Miika Nurminen

Optimaalisen polun haku graafista A* -algoritmilla

TIE340 (Tekoäly) -kurssin
harjoitustyöraportti
3.3.2003

Jyväskylän yliopisto
Tietotekniikan laitos

Tietoja raportista

Tekijä: Miika Nurminen (minurmin@cc.jyu.fi)

Työn nimi: Optimaalisen polun haku graafista A* -algoritmilla

Title in English: Searching optimal path in graph using A* -algorithm

Työ: TIE340 (Tekoäly) -kurssin harjoitustyöraportti.

Sivumäärä: 13

Tiivistelmä: Raportissa kuvataan A*-algoritmiin perustuvaa ratkaisua optimaalisen polun löytämiseen suunnatussa ja kaarilta painotetussa graafissa. Lisäksi kuvataan Python-kielellä toteutettua graafista sovellusta, jota käytetään algoritmin toiminnan seuraamiseen ja verkkomallin määrittelyyn. Sovellus on saatavilla tekijän kotisivujen alta:
<URL: http://users.jyu.fi/~minurmin/opiskelu/ai/>

Avainsanat: A*-algoritmi, GXL, heuristinen haku, Python, tekoäly, verkkoteoria.

Keywords: A*-algorithm, Artificial intelligence, graph theory, GXL, heuristic search, Python.

Sisältö

1 Tehtävän kuvaus **1**

2 Ratkaisumenetelmä **2**

3 PathFinder-sovellus **3**
3.1 Järjestelmän rakenne ja rajapinnat 3
3.2 Käyttöliittymä ja olemassaolevan koodin käyttö 4

Lähteet **6**

Liitteet **7**

A Hakuesimerkki **7**

B A*-toteutuksen ohjelmalistaus **8**

C GXL-esimerkki **10**

1 Tehtävän kuvaus

Lyhimmän polun hakeminen graafista on merkittävä ongelma monilla tekoälyn sovellusalueilla. Ongelmaan on olemassa tarkkoja ratkaisuja, mutta suurilla graafeilla tarkan ratkaisun laskeminen voi olla hidasta. Esimerkiksi Floydin algoritmin kompleksisuus on $O(n^3)$.

Harjoitustyön tarkoituksena on lyhimmän polun hakeminen suunnatussa ja kaarilta painotetussa graafissa kahden annetun solmun väliltä. Lisäksi oletetaan, että jokaisesta solmusta tiedetään sijainti, jonka perusteella lasketaan kaarille painot. Solmujen sijaintia käytetään myös heuristisen kustannusfunktion laskemiseen.

Tehtävän havainnollistamiseksi varsinaisen ratkaisualgoritmin lisäksi toteutettiin yksinkertainen sovellus, jolla käyttäjä voi luoda tai muuttaa verkon rakennetta graafisesti ja seurata polun hakemisen etenemistä. Lisäksi pyrittiin huomioimaan liitännät muihin graafeja käsitteleviin sovelluksiin.

Algoritmia testattiin Jyväskylän keskustan katuja likimääräisesti mallintavalla suppeahkolla graafilla, jossa solmut vastaavat tienristeyksiä ja kaaret katuja. Yksisuuntaiset kadut otetaan huomioon kaarien suuntauksella ja solmujen suhteelliset paikat vastaavat sijaintia kartalla. Solmut on esitetty kaksiulotteisessa euklidisessa avaruudessa, mutta algoritmi on sovellettavissa myös yleisempään tilanteeseen.

2 Ratkaisumenetelmä

Lyhimmän polun laskemiseen käytetään heuristista hakua A*-algoritmilla. Paikkainformaation ansiosta haku on selvästi tarkkaa ratkaisua nopeampi ja voidaan osoittaa, että A*-algoritmi saavuttaa tarkkojen algoritmien tapaan parhaan hakutuloksen. [2, 3]

Heuristiikkana käytettiin solmujen paikoista laskettua euklidista etäisyyttä käsiteltävästä solmusta loppusolmuun. Toteutuksessa käytetty algoritmi on seuraava (ks. liite B):

