

Treball pràctic individual

Programació d'un algoritme A^*

Algorítmia i combinatòria en grafs. Mètodes heurístics

Sergi Cantón Simó
1569251

10/6/2020

1 Introducció

L' A^* és un algoritme molt utilitzat en ciències de la computació, el qual permet trobar el camí òptim entre dos nodes donats en grafs on cada aresta o node té un cert pes o cost. Per aquest motiu, el seu ús és freqüent en problemes de cerca de rutes més curtes en mapes de carrers i carreteres, on les distàncies es poden traduir en cost.

El problema que cal resoldre en aquesta pràctica és el següent. Tenim dos fitxers amb la informació d'una sèrie de carrers del centre de Sabadell. Cada carrer es representa com un conjunt de nodes units per arestes. A partir d'aquestes dades, volem implementar un algoritme A^* que calculi el camí més curt entre dos nodes donats.

Per això, necessitem representar la informació emmagatzemada als fitxers `Nodes.csv` i `Carrers.csv` com un graf. A més, és important fer ús de llistes enllaçades i estructures per poder tractar de manera efectiva les dades de les quals disposem.

2 Estructura del codi

El codi de la pràctica es pot dividir en quatre parts. La primera està formada per les dues estructures que encapsulen la informació dels nodes i de les arestes del mapa. La segona està constituïda per una sèrie de funcions i procediments auxiliars, però fonamentals, a l'hora de desenvolupar l' A^* . La tercera part consisteix en la funció que pròpiament realitza l'algorisme A^* . Finalment, la quarta és el `main` o codi principal on es llegeixen els fitxers de dades i s'imprimeix el resultat de l'aplicació de l' A^* al graf.

2.1 Estructures

La principal estructura és `Node`. Com el seu nom indica, aquesta conté tota la informació d'un node. A més, té algunes variables pròpies que permeten que un node sigui un element d'una llista enllaçada, la qual cosa és de gran utilitat a la funció de l'algoritme A^* .

Concretament, les variables de `Node` que podríem anomenar com a generals, és a dir, que no varien segons la ruta que volgüem dur a terme i que s'obtenen a partir de `Nodes.csv`, són les següents.

- **id**: és una variable de tipus `long int`, la qual serveix d'identificador d'un node particular.
- **latitud** i **longitud**: com el seu nom indica, aquestes variables, de tipus `double`, representen la latitud i la longitud del punt del mapa que un node concret simula.
- **numArestes**: és de tipus `int`, i el seu valor és el nombre d'arestes que connecta entre sí el node. Per exemple, si un node és la intersecció de dos carrers, **numArestes** valdrà 4, ja que cal comptar que el node connecta les dues bandes de dos carrers, és a dir, en total 4 arestes.
- **arestes[]**: és el vector d'estructures **Aresta** que conté les estructures relatives a les arestes que connecta el node.

Les altres variables seran detallades i la seva utilitat serà exemplificada a la funció que realitza l'algoritme de l'A*. No obstant, ara en farem una menció.

- **seguent**: és un apuntador a un altre node. Aquesta variable és fonamental per poder tractar un **Node** com a un element d'una llista enllaçada, ja que, si aquest **Node** es troba formant part d'una llista enllaçada, **seguent** apunta a el següent element de la llista, o a `NULL` en cas que el node en sigui l'últim element.
- **cost**: de tipus `double`, aquesta variable guarda el valor de la distància recorreguda des del node inicial d'un camí fins aquest havent seguit, en tot moment, camins d'arestes permesos.
- **idPare**: és una variable de tipus `long int`, la qual conté la identificació del node predecessor a aquest si el node és part d'un camí òptim.

Com ja s'ha comentat, les arestes es guarden en forma d'una estructura **Aresta**. Aquesta conté les següents dues variables.

