



Universitat de Lleida

Pac-Man

Segura Paz, Aleix

Serrano Ortega, Aniol

Data: 06/11/2023

Pràctica 1

Intel·ligència