UNIVERSITAT ROVIRA i VIRGILI

Pràctica 1: Cerca informada

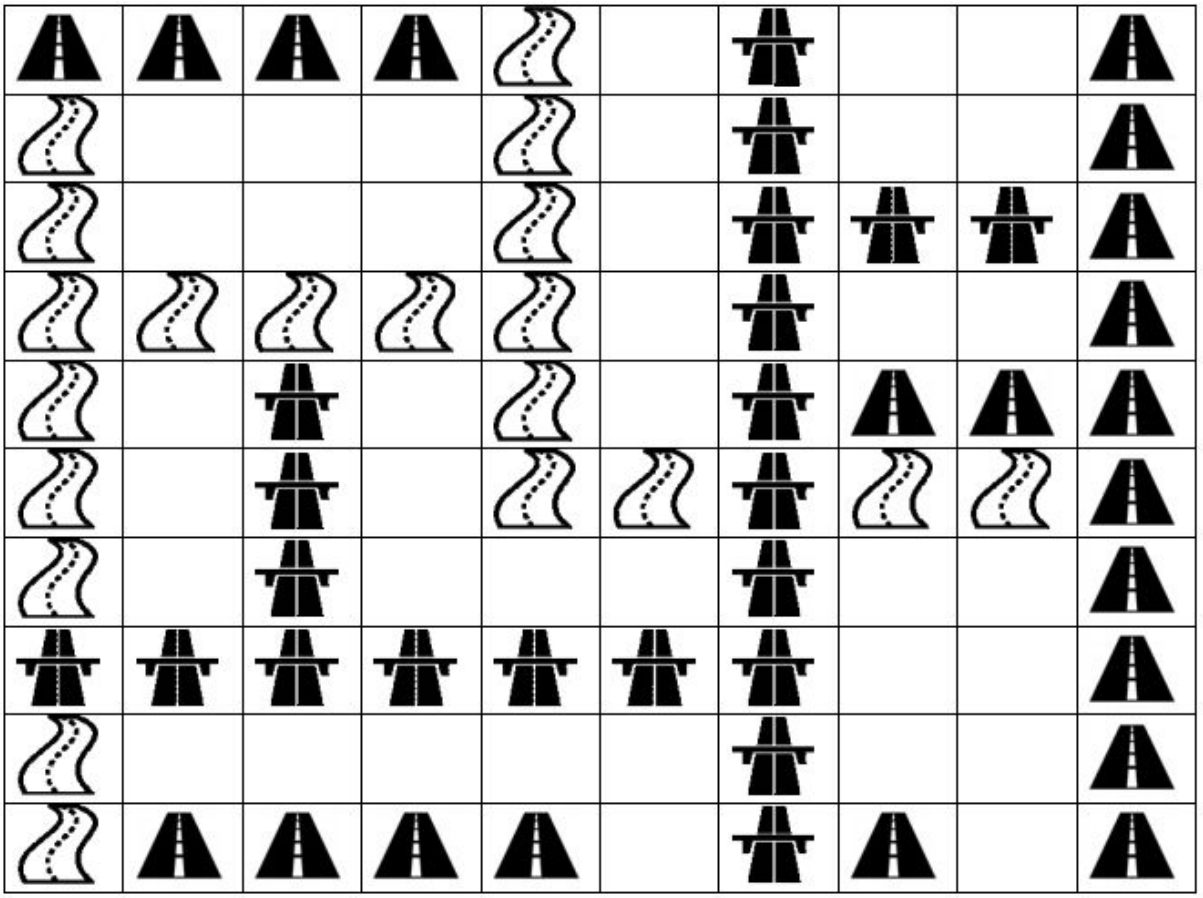
Nerea A. López Martín

28 - 10 - 2019

Índex

Enunciat

En aquesta pràctica volem estudiar la forma més ràpida de moure'ns per un mapa amb diferents tipus de carreteres. El mapa estarà representat per una matriu de 10*10 caselles. Cada problema estarà definit per una configuració del mapa, una certa casella inicial (xi, yi) i una casella final (xf, yf). L'aspecte que pot tenir un problema concret podria ser el següent:



Sortida: (0,0) Destí: (9,9)

 : Autovia  : Carretera nacional  : Carretera comarcal

Per les caselles en blanc no es pot circular.

Podem desplaçar-nos en horitzontal i en vertical (però no en diagonal) una casella cada vegada. El temps que triguem en moure'ns d'una casella a una altra dependrà del tipus de carretera de la casella destí.

- Autovia: 1 unitat de temps
- Carretera nacional: 2 unitats de temps
- Carretera comarcal: 3 unitats de temps

Es demana és el següent:

- Formalitzeu el problema definint els estats i els operadors.

- Doneu 3 heurístiques ben diferenciades (no tenen per què ser les 3 millors, però han de ser ben diferents) per intentar trobar el camí/camins més ràpids des de l'estat inicial al final.

