v be improved by routing its end through u and does so if necessary

Otherwise the algorithm greatly resembles the breadth-first search

- it finds the paths in the order of increasing length
- once the vertex u is taken from Q its weighted distance from s is $d[u] = \delta(s, u)$ for sure
  - if the vertex taken from the priority queue is the goal vertex, the execution of the algorithm can be ended

```
DIJKSTRA(s, w)                          (the algorithm gets the source s as parameter)
1   ▷ in the beginning the fields of each vertex are colour = WHITE, d = ∞, π = NIL
2   s→colour := GRAY                    (mark the source found)
3   s→d := 0                            (the distance from the source to itself is 0)
4   PUSH(Q, s)                          (push the source into the priority queue)
5   while Q ≠ ∅ do                      (continue while there are vertices left)
6       u := EXTRACT-MIN(Q)             (take the next vertex from the priority queue)
7       for each v ∈ u→Adj do           (go through the neighbours of u)
8           if v→colour = WHITE then    (if the node has not been visited . . . )
9               v→colour := GRAY        (. . . mark it found)
10                  PUSH(Q, v)          (push the vertex into the queue)
11          RELAX(u, v, w)
12      u→colour := BLACK               (mark u as processed)
```

```
RELAX(u, v, w)
1   if v→d > u→d + w(u, v) then         (if a new shorter route to v was found...)
2       v→d := u→d + w(u, v)            (...decrement the distance of v from the source)
3       v→π := u                        (mark the v was reached through u)
```

Running-time:

- the while loop makes atmost $O(V)$ rounds and the for loop atmost $O(E)$ rounds in total
- the priority queue can be implemented efficiently with a heap or less efficiently with a list

with a heap:	with a list:	
line 4: $\Theta(1)$	$\Theta(1)$	(adding to an empty data structure)
line 5: $\Theta(1)$	$\Theta(1)$	(is the priority queue empty)
line 6: $O(\lg V)$	$O(V)$	(Extract-Min)
line 10: $\Theta(1)$	$\Theta(1)$	(the priority of a white vertex is infinite, its correct location is at the end of the queue)
line 11: $O(\lg V)$	$\Theta(1)$	(relaxation can change the priority of the vertex)

- one $O(\lg V)$ operation is done on each round of the while and the for loop when using a heap implementation
  $\Rightarrow$ the running-time of the algorithm is $O((V + E) \lg V)$

11.6 A*algorithm

Dijkstra's algorithm finds the shortest weighted route by accessing the nodes in the order of shortest route. I.e.: Dijkstra only uses information about routes found so far.

A* algorithm enhances this by adding a heuristic (= assumption) about the shortest possible distance to the goal. (For example, the direct distance ("as the crow flies") in a road network.)

- find the shortest weighted route from starting node s to given goal node g. **Does not** find the shortest route to *all* nodes (like Dijkstra).
- assumes that edge weights are non-negative (like Dijkstra)
- assumes that the minimum possible distance from any node to the goal can be calculated (i.e., the found shortest route cannot be shorter).
- chooses in each situation a new node, whose (shortest path from start to node + estimated minimum remaining distance to goal) is smallest.

The only difference between A* and Dijkstran is in relaxation (and the fact that A* should be terminated immediately when the goal is found, since it doesn't calculate shortest routes to all nodes).

RELAX-A*(u, v, w)

```
1  if v→d > u→d + w(u,v) then        (if a shorter route to node v is found...)
2       v→d := u→d + w(u,v)          (...new route this far...)
3       v→de := v→d + min_est(v,goal) (...and minimum estimate for whole route)
4       v→π := u                     (mark the v was reached through u)
```

In the priority queue used by A* the whole route's distance estimate $v{\rightarrow}de$ is used as the priority.

(Dijkstra's algorithm is a special case of A*, where $min\_est(a, b)$ is always 0.)