- **idCarrer[]**: és un vector de 12 posicions de tipus `char[]`, el qual representa l'identificador del carrer en el qual es troba l'aresta.
- **numSeguentNode**: és una variable de tipus `int`, el valor de la qual és la posició del següent node de l'aresta en un vector **nodes**, l'utilitat del qual s'explicarà posteriorment.

2.2 Funcions i procediments auxiliars

En aquesta part s'explicaran totes aquelles funcions i procediments que serveixen per dur a terme tasques com manipular o obtenir informació de llistes enllaçades, extraure informació d'un vector de nodes o realitzar determinats càlculs.

La primera funció és la següent.

- **distancia()**:

Aquesta funció rep com a paràmetres dos apuntadors a **Node**. **distancia()** transforma les coordenades esfèriques, obtingudes a partir de la latitud i la longitud on es troben els dos nodes, a coordenades cartesianes espacials. A partir dels valors trobats, la funció retorna la distància euclídia entre els dos punts.

La següent funció serveix per cercar elements en un vector d'apuntadors a nodes.

- **buscaNode()**:

Rep com a paràmetres l'apuntador a un vector de **Node**, el nombre de nodes al vector, en forma d'`int`, i el número identificador d'un node. La funció busca el node de la llista de nodes

que té la identificació passada per paràmetre mitjançant la cerca binària. Per aquest motiu, és important que els nodes del vector de nodes estiguin ordenats de menor a major `id`. Si es troba el node buscat, el programa retorna l'índex d'aquest en el vector. En cas contrari, la funció retorna `-1` per indicar que no s'ha pogut trobar.

Finalment, les següents cinc funcions permeten obtenir informació i manipular una llista enllaçada.

- **`insereixNodeLlista()`:**

Com el seu nom indica, aquesta funció insereix un element de tipus **`Node`** en una llista enllaçada a la posició que li toca segons la seva `id`. Rep com a paràmetres l'apuntador de l'apuntador que apunta al node inicial de la llista i l'apuntador al node el qual volem afegir a la llista, anomenat **`actual`**. Si la llista enllaçada és buida, **`actual`** en passa a ser l'únic element. Per tant, el **`seguent`** de l'**`actual`** passa a ser **`NULL`** i l'inici de la llista passa a ser **`actual`**. En cas que la llista no sigui buida, es recorre la llista fins que s'arriba a la posició on cal inserir el node. Allà, es fa que **`seguent`** del node anterior a on volem inserir **`actual`** apunti al node **`actual`**. A més, es fa que el **`seguent`** d'**`actual`** passi ara a ser el **`seguent`** del node anterior.

- **`esborraNodeLlista()`:**

Aquesta funció rep com a paràmetres l'apuntador de l'apuntador que apunta al node inicial d'una llista enllaçada i la `id` d'un **`Node`**, de tipus `long int`. La seva utilitat és la d'esborrar el node amb la identificació passada per paràmetre de la llista enllaçada. Si el primer node de la llista ja és el node buscat, el programa fa que l'inici de la llista passi a ser el segon node de la llista. Si no, la funció va recorrent la llista enllaçada fins que acaba o fins que troba un node amb número d'identificació igual al passat com a paràmetre. Si es troba el node buscat, es fa que el **`seguent`** de l'anterior d'aquest node passi a apuntar al **`seguent`** del **`seguent`**. És a dir, el **`seguent`** del node anterior deixa d'apuntar al node que volem esborrar i passa a apuntar al **`seguent`** del node que volem esborrar. En cas que s'hagi recorregut tota la llista i no s'hagi trobat el node buscat, la funció realitza un **`exit(1)`** com a error.

- **`llistaEsBuida()`:**

És una funció molt senzilla. Rep com a paràmetres l'apuntador a un apuntador a un **`Node`**, el qual representa l'inici d'una llista enllaçada. **`llistaEsBuida()`** retorna `1` si l'apuntador al node inicial val **`NULL`** i retorna `0` en cas contrari. És a dir, la funció comprova i retorna si la llista passada com a paràmetre és buida o no.