- Per cada heurística, indiqueu si són o no admissibles respecte al temps. No cal que les 3 heurístiques que dissenyeu siguin admissibles, però almenys una d'elles ho hauria de ser.

- Feu un programa que resolgui el problema fent una cerca heurística amb el mètode best first.

- Feu un programa que resolgui el problema fent una cerca heurística amb el mètode A*.

- Proveu ambdós algorismes amb les 3 heurístiques per a diferents problemes (el de l'enunciat i, almenys, un altre mapa que dissenyeu vosaltres) indicant:
    - La solució (camí) que s'ha trobat amb el temps que li correspon.
    - El nombre de estats que ha "tractat" l'algorisme de cerca per trobar el camí.
    - Si la solució trobada és l'òptima respecte al temps o no.
- Per a cada heurística que heu dissenyat, hauríeu trobat la mateixa solució si haguéssiu aplicat l'algorisme hill climing? No cal implementar l'algorisme hill climbing només justificar-ho.

Comentaris d'implementació

En quant a la formalització del problema els estats serien les posicions i els operadors els costos de les carreteres. A l'implementació tot queda a la classe estat ja que el cost dels operadors depèn del tipus de carretera de l'estat següent, llavors cada estat té el guardat el seu propi cost.

Heurística 1

Aquesta calcula la distància entre el node en qüestió i el final. La idea és que al ser un mapa quadrat en el que no hi ha moviment en diagonal només hi haurà moviment en un eix i per tant la distància només es calcula per aquest ja que l'altre serà 0.
Aquesta heurística és admissible ja que el seu resultat no serà mai superior al cost real, donat que en el millor dels casos (camí mínim) serà igual a aquest.

Heurística 2

Retorna el cost del node i així s'escull aquell amb un cost menor.

Heurística 3

Aquesta suma les dues heurístiques anteriors. A més revisa la llista de successors del node actual i del final per veure si hi ha estats comuns donat que en aquest cas es una bona opció.

Cap de les heurístiques anteriors aseguraria trobar una solució amb l' algorisme hill climbing ja que no tenen perquè anar per el camí òptim a la primera i dins de la cerca a l'arbre d'estats poden visitar branques que no arriben al node final.

Codi dels algorismes i heurístiques

Best First

```
public class BestFirst {

  protected Heuristic heuristic;
  protected List < NodeBestFirst > pendingList;
  protected HashMap< Integer, State > treatedMap;
  protected List < NodeBestFirst > way;

  public BestFirst() {

    pendingList = new ArrayList<>();
    heuristic = new Heuristic();
    treatedMap = new HashMap<>();
  }


  public List< NodeBestFirst > Search(State iniState, State finalState ) {

    NodeBestFirst previousNode = null;
    NodeBestFirst currentNode = new NodeBestFirst( iniState, previousNode,
heuristic.roadType(iniState) );
    boolean found = false;
    addPending( currentNode );


    while ( !found && !pendingList.isEmpty() ) {

      currentNode = getPending();

      if( currentNode.getState().equals( finalState ) ) {

        found = true;

        way = new ArrayList<>();

        while (currentNode != null) {

          way.add(currentNode);
          currentNode = currentNode.getPreviousNode();
        }
```

```
            Collections.reverse(way);
        }

        else {

            for ( State state : currentNode.getState().getSuccessorList() )
                if( !treatedMap.containsValue( state ) ) {

                    NodeBestFirst newNode = new NodeBestFirst( state, currentNode,
heuristic.roadType(state));
                    addPending( newNode );
                }

            treatedMap.put( currentNode.getState().hashCode(), currentNode.getState() );
        }
    }

    return way;
  }

  public int getWayCost( List< NodeBestFirst > way ) {

    int cost = 0;

    int costInitalNode = way.get( 0 ).getState().getRoadType();

    for ( NodeBestFirst node : way )
      cost += node.getState().getRoadType();

    return cost - costInitalNode;
  }


  public void addPending( NodeBestFirst node ) {

    boolean foundEqual = false;

    // The BF objective is being fast finding a way so repeated nodes are ignored.
    for ( NodeBestFirst auxNode : pendingList )
      if ( auxNode.getState().equals( node.getState() ) )
        foundEqual = true;

    if ( !foundEqual ) {

      pendingList.add( node );
      Collections.sort( pendingList );
```

```
    }
  }

  public NodeBestFirst getPending() {

    int pos = 0;
    NodeBestFirst node = pendingList.get( pos );

    pendingList.remove( node );
    Collections.sort( pendingList );

    return node;
  }
}

public class NodeBestFirst implements Comparable< NodeBestFirst > {

  private State state;
  private NodeBestFirst previousNode;
  private int heuristic;


  public NodeBestFirst(State state, NodeBestFirst previousNode, int heuristic) {

    this.state = state;
    this.previousNode = previousNode;
    this.heuristic = heuristic;
  }

  public int compareTo(NodeBestFirst node) {

    int result;

    if (this.heuristic < node.heuristic)
      result = -1;

    else if (this.heuristic == node.heuristic)
      result = 0;

    else result = 1;

    return result;
  }


  public State getState() {
```

```
    return state;
  }

  public NodeBestFirst getPreviousNode() {
    return previousNode;
  }
```

