**Djakstra Algorithm**

Use priority\_queue, finding the shortest path to every node along the way

Each node has:

* int id, bool visited
* G\_value (= inf): Minimum cost required from start to the node (to be compared)
* Parent\_node: Previous **node** used to reach this node with minimum cost
* Parent\_edge: Previous **edge** used to reach this node with minimum cost (if there are multiple edges connecting 2 nodes)

Pseudocode: // implementation assumes that all nodes for all vertices has been defined

// if we create the nodes in-place while traversing, we need another dict

|  |
| --- |
| priority\_queue pq = new() (set up such minimum G-value always at the front)(Greedy)  pq.push(start\_node (g = 0)  while (pq.size() != 0):  curr = pq.pop() // Note that even if "end" is in queue, we don’t necessarily explore it right away, since there might be shorter paths  if (curr.value == end.value):  // Reconstruct the path from Parent\_node  for (adj\_node in curr.adj\_nodes):  pq.push(adj\_node)  g\_calculate = curr.G\_value + cost\_to\_reach\_adj\_node  if (g\_calculate < adj\_node.G\_value):  adj\_node.G\_value = g\_calculate  Parent\_node = curr |

Complexity: normally O(v2), if pq is efficient, O(e + vlog(v))

**A\* Algorithm**

Djakstra, but with heuristic value added (eucledian distance to the end node)