Toteutusdokumentti

Yhteenveto

Kaikki määrittelydokumentissa mainitut algoritmit ja tietorakenteet onnistuttiin toteuttamaan toimivasti ja tehokkaasti. Ohjelman toimii tekstitilassa ja ohjelmaa käytetään komentoriviparametreillä.

Ohjelman ominaisuudet

Algoritmien nopeusvertailu

Ohjelma generoi halutun kokoisen labyrintin, ja etsii siinä lyhimmän kuljettavan polun kahden satunnaisesti valitun pisteen välillä. Polun etsintää suoritetaan annettu määrä hakuja, ja löydetyn reitin pituus ja algoritmin käyttämä aika tulostetaan käyttäjälle. Mikäli hakualgoritmilla kestää yli 15 sekuntia suoriutua ratkaisusta, arvioidaan lopullinen aika jo suoritettujen hakujen avulla.

Satunnaisen labyrintin generointi

Ohjelma generoi ja tulostaa satunnaisen labyrintin annetuilla parametreilla. Ohjelman tuloste voidaan ohjata esimerkiksi tiedostoon muokkausta varten.

Lyhimmän polun etsiminen luetusta labyrintti-tiedostosta

Ohjelma lukee halutun tiedoston ja ratkaisee sen kuvaaman labyrintin. Ratkaisuun käytetään A* algoritmia.

Satunnaisen labyrintin generointi ja lyhimmän polun etsintä

Kahden edellisen kohdan yhdistelmä; ohjelma generoi satunnaisen labyrintin ja etsii siitä lyhimmän polun.

Puutteet

- Huonot konfiguraatiomahdollisuudet ja osa toiminnallisuudesta on kovakoodattua.
- Käyttäjän syötteen oikeellisuuden virheentarkistus on puutteellista.

Algoritmien oikeellisuus ja aikavaativuudet

Yhteiset apumetodit

```
1.        protected void initializeNodes() {
2.              for (Node n : graph.getNodes()) {
3.                    n.reset();
4.
5.                    if (n.getType() == START) {
6.                          start = n;
7.                          start.setDistance(0);
8.                          start.setDistanceEstimate(0);
9.                    }
10.
11.                   if (n.getType() == TARGET) {
12.                         target = n;
13.                   }
14.             }
15.       }
```

Apumetodi alustaa kaikki solmut. Aikavaativuus on **O(|V|)**.

```
1. public LinkedList<Node> getNeighbours(Node node) {
2.        LinkedList<Node> list = new LinkedList<>();
3.
4.        int x0 = node.getX(), y0 = node.getY();
5.
6.        if (!isWithinGraph(x0, y0)) {
7.              return list;
8.        }
9.
10.       for (int y = y0 - 1; y <= y0 + 1; ++y) {
11.             for (int x = x0 - 1; x <= x0 + 1; ++x) {
12.                   if (y == y0 && x == x0) {
13.                         continue;
14.                   }
15.
16.                   if (!isWithinGraph(x, y)) {
17.                         continue;
18.                   }
19.
20.                   list.add(graph[y][x]);
21.             }
22.       }
23.
24.       return list;
25.}
```

Apumetodi palauttaa kaikki solmun naapurit. Aikavaativuus on **O(1)**.

Bellman-Fordin algoritmi

```
1.  public void findPath() {
2.        initializeNodes();
3.
4.        final LinkedList<Node> nodes = getGraph().getNodes();
5.
6.        for (int i = 0; i < nodes.size() - 1; ++i) {
7.              for (Edge e : edges) {
8.                    relax(e.u, e.v, e.w);
9.              }
10.       }
11. }
12.
13. private void relax(Node u, Node v, double w) {
14.       if (u.getDistance() == Double.MAX_VALUE) {
15.             return;
16.       }
17.
18.       double uw = u.getDistance() + w;
19.
20.       if (v.getDistance() > uw) {
21.             v.setDistance(uw);
22.             v.setNearest(u);
23.       }
24. }
```

Algoritmin lähteenä on käytetty Tietorakenteet ja algoritmit -kurssin kurssimateriaalia. Koodin rakenne ja ulkoasu on tarkoituksella pyritty pitämään kurssimateriaalin pseudokoodia vastaavana. Erilaista on ainoastaan 14. rivin ehto, joka estää turhan vertailun.

Rivin 2 aikavaativuus **O(|V|)**

Rivin 8 aikavaativuus **O((|V|-1)|E|)**

Algoritmin aikavaativuus on **O(|V||E|)**.

Dijkstran algoritmi

```
1. public void findPath() {
2.         initializeNodes();
3.
4.         final LinkedList<Node> nodes = getGraph().getNodes();
5.         final MinimumHeap<Node> heap = new MinimumHeap<>(1);
6.         for (Node node : nodes) {
7.                 heap.insert(node);
8.         }
9.
10.        while (true) {
11.                final Node node = heap.getMin();
12.
13.                if (node == getTarget()) {
14.                        return;
15.                }
16.
17.                for (Node neighbour : getGraph().getNeighbours(node)) {
18.                        if (neighbour.getType() == WALL) {
19.                                continue;
20.                        }
21.
22.                        double d = node.getDistance() + getDistance(node, neighbour);
23.
24.                        if (d < neighbour.getDistance()) {
25.                                neighbour.setDistance(d);
26.                                neighbour.setNearest(node);
27.
28.                                heap.decreaseKey(neighbour);
29.                        }
30.                }
31.        }
32. }
```

Tämäkin algoritmi on kirjoitettu kurssimateriaalia mukaillen. Ainoastaan relax() metodi on kirjoitettu auki vierussolmut läpikäyvään looppiin.