A*

```
public class AStar {

  protected List< NodeAStar > pendingList;
  protected HashMap< Integer, State > treatedMap;
  protected List < NodeAStar > way;
  protected Heuristic heuristic;

  public AStar() {

    pendingList = new ArrayList<>();
    treatedMap = new HashMap<>();
    heuristic = new Heuristic();
  }


  public List< NodeAStar > Search( State iniState, State finalState ) {

    NodeAStar previousNode = null;
    NodeAStar currentNode = new NodeAStar( iniState, previousNode, heuristic.roadType(iniState),
iniState.getRoadType() );
    boolean found = false;
    addPending( currentNode );


    while ( !found && !pendingList.isEmpty() ) {

      currentNode = getPending();

      if( currentNode.getState().equals( finalState ) ) {

        found = true;

        // Creation of the way list.
        way = new ArrayList<>();

        while( currentNode != null ) {
```

```
            way.add( currentNode );
            currentNode = currentNode.getPreviousNode();
          }

          Collections.reverse( way );
        }
        else {

          for ( State state : currentNode.getState().getSuccessorList() )
            if( !treatedMap.containsValue( state ) ) {

              NodeAStar newNode = new NodeAStar( state, currentNode,  heuristic.roadType(state) ,
state.getRoadType() );
              addPending( newNode );
            }

          treatedMap.put( currentNode.getState().hashCode(), currentNode.getState() );
        }
      }

      return way;
    }

    public int getWayCost( List< NodeAStar > way ) {

      int cost = 0;

      int costInitalNode = way.get( 0 ).getState().getRoadType();

      for ( NodeAStar node : way )
        cost += node.getState().getRoadType();

      return cost - costInitalNode;
    }


    public void addPending(NodeAStar node) {

      // In order to find an optimum result and knowing that the heuristic is
      // not necessary true, in case of repeated nodes we save both.
      pendingList.add(node);
      Collections.sort(pendingList);
    }

    public NodeAStar getPending() {
```

```
      int pos = 0;
      NodeAStar node = pendingList.get(pos);

      pendingList.remove(node);
      Collections.sort(pendingList);

      return node;
    }
}
```

public class NodeAStar implements Comparable< NodeAStar >{

```
    private State state;
    private NodeAStar previousNode;
    private int heuristic;
    private int accumCost;

    public NodeAStar( State state, NodeAStar previousNode, int heuristic, int currentCost ) {

      this.state = state;
      this.previousNode = previousNode;
      this.heuristic = heuristic;
      accumCost = getAccumCost( currentCost );
    }

    public int compareTo( NodeAStar node ) {

      int result;

      if ( this.getFunction() < node.getFunction())
        result = -1;

      else if ( this.getFunction() == node.getFunction())
        result = 0;

      else result = 1;

      return result;
    }


    public int getAccumCost( int accumCost ) {

      if( previousNode != null )
        accumCost += previousNode.getAccumCost( accumCost );
```

```
    return accumCost;
  }

  public int getFunction() {
    return heuristic + accumCost;
  }


  public State getState() {
    return state;
  }

  public NodeAStar getPreviousNode() {
    return previousNode;
  }
}
```

Heurístiques

Heurística 1

```
public int distance( State currentState, State finalState ) {

    int heuristic;

    int incrementX;
    int incrementY;

    int iniX = currentState.getColumn();
    int iniY = currentState.getRow();
    int finalX = finalState.getColumn();
    int finalY = finalState.getRow();


    incrementX = finalX - iniX;
    if( incrementX < 0 ) incrementX = - incrementX;

    incrementY = finalY - iniY;
    if( incrementY < 0 ) incrementY = - incrementY;


    // There are no diagonals so the increment will be exclusive for an axis.
    if ( incrementX > 0 )
      heuristic = incrementX;
```

```
    else heuristic = incrementY;


    return heuristic;
  }
```

Heurística 2

```
public int roadType( State state ) {

    int heuristic = state.getRoadType();

    return heuristic;
  }
```

Heurística 3

```
public int distanceRoadType( State currentState, State finalState ) {

    int heuristic;
    int distance = distance( currentState, finalState );
    int roadType = roadType( currentState );

    heuristic = distance + roadType;

    for( State stateC : currentState.getSuccessorList() )
       for( State stateF : finalState.getSuccessorList() )
          if( stateC == stateF )
             heuristic += 10;

    return heuristic;
  }
```

Jocs de proves

Falta documentar aquest apartat, tot i fer la entrega la meva idea és presentarme al següent termini.