1. Alustus
    1 Määrittele alku- ja loppusolmut (`startNode`), (`endNode`) sekä listat `open` ja `closed`.
    2 Aseta alkusolmun kuljetuksi matkaksi (`minBeginCost`) 0.
    3 Lisää `open`-listaan alkusolmu.
2. Iterointi
    1 Jos `open`-lista on tyhjä, mene pääkohtaan 3.
    2 Poimi `open`-listan ensimmäinen alkio arvioitavaksi.
    3 Lisää alkio `closed`-listaan hakustrategian mukaiselle paikalle.
    4 Jos alkio on loppusolmu, mene pääkohtaan 3.
    5 Tutkitaan alkion naapurisolmut. Jos jokin naapurisolmu on jo `open` tai `closed`-listassa ja niiden kuljettu matka on suurempi kuin tämänhetkinen, korjaa `open` ja `closed` -listojen solmujen kuljetut matkat rekursiivisesti.
    6 Aseta lopuille alkioille kuljettu matka ja lisää ne `open`-listaan hakustrategian mukaiselle paikalle.
    7 Palaa kohtaan 1.
3. Optimaalisen polun palautus
    1 Alusta lista optimaalisista poluista ja valitse loppusolmu.
    2 Lisää valittu solmu listan alkuun.
    3 Jos valittu solmu on alkusolmu, lopeta ja palauta lista.
    4 Käy läpi solmut, joista on kaari valittuun solmuun.
    5 Valitse uudeksi solmu, jonka kuljettu matka on pienin. Jos solmua ei ole, lopeta ja palauta tyhjä lista.
    6 Palaa kohtaan 2.

3 PathFinder-sovellus

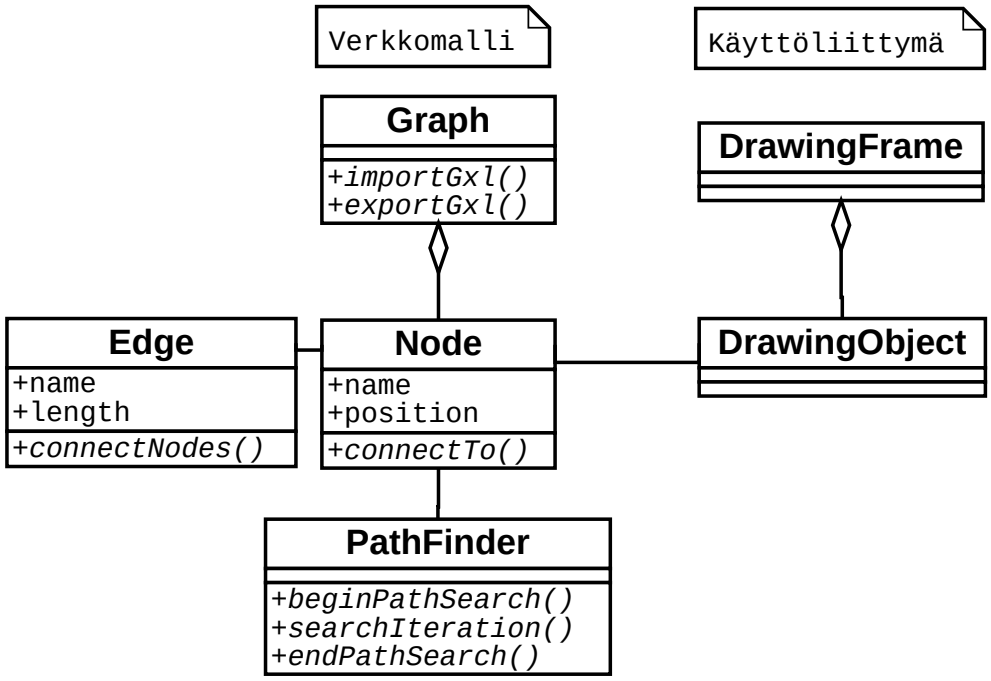
Sovelluksen toteutuskieleksi valittiin Python 2.2. Syinä tähän ovat Pythonin käyttöjärjestelmäriippumattomuus, oliopohjaisuus ja tehokkaat listojen ja assosiaatiotaulujen käsittelytoiminnot [4].

Sovelluksen tavoitteeksi asetettiin yleiskäyttöisyys, laajennettavuus ja mahdollisuus toimia muiden graafiohjelmien kanssa. Tavoitteet huomioitiin järjestelmän rakenteessa (verkkomallin esitys ja käyttöliittymä on erotettu toisistaan) sekä tallennusformaatin (GXL) ja käyttöliittymäkirjaston (wxPython) valinnoissa.

