Tietorakenteet ja algritmit harjoitustyö

A* ja Dijkstran algoritmin vertailuun tarkoitettu ohjelma

Toteutusdokumentti

Juhani Heliö

Tehokkuudet

Algoritmien ja tietorakenteiden aikavaativuudet ja tilavaativuudet ovat määriteltyä luokkaa.

A* pseudokoodi:

```
public int laskeReitti(){
      openList.clear();
      closedList.clear();
      openList.add(v.getAlkuNode());

      while(!openList.isEmpty()){    // while-loop pitää suorittaa pahimmassatapauksessa n+m
                                        kertaa, missä n=solmujen määrä ja m=kaarien määrä
         Node current=openList.remove();   //minimikeosta poistaminen tapahtuu O(log n) ajassa
         if(current.equals(v.getMaaliNode())){
            return lyhinReitti(v.getMaaliNode());
         }

         closedList.add(current);
         for(int i=0;i<current.getNaapurit().size();i++){    //vakioaikainen, koska suoritetaan max 4
                                                               kertaa joka solmulle
            Node node=current.getNaapurit().get(i);
            if(!closedList.contains(node)&&!node.isObstacle()){
               int matkaAlkuun=current.getMatkaAlkuun()+1;
               if(!openList.contains(node) || matkaAlkuun<current.getMatkaAlkuun()){
                  node.setEdellinen(current);
                  node.setMatkaAlkuun(matkaAlkuun);
                  node.setMatkaMaaliin(node.getMatkaAlkuun()+Heuristic.matka(node,
                  v.getMaaliNode()));
                  if(!openList.contains(node)){
                     openList.add(node);   //minimikekoon lisääminen tapahtuu O(log n) ajassa
                  }
               }
            }
         }
      }
      return -1;
   }
```

eli yhteensä siis pahimman tapauksen aikavaativuus on määrittelydokumentin luokkaa O((n+m)log n). Tilavaativuus on myös määriteltyä O(n) luokkaa.

```
public void laskeReitti(){
      nodeList.clear();
      closedList.clear();
      nodeList.add(v.getAlkuNode());
      v.getAlkuNode().setPaino(0);

      while(!nodeList.isEmpty()){     //kuten yllä, suoritetaan pahimmassa tapauksessa n+m kertaa
         Node current=nodeList.remove(); //minimikeosta poistaminen tapahtuu O(log n) ajassa
         closedList.add(current);  //ArrayListiin lisääminen vakioaikaista, paitsi jos listaa
                                     pidennetään, mutta suurimmaksi osaksi vakioaikaista.
```

```
            if(current.equals(v.getMaaliNode())){
                haeReitti();
                return;
            }

            for(int i=0;i<current.getNaapurit().size();i++){       //vakioaikainen, koska suoritetaan max 4
                                                                   kertaa joka solmulle
                Node node=current.getNaapurit().get(i);
                if(!node.isObstacle()&&!closedList.contains(node)){
                    int matka=current.getMatkaAlkuun()+1;
                    if(matka<node.getPaino()){
                        node.setEdellinen(current);
                        node.setPaino(matka);
                        node.setMatkaAlkuun(matka);
                        nodeList.add(node);   //minimikekoon lisääminen tapahtuu O(log n) ajassa
                    }
                }
            }
        }
    }
```

Eli samoin kuin A*:issa, aika- ja tilavaativuudet ovat määrittelydokumentin luokkaa O((n+m)logn) ja O(n).

Tietorakenteet

NodeLista toimii kuten ArrayList, jonka tyyppinä olisi ohjelmassa käytetty ja itse toteutettu Node. Sen kaikki muut metodit toimivat O(1) ajassa, paitsi add ja contains jotka kummatkin voivat toimia pahimmassa tapauksessa O(n) ajassa. Add toimii O(1) ajassa melkein aina ja O(n) vain jos listaa täytyy pidentää.

OmaPriorityMinHeap toimii kuten javan PriorityQueue, eli käyttää minimikekoa. Sen kaikki muut paitsi add ja remove metodit (ja heapify, joka tosin on private) toimivat O(1) ajassa

```
public void add(Node node){
        heap.add(node);
        int paikka=heap.size()-1;
        if(heap.isEmpty()){
            heap.add(node);
            return;
        }
        while(paikka>=0&&paikka<heap.size()&&node.compareTo(heap.get(parent(paikka)))<0){
            //Suoritetaan pahimmassa tapauksessa puun syvyyden verran kertoja eli O(log n) ajassa
            swap(paikka, parent(paikka));
            paikka=parent(paikka);
        }
    }
```

eli add toimii O(log n) ajassa

```
public Node remove(){
        if(heap.isEmpty()){
```

```
            return null;
        }
        else{
            Node n=heap.get(0);
            heap.set(0, heap.get(heap.size()-1));
            heap.remove(heap.size()-1);
            heapify(0); //heapify toimii O(log n) ajassa, joten remove toimii O(log n) ajassa
            return n;
        }
    }
```

Remove toimii O(log n) ajassa, koska heapify toimii O(log n) ajassa.

```
private void heapify(int i){
        int vasenLapsi=vasenL(i);
        int oikeaLapsi=oikeaL(i);
        int pienin;

        if(vasenLapsi>=0
            &&vasenLapsi<heap.size()
            &&heap.get(vasenLapsi).compareTo(heap.get(i))<0){

            pienin=vasenLapsi;
        }
        else{
            pienin=i;
        }

        if(oikeaLapsi>=0
            &&oikeaLapsi<=heap.size()-1
            &&heap.get(oikeaLapsi).compareTo(heap.get(pienin))<0){
            pienin=oikeaLapsi;
        }

        if(pienin!=i){
            swap(i, pienin);
            heapify(pienin); //rekursiota kutsutaan korkeintaan puun korkeuden verran kertoja eli log n
                                kertaa
        }
    }
```

Eli heapify toimii O(log n) ajassa.