Ideas Implemented

Improve 1

You can cut down on the search time further by exploiting the Euclidean geometry of the problem, as described in Sedgewick 21.5. For general graphs, Dijkstra's relaxes edge v-w by updating wt[w] to the sum of wt[v] plus the distance from v to w. For maps, we instead update wt[w] to be the sum of wt[v] plus the distance from v to w plus the Euclidean distance from w to d minus the Euclidean distance from v to d. This is known as the A\* algorithm. This heuristics affects performance, but not correctness. (See Sedgewick 21.5 for a proof of correctness.)

Improving question 1, stop when you find a point

We use the idear one, here we search in the data structure and stop only if we find the item.

We o not need to continue since we already have the value we wanted.

These changes are made in file \_ Dijkstra\_improve\_p1.java

Parts Modified

// print shortest path from s to d (interchange s and d to print in right order)

public void showPath(int d, int s) {

dijkstra(s, d, 0);

if (pred[d] == -1) {

System.out.println(d + " is unreachable from " + s);

return;

}

for (int v = d; v != s; v = pred[v]) {

System.out.print(v + "-");

}

System.out.println(s);

}

// plot shortest path from s to d

public void drawPath(int s, int d) {

dijkstra(s, d, 1);

if (pred[d] == -1) {

return;

}

Turtle.setColor(Color.red);

for (int v = d; v != s; v = pred[v]) {

G.point(v).drawTo(G.point(pred[v]));

}

Turtle.render();

}

// Dijkstra's algorithm to find shortest path from s to d

public void dijkstra(int s, int d, int draw) {

int V = G.V();

// initialize

dist = new double[V];

pred = new int[V];

for (int v = 0; v < V; v++) {

dist[v] = INFINITY;

}

for (int v = 0; v < V; v++) {

pred[v] = -1;

}

// priority queue

IndexPQ pq = new IndexPQ(V);

//changes made here too

for (int v = 0; v < V; v++) {

pq.insert(v, dist[v]);

}

// set distance of source

dist[s] = 0.0;

pred[s] = s;

pq.change(s, dist[s]);

// run Dijkstra's algorithm

while (!pq.isEmpty()) {

int v = pq.delMin();

//// System.out.println("process " + v + " " + dist[v]);

if (d == v) {

break;

}

// v not reachable from s so stop

if (pred[v] == -1) {

break;

}

// scan through all nodes w adjacent to v

IntIterator i = G.neighbors(v);

while (i.hasNext()) {

int w = i.next();

//improve goes here

if (dist[v] - G.distance(v, d) + G.distance(v, w) + G.distance(w, d) < dist[w] - EPSILON) {

dist[w] = dist[v] - G.distance(v, d) + G.distance(v, w) + G.distance(w, d);

pq.change(w, dist[w]);

pred[w] = v;

}

}

}

}

improve 2

Improve the third question. Multipath heap

Use a faster priority queue. There is some room for optimization in the

supplied priority queue. You could also consider using a multiway heap as in

Sedgewick Program 20.10.

This idea is somehow simple but require another file as stated in the document,

that is, the binary data structure used in the original algorithm is changed to a multipurpose heap to achieve an improved algorithm.

Since the purpose of the changes is to improve perfomance, we can change the x parameter in the,

IndexMultiwayMinPQ(V, x) of multiway heap.

Purpose of performance. Thus, the value of the way parameter in the IndexMultiwayMinPQ(V, way) of the multiway heap can be changed.

Changes can be see in Dijkstra\_improve\_p2.java

// Dijkstra's algorithm to find shortest path from s to d

public void dijkstra(int s, int d, int draw) {

int V = G.V();

// initialize

dist = new double[V];

pred = new int[V];

for (int v = 0; v < V; v++) {

dist[v] = INFINITY;

}

for (int v = 0; v < V; v++) {

pred[v] = -1;

}

// priority queue

IndexMultiwayMinPQ pq = new IndexMultiwayMinPQ(V, way);

for (int v = 0; v < V; v++) {

pq.insert(v, dist[v]);

}

// set distance of source

dist[s] = 0.0;

pred[s] = s;

pq.changeKey(s, dist[s]);

// run Dijkstra's algorithm

while (!pq.isEmpty()) {

int v = pq.delMin();

if (d == v) {

break;

}

// v not reachable from s so stop

if (pred[v] == -1) {

break;

}

// scan through all nodes w adjacent to v

IntIterator i = G.neighbors(v);

while (i.hasNext()) {

int w = i.next();

if (dist[v] - G.distance(v, d) + G.distance(v, w) + G.distance(w, d) < dist[w] - EPSILON) {

dist[w] = dist[v] - G.distance(v, d) + G.distance(v, w) + G.distance(w, d);

pq.changeKey(w, dist[w]);

pred[w] = v;

}

}

}

}