Toteutuksessa keskityttiin lähinnä A*-algoritmin toteutukseen ja graafien esittämiseen. Käyttöliittymä tehtiin wxPython-käyttöliittymäkirjaston mukana tulleen pySketch-esimerkkiohjelman päälle. Muokattu sovellus on saatavilla tekijän kotisivujen alta:
<URL: http://users.jyu.fi/~minurmin/opiskelu/ai/>

3.1 Järjestelmän rakenne ja rajapinnat

Kuvassa 1 on esitetty järjestelmän rakenne UML-luokkakaaviona.



Kuva 1: Järjestelmän luokkakaavio.

Luokat `Graph`, `Node` ja `Edge` kuvaavat verkon rakenteen. `Graph` toimii säiliöluokkana solmuille, `Node` sisältää assosiaatiotaulun solmuun lähtevistä ja tulevista kaarista. Solmujen heuristiset kustannukset lasketaan `position`-attribuutin avulla. Varsinainen optimaalisen polun haku on kapseloitu `PathFinder`-luokkaan, jonka keskeisimmät metodit on listattu liitteessä B.

`Graph`-luokka sisältää metodit verkon tietojen tuomiseen ja viemiseen GXL-muodossa. GXL (Graph eXchange Language) on XML-pohjainen tallennusformaatti, joka on suunniteltu verkkomuotoisen tiedon välittämiseen sovellusten välillä [1]. Formaatti on suunnitelmtu mahdollisimman yleiskäyttöiseksi ja sen avulla voi kuvata tavallisten graafien lisäksi hypergraafeja ja sisäkkäisiä graafeja. Lisäksi solmuihin voi liittää mielivaltaisia attribuutteja, tämän sovelluksen tapauksessa nimi- ja paikkatiedot. Esimerkki GXL-tiedoston rakenteesta on liitteessä C. GXL-tietojen lukemiseen käytettiin Python-kielen MiniDOM -XML-jäsennintä.

Luokat `DrawingFrame`- ja `DrawingObject` kuuluvat verkkomallista riippumattomaan käyttöliittymään. Algoritmin toimintaa voi tarvittaessa testata suoraan tekstitilassa PathFinder-luokkaa käyttämällä. Kuvassa olevien luokkien lisäksi sovellukseen kuuluu muutamia apuluokkia ja -funktioita, kuten XML-tiedon kirjoittaja ja vektorilaskennan perusfunktiot.

3.2 Käyttöliittymä ja olemassaolevan koodin käyttö

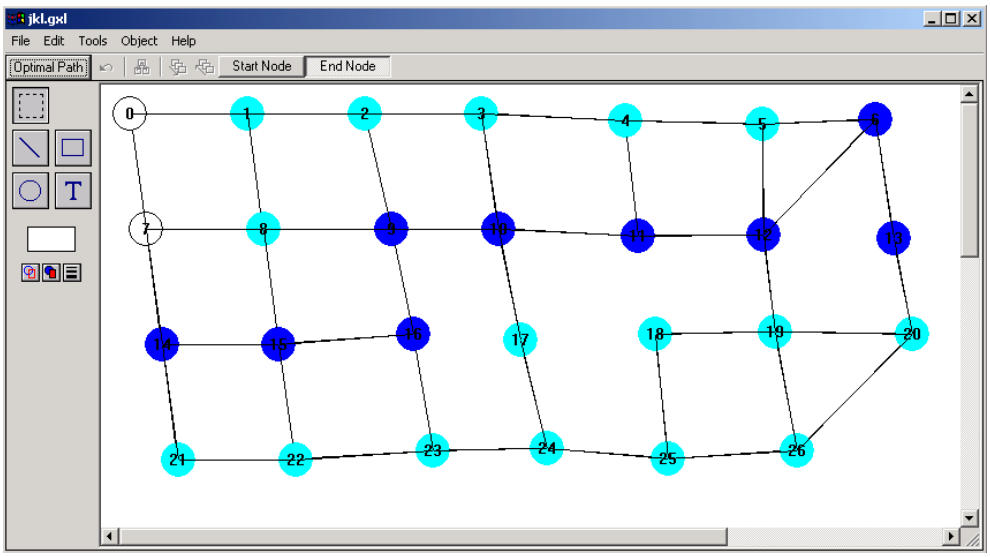
wxPython on käyttöjärjestelmäriippumaton kirjasto graafisten sovellusten kirjoittamiseen. Se on Python-kielinen rajapinta C++ -kieliseen wxWindows -käyttöliittymäkirjastoon. [5]