- **`nodeEsALlista()`:**

D'igual manera que les anteriors funcions, aquesta rep com a primer paràmetre l'inici d'una llista enllaçada. A més, com a segon paràmetre rep l'apuntador a un **`Node`**. Si la llista és buida, la funció directament retorna `0`. Si no, va recorrent tota la llista enllaçada fins que es troba un node de la llista que tingui la mateixa `id` que el node passat com a paràmetre. En aquest cas, la funció retorna `1`. En cas que no es trobi cap node que compleixi la condició anterior, el programa retorna `0`. És a dir, la funció comprova i retorna si la llista passada com a paràmetre conté el node passat, també, com a paràmetre o no el conté.

- **`buscaNodeMenorCost()`:**

Aquesta funció rep l'inici d'una llista enllaçada i l'apuntador a un node anomenat **`nodeFinal`**. La funció retorna la `id` del node de la llista enllaçada el qual tingui la menor suma del seu cost més la distància euclídia d'aquest node al **`nodeFinal`**. Inicialment, es declara la variable,

de tipus `double`, `menorCost` i `idMenorCost`, de tipus `long int`. La primera s'inicialitza a `INFINIT` i, la segona, a `-1`. La funció va recorrent tota la llista enllaçada, i per cada `Node` de la llista, calcula la suma del seu `cost` més la seva distància euclídia fins el `nodeFinal`. Si aquest valor és menor que `menorCost`, `menorCost` passa a ser aquest valor i `idMenorCost` passa a ser la `id` del node pel qual s'està iterant. Finalment, la funció retorna l'identificador del node que ha minimitzat el valor de la suma del seu cost més la seva distància euclídia fins el `nodeFinal`.

2.3 Algoritme A*

Com ja s'ha explicat, l'A* és un algoritme de cerca de camins òptims on cada aresta o node, en aquest cas, té un pes o un cost. Concretament, aquí cada node té dos costos diferents. Primer, té el cost que es guarda a la variable `cost` de l'estructura `Node`. Aquest correspon a la distància recorreguda en arribar fins aquest node anant pel camí d'arestes permeses que minimitza la distància. El segon cost és la distància heurística, la qual representa la distància euclídia aproximada entre el node i el node final, és a dir, el node on volem arribar.

L'algoritme A* es troba encapsulat en la funció `AEstrella()`. Aquesta rep una sèrie de paràmetres, a partir dels quals realitza l'A* i retorna el camí òptim entre els dos punts desitjats en forma d'un vector d'identificadors de nodes.

Els paràmetres que necessita la funció `AEstrella()` són els següents.

- **idNodeInicial:**

De tipus `long int`, correspon a l'identificador del node inicial, és a dir, del node des del qual es vol començar el camí.

- **idNodeFinal:**

De manera anàloga a `idNodeInicial`, aquesta variable guarda la `id` del node fins el qual es vol arribar.

- **nodes:**

Aquesta variable és el vector on estan guardats tots els nodes del mapa, és a dir, tots els nodes pels quals està permès passar.

- **numNodes:**

De tipus `int`, correspon al nombre de nodes que conté el vector `nodes`.

Al començament d'`AEstrella()`, s'inicialitzen algunes variables necessàries per dur a terme l'algoritme A*. Aquestes són les següents, i totes són apuntadors a `Node`.

- **nodeInicial:**

Aquesta variable apunta al node inicial, des d'on comença el camí. La posició de memòria del node inicial s'obté aplicant la funció `buscaNode()` per tal de buscar l'índex del node inicial en el vector `nodes`. A continuació, `nodeInicial` passa a apuntar a la posició de memòria corresponent a l'índex de `nodes` trobat.

- **nodeFinal:**

Apunta al node final, on volem arribar. La posició de memòria on `nodeFinal` ha d'apuntar s'obté de manera anàloga que amb `nodeInicial`.