Rivin 2 aikavaativuus on **O(|V|)**

Rivin 7 aikavaativuus on **O(|V|log|V|)**

Rivin 11 aikavaativuus on **O(|V|log|V|)**

Rivin 28 aikavaativuus on **O(|E|log|V|)**

Algoritmin aikavaativuus on **O((|E|+|V|)log|V|)**.

A*-algoritmi

```
1. public void findPath() {
2.         initializeNodes();
3.         heap = new MinimumHeap<>();
4.         heap.insert(getStart());
5.
6.         while (!heap.isEmpty()) {
7.                 final Node node = heap.getMin();
8.                 node.setClosed();
9.
10.                if (node == getTarget()) {
11.                        return;
12.                }
13.
14.                for (Node neighbour : getGraph().getNeighbours(node)) {
15.                        processNode(node, neighbour);
16.                }
17.        }
18. }
19.
20. void processNode(Node node, Node neighbour) {
21.        if (neighbour.getType() == PathFinder.WALL || neighbour.isClosed()) {
22.                return;
23.        }
24.
25.        final double d = node.getDistance() + getDistance(node, neighbour);
26.        if (d < neighbour.getDistance()) {
27.                neighbour.setDistance(distance);
28.                neighbour.setNearest(node);
29.
30.                if (neighbour.isOpen()) {
31.                        heap.decreaseKey(neighbour);
32.                }
33.
34.                if (!neighbour.isOpen()) {
35.                        final int dx = neighbour.getX() - getTarget().getX();
36.                        final int dy = neighbour.getY() - getTarget().getY();
37.
38.                        neighbour.setDistanceEstimate(heuristic.distance(dx, dy));
39.                        neighbour.setOpen();
40.
41.                        heap.insert(neighbour);
42.                }
43.        }
44. }
```

Rivin 2 aikavaativuus on **O(|V|)**

Rivin 7 aikavaativuus on **O(|V|log|V|)**

Rivin 31 aikavaativuus on **O(|E|log|V|)**

Rivin 41 aikavaativuus on **O(|V|log|V|)**

Algoritmin aikavaativuus on **O((|E|+|V|)log|V|)**.

Minimikeko

```
1. private int left(int i) {
2.       return (i == 0 ? 1 : 2 * i);
3. }
4.
5. private int parent(int i) {
6.       return i / 2;
7. }
8.
9. private int right(int i) {
10.      return (i == 0 ? 2 : 2 * i + 1);
11.}
```

left(), right() ja parent() metodien aikavaativuus on **O(1)**.

```
1. private void heapify(int i) {
2.       int left = left(i);
3.       int right = right(i);
4.
5.       if (right < heapSize) {
6.             int smallest;
7.
8.             if (get(left).getKey() < get(right).getKey()) {
9.                   smallest = left;
10.            } else {
11.                  smallest = right;
12.            }
13.
14.            if (get(i).getKey() > get(smallest).getKey()) {
15.                  swap(i, smallest);
16.                  heapify(smallest);
17.            }
18.      }
19.      else if (left == (heapSize - 1) && get(i).getKey() > get(left).getKey()) {
20.            swap(i, left);
21.      }
22.}
```

heapify() kutsuu itseään rekursiivisesti alkaen indeksistä i ja kaksinkertaistuu joka kutsulla, maksimiarvonaan keon koko. Tämän perusteella heapify() metodin aikavaativuus on **O(log n)**.

```
1. public E getMin() {
2.         E element = get(0);
3.         heapSize--;
4.
5.         if (isEmpty()) {
6.                 elements[0] = null;
7.         } else {
8.                 assign(0, heapSize);
9.                 heapify(0);
10.        }
11.
12.        return element;
13. }
```

Rivin 9 aikavaativuus on **O(log n)**

getMin() metodin aikavaativuus on **O(log n)**.

```
1. public void insert(E e) {
2.         heapSize++;
3.         if (heapSize > elements.length) {
4.                 elements = copyArray(elements, elements.length * 2);
5.         }
6.
7.         int i = heapSize - 1;
8.         while (i > 0 && get(parent(i)).getKey() > e.getKey()) {
9.                 assign(i, parent(i));
10.                i = parent(i);
11.        }
12.
13.        elements[i] = e;
14. }
```

Rivin 4 aikavaativuus tasoitettu **O(1)**

Rivien 8-11 aikavaativuus on **O(log n)**

insert() metodin aikavaativuus on **O(log n)**.

```
1. public void decreaseKey(E e) {
2.         int i = e.getIndex();
3.
4.         if (i == -1) {
5.                 return;
6.         }
7.
8.         while (i > 0 && get(parent(i)).getKey() > e.getKey()) {
9.                 swap(i, parent(i));
10.                i = parent(i);
11.        }
12. }
```

Rivien 8-11 aikavaativuus on **O(log n)**

decreaseKey() metodin aikavaativuus on **O(log n)**.

Linkitetty lista

```
1. public boolean add(E e) {
2.         Node n = new Node(e);
3.         tail.next = n;
4.         tail = n;
5.         size++;
6.
7.         return true;
8. }
9.
10.public void clear() {
11.        head.next = null;
12.        tail = head;
13.        size = 0;
14.}
15.
16.public E get(int index) {
17.        Node node = head.next;
18.        for (int i = 0; i < index; ++i) {
19.                node = node.next;
20.        }
21.
22.        return node.data;
23.}
```

add() metodin aikavaativuus on **O(1)**.

clear() metodin aikavaativuus on **O(1)**.

get() metodin aikavaativuus on **O(n)**.