Sovelluksen käyttöliittymä perustuu wxPython-levityspaketin mukana tulleeseen, Erik Wekstran kirjoittamaan pySketch-piirto-ohjelmaan. Tekijä on ilmoittanut sovelluksen olevan vapaa ja sallii muutosten tekemisen.

Alkuperäinen Sketch-ohjelma mahdollisti viivojen, suorakulmioiden, ellipsien ja tekstien vapaan sijoittelun työtilaan. Lisäksi ohjelma latasi ja tallensi tietoja omassa formaatissaan. Graafeihin liittyviä apuvälineitä ohjelmassa ei kuitenkaan ollut eikä esim. tukea kiinnittää viivoja objekteihin.

Sketch-sovellusta muokattiin niin, että käyttäjän tekemät ellipsit edustavat verkon solmuja ja viivat kaaria. Käyttäjän editoidessa työtilaa verkkomalli muuttuu vastaavasti. Solmujen paikat päätellään työtilan koordinaateista lineaarisella muunnoksella ja niiden siirto onnistuu. Myös GXL-tietojen lataus ja tallennus lisättiin ohjelmaan.

Kuvassa 2 on esitetty muokatun sovelluksen käyttöliittymä optimaalisen polun haun jälkeen.



Kuva 2: Sovelluksen käyttöliittymä.

Optimaalisen polun etsintää varten sovellukseen lisättiin mahdollisuus valita polun alku- ja loppusolmut (`StartNode`- ja `EndNode`-painikkeet työkalupalkissa) sekä A*-algoritmin ajaminen ja näyttäminen askel kerrallaan.

Kuvassa algoritmia on ajettu Jyväskylän keskustan pelkistetyssä mallissa Kalevankatu/Kauppakatu -risteyksestä Vasankatu/Vapaudenkatu -risteykseen. Tummansiniset solmut kuvaavat optimaalista reittiä, turkoosit solmut ovat muut reitin etsimisessä tutkitut solmut. Liitteessä A on kuvattu `PathFinder`-luokat tulostamat viestit haun edetessä.

Lähteet

[1] GXL-kuvauskielen kotisivu
<URL: http://www.gupro.de/GXL/>.

[2] Pasi Koikkalainen, ”Tekoäly” (luentomoniste), Jyväskylän yliopisto, 2001.

[3] Nils Nilsson, ”Artificial Intelligence: A New Synthesis”, Morgan Kaufmann Publishers, 1998.

[4] Python-ohjelmointikielen kotisivu
<URL: http://www.python.org/>.

[5] wxPython-käyttöliittymäkirjaston kotisivu
<URL: http://wxpython.org/>.

Liitteet

A Hakuesimerkki

```
Searching: Node 13 (6.63,1.08)
Searching: Node 20 (6.79,1.91)
Searching: Node 19 (5.60,1.89)
Searching: Node 18 (4.57,1.91)
Searching: Node 6 (6.47,0.05)
Now fixing cost values...
Searching: Node 5 (5.49,0.09)
Searching: Node 4 (4.30,0.06)
Searching: Node 26 (5.79,2.92)
Now fixing cost values...
Searching: Node 12 (5.50,1.05)
Now fixing cost values...
Searching: Node 3 (3.06,0.00)
Searching: Node 11 (4.41,1.06)
Now fixing cost values...
Searching: Node 25 (4.67,2.99)
Searching: Node 24 (3.62,2.90)
Searching: Node 23 (2.63,2.92)
Searching: Node 2 (2.04,0.00)
Searching: Node 10 (3.20,1.00)
Now fixing cost values...
Searching: Node 22 (1.44,3.00)
Searching: Node 17 (3.39,1.96)
Searching: Node 1 (1.02,0.00)
Searching: Node 9 (2.28,1.00)
Searching: Node 8 (1.16,1.00)
Searching: Node 15 (1.30,2.00)
Searching: Node 21 (0.42,3.00)
Searching: Node 14 (0.28,2.00)
Optimal path:
 Node 13 (6.63,1.08) -> Node 6 (6.47,0.05) ->
 Node 12 (5.50,1.05) -> Node 11 (4.41,1.06) ->
 Node 10 (3.20,1.00) -> Node 9 (2.28,1.00) ->
 Node 16 (2.46,1.91) -> Node 15 (1.30,2.00) ->
 Node 14 (0.28,2.00)
```