- **llistaOberta:**

Aquest apuntador representa l'inici d'una llista enllaçada, la qual s'inicialitza a **NULL**. En aquesta llista, posteriorment es guardaran els nodes que han sigut visitats però els successors del qual no han sigut explorats.

- **llistaTancada:**

També és l'inici d'una llista enllaçada que comença valent **NULL**. En aquesta llista, més tard es guardaran els nodes que han sigut visitats i els seus successors han sigut explorats.

- **nodeActual:**

És un apuntador auxiliar. Serveix perquè apunti temporalment al node el qual en un cert moment estem explorant.

- **nodeSuccessor:**

Aquesta variable va apuntant als successors del **nodeActual**. D'aquesta manera, es poden iterar tots els nodes que venen a continuació d'un cert **nodeActual**.

Les variables **costActualSuccessor**, **numNodeSuccessor** i **idNodeMenorCost** guarden valors que són útils més endavant.

Un cop inicialitzades aquestes variables, comença a realitzar-se l'algoritme A^* . Com posteriorment explicarem, el cost de tots els nodes al començar l' A^* és **INFINIT**, és a dir, el màxim **double**.

Inicialment, es posa el cost del node inicial a 0 i s'insereix aquest node a la llista oberta.

A continuació, hi ha un bucle **while**, el qual s'executa mentre la llista oberta no estigui buida. Dins d'aquest bucle, el duen a terme els següents processos.

Per començar, es busca dins la llista oberta el node amb menor suma del seu cost més la distància heurística d'aquest node al node final. El **nodeActual** passa a apuntar a aquest node que, el primer cop que s'executi el **while**, serà **nodeInicial**, ja que és l'únic node de la llista oberta. A més, s'esborra el node agafat de la llista oberta.

Després es comprova si la **id** del **nodeActual** és la mateixa que la del **nodeFinal**. Si la condició es compleix, vol dir que s'ha trobat un camí solució. Per tant, el programa surt automàticament del **while**.

Si no es compleix la condició, el programa passa a executar un bucle **for**, encara dins del **while**. Aquest es repeteix tantes vegades com el valor de **numArestes** del **nodeActual**.

Al començar cada iteració del **for**, es fa que **nodeSuccessor** apunti al següent node de l'aresta que surt del **nodeActual** corresponent al número de la iteració. Per exemple, si **nodeActual** connecta més de tres arestes, a la tercera iteració del **for**, **nodeSuccessor** apunta al següent node de la tercera aresta de **nodeActual**. També, la variable **costActualSuccessor** passa a valer la suma del cost del **nodeActual** més la distància euclídia del **nodeActual** al **nodeSuccessor**.

Posteriorment, es comprova si el **nodeSuccessor** és a la llista oberta. En cas que hi sigui, si el cost del **nodeSuccessor** és menor o igual al valor de **costActualSuccessor**, la iteració actual del **for** finalitza i en comença la següent. Si **nodeSuccessor** no és a la llista oberta però sí que hi és a la llista tancada, es torna a comprovar si el cost del **nodeSuccessor** és menor o igual al valor de **costActualSuccessor** i, en cas que sí, la iteració actual del **for** finalitza i en comença la següent. Si

no, el `node` successor es mou de la llista tancada a la llista oberta. Finalment, si el `nodeSuccessor` no és a cap de les dues llistes, s'insereix a la llista oberta.

Més tard i, encara dins del `for`, el `cost` del `nodeSuccessor` passa a valer `costActualSuccessor` i la variable `idPare`, també del `nodeSuccessor`, es posa al valor de la `id` del `nodeActual`.

Un cop acaba cada iteració del `for`, s'insereix el node `nodeActual` a la llista tancada, degut a que ja han sigut visitats tots els seus successors.

Quan finalitza el bucle `while`, es comprova si el `nodeActual` té la mateixa `id` que el `nodeFinal`. En cas que no, significa que no s'ha trobat cap camí, ja que l'algoritme s'ha acabat sense haver arribat al `nodeFinal`. Aleshores, s'imprimeix per pantalla un missatge d'error i s'executa un `exit(0)`.