B A*-toteutuksen ohjelmalistaus

```
"""A* -based optimal pathfinder class - Miika Nurminen, 21.02.2003"""

def __init__(self,nodeList,startNode,endNode):
  """Constructor for PathFinder-class"""
  self.nodeList = nodeList
  self.startNode=startNode
  self.endNode=endNode
  self.countHeuristicCost()
  for x in self.nodeList:
    x.minBeginCost=-1 #-1 stands that node in not searched yet
  self.searchState=0

def countHeuristicCost(self):
  for x in self.nodeList:
    x.length = vectorMath.distance(
      x.position,self.endNode.position)

def newNodeCost(self,oldnode, newnode):
  return oldnode.minBeginCost+vectorMath.distance(
    oldnode.position,newnode.position)

def recursiveBeginCostFix(self,totalList,src,target):
  "Fixes heuristic minimal costs. doesn't sort lists"
  print "Now fixing cost values..."
  target.minBeginCost = self.newNodeCost(src,target)
  for x in target.edgeMap.keys():
    try:
      totalList.index(x)
      if x.minBeginCost > self.newNodeCost(target,x):
        recursiveBeginCostFix(totalList,target,x)
    except:
      continue

def getOptimalPathList(self,node,list):
  """Recursive function returning optimal path based on begincost.
     Takes endnode and an empty list as argument"""
  list.insert(0,node)
  if node==self.startNode:
    return list
  nodes = filter(lambda x:x.minBeginCost>=0,node.edgeMapIn.keys())
  if len(nodes)==0: return [] #no solution
```

```
    newnode = nodes[0]
    for x in nodes:
      if x.minBeginCost< newnode.minBeginCost:
        newnode = x
    return self.getOptimalPathList(newnode,list)

  def beginPathSearch(self):
    """Initialization step for A*-algorithm"""
    self.startNode.minBeginCost=0
    self.open = SortedNodeList()
    self.closed = SortedNodeList()
    self.open.addNodeWithSortedCost(self.startNode)
    self.traversed = []
    self.searchState = 1

  def searchIteration(self):
    """Iteration step for A*-algorithm"""
    assert self.searchState==1
    if len(self.open.nodeList)==0:
      self.searchState=2
      return 1
    n = self.open.nodeList.pop(0) #next node to evaluate
    print "Searching: "+str(n)
    i = self.closed.addNodeWithSortedCost(n)
    self.traversed.append(n)
    if n==self.endNode:
      self.searchState=2
      return 1
    newNodes = []
    totalList = self.open.nodeList + self.closed.nodeList
    for x in n.edgeMap.keys(): # fixing existing path values
      if x.minBeginCost==-1:
        newNodes.append(x)
      if x.minBeginCost > self.newNodeCost(n,x):
        self.recursiveBeginCostFix(totalList,n,x)
    for x in newNodes:
      x.minBeginCost = self.newNodeCost(n,x)
      self.open.addNodeSortedCost(x)

  def endPathSearch(self):
    """Returns optimal path if available"""
    assert self.searchState==2
    return self.getOptimalPathList(self.endNode,[]) #solution
```

C GXL-esimerkki

```
<?xml version="1.0" encoding="iso-8859-1"?>
<gxl>
<graph edgeids="false" id="JKL" edgemode="directed">
<node id="11">
<attr name="label">
<string>11</string>
</attr>
<attr name="x1">
<string>4.3</string>
</attr>
<attr name="x2">
<string>1.06</string>
</attr>
</node>
<edge to="12" isdirected="true" from="11">
</edge>
<edge to="10" isdirected="true" from="11">
</edge>
<node id="10">
<attr name="label">
<string>10</string>
</attr>
<attr name="x1">
<string>3.1</string>
</attr>
<attr name="x2">
<string>1.0</string>
</attr>
</node>
<!-- ... -->
</graph>
</gxl>
```