Si, en canvi, sí s'ha trobat un camí, el programa ha de guardar en un vector les `id` dels nodes que el formen. Per això, es recorre el camí al revés, fent servir les variables `idPare`. Es comença pel `nodeFinal`. Després, es va al node que té com a `id` el valor de la `idPare` del `nodeFinal`, és a dir, el node anterior de `nodeFinal`. De la mateixa manera, es va al node anterior de l'anterior de `nodeFinal`, i així successivament fins arribar al `nodeInicial`. D'aquesta manera, el programa compta el nombre de nodes que formen el camí. Així, es pot crear un vector `idNodesCamí` amb tantes posicions com el nombre de nodes més 1. Posteriorment, es torna a recórrer la llista de nodes que formen el camí de la mateixa manera que abans, però amb la diferència que, aquest cop, es va guardant a `idNodesCamí` les `id` dels nodes del camí. L'última posició del vector es fa servir per guardar-hi un `-1` per tal que, a l'hora de llegir el vector, se sàpiga on acaba.

Finalment, la funció `AEstrella()` retorna el vector `idNodesCamí`.

2.4 Codi principal

L'anomenat codi principal consisteix en la funció `main()`. Aquesta s'executa quan l'usuari fa servir el programa.

La funció `main()` rep com a arguments el nombre d'arguments passats per línia de comandes (més tard s'especificarà com es passen els arguments), de tipus `int`, i un vector de vectors `char[]`, on es guarden els arguments passats.

Inicialment, al `main()` es creen els apuntadors als fitxers. També, s'inicialitzen les variables `numNodes` i `numCarrers`, les quals guarden el nombre de línies en els fitxers `Nodes.csv` i `Carrers.csv`, respectivament. A més, es crea el vector `nodes`, on es guarden les estructures `Node`; les variables on es guarden les `id` del node inicial i final, `idNodeInicial` i `idNodeFinal`, de tipus `long int` i, finalment, un `char` auxiliar necessari per llegir els fitxers.

Posteriorment, el programa retorna un error en cas que el nombre d'arguments sigui diferent a tres. Si és tres, es transformen els arguments del `main()` corresponents a les `id` dels nodes inicial i final de `char` a `long int` per tal que puguin ser utilitzats correctament i es guarden a `idNodeInicial` i `idNodeFinal`.

Després, s'obre el fitxer `Nodes.csv` per tal de llegir-lo. Si no es troba el fitxer, el programa retorna un error. Primer, es compta el nombre de línies. Segon, es fa un `for` de `numNodes` iteracions, una per cada línia. Per cada línia *i*, es llegeix i es guarda a l'*i*-èssima posició del vector `nodes` la `id`, la `latitud` i la `longitud`. També, s'inicialitza `numArestes` a 0 i `cost` a `INFINIT`. Quan s'ha llegit tot el fitxer, es tanca.

A continuació, el programa obre **Carrers.csv** per extraure'n la informació. Cada línia d'aquest fitxer conté la identificació del carrer i les **id** dels nodes que hi pertanyen. D'igual manera que amb **Nodes.csv**, si no es troba el fitxer, el programa retorna un error. En un **for**, el programa llegeix, per a cada línia, la informació i, amb aquesta, va omplint les estructures **Aresta** del vector **arestes** de cada **Node**. Un cop s'acaba de llegir el fitxer, aquest es tanca.

Posteriorment, el programa crida la funció **AEstrella()** passant-li tots els paràmetres necessaris i el resultat es guarda al vector **camí**.

Finalment, partir del vector **camí**, s'imprimeix per pantalla la informació de tots els nodes que conformen el camí que va del node inicial al node final, la **id** dels quals és la que ha sigut passada com a paràmetre a la funció **main()**.

3 Execució del programa

3.1 Instruccions d'execució

El programa està pensat per ser executat per línia de comandes en un terminal de Linux. Per començar s'ha de compilar el fitxer **AEstrella.c** amb la comanda

```
gcc -o AEstrella AEstrella.c -lm
```

per generar l'executable **AEstrella**.

Després, cal escriure en un terminal de Linux **./AEstrella**, seguit dels identificadors del node de partida i del node al qual volem arribar, separats per un espai. La sintaxi d'execució, llavors, és de la següent manera.

```
./AEstrella idNodeInicial idNodeFinal
```

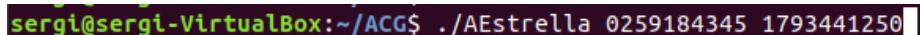
on **idNodeInicial** és el número identificador del node inicial i **idNodeFinal** és l'identificador del node final.

Un cop s'executa el programa, es mostra per pantalla la distància que es recorre en arribar del node inicial al final. A sota, també s'imprimeix la informació de tots els nodes pels quals cal passar en el camí òptim del node inicial al final.

3.2 Exemple d'execució

A continuació, es mostrarà un exemple d'ús del programa **AEstrella**.

Si un usuari volgués trobar el camí més curt per anar del node amb identificador **0259184345** fins el node amb identificador **1793441250**, hauria d'executar la comanda que es veu en la figura 1 en un terminal de Linux.



```
sergi@sergi-VirtualBox:~/ACG$ ./AEstrella 0259184345 1793441250
```

Figura 1: Comanda per executar el codi en un terminal de Linux.

Un cop executada, el programa mostraria per pantalla la informació de tots els nodes que formen el camí òptim del node d'inici al node final. El resultat es veu a la figura 2.

```
sergi@sergi-VirtualBox:~/ACG$ ./AEstrella 0259184345 1793441250

La distancia de 0259184345 a 1793441250 es de 507.886803 metres.
Camí optim:
-----
Id=0259184345 | 41.545380 | 2.106830 | Dist=0.000000
Id=0259437888 | 41.545752 | 2.106744 | Dist=42.042028
Id=0259437905 | 41.546388 | 2.106495 | Dist=115.722403
Id=0259438253 | 41.546734 | 2.107858 | Dist=235.476557
Id=0965459173 | 41.546963 | 2.107746 | Dist=262.617110
Id=0960085142 | 41.547169 | 2.107648 | Dist=286.900380
Id=1944921315 | 41.547281 | 2.107553 | Dist=301.623075
Id=2412854895 | 41.547777 | 2.107092 | Dist=368.879521
Id=1944921533 | 41.548161 | 2.107668 | Dist=433.058491
Id=1944921536 | 41.548240 | 2.107798 | Dist=446.899874
Id=1944921547 | 41.548280 | 2.107864 | Dist=454.047082
Id=1793441253 | 41.548396 | 2.108055 | Dist=474.495886
Id=1944921549 | 41.548399 | 2.108073 | Dist=476.041257
Id=1955175329 | 41.548407 | 2.108110 | Dist=479.200109
Id=1955175330 | 41.548452 | 2.108329 | Dist=498.154671
Id=1793441250 | 41.548481 | 2.108440 | Dist=507.886803
-----
sergi@sergi-VirtualBox:~/ACG$
```

Figura 2: Resultat de l'execució del programa.

Si l'usuari introdueix dades incorrectes, no col·loca els fitxers necessaris correctament o es produeix qualsevol altre error que causa un mal funcionament del programa, l'execució s'atura i es mostra un missatge d'error.

Per exemple, si l'usuari decidís executar el codi introduint un número d'identificació d'un node que no existeix, es mostraria el missatge de la figura 3 i s'aturaria l'execució.

```
sergi@sergi-VirtualBox:~/ACG$ ./AEstrella 0259184345 -3
ERROR. No s'han trobat els nodes amb les id introduïdes.
sergi@sergi-VirtualBox:~/ACG$
```

Figura 3: Resultat de l'execució del programa introduint la identificació d'un node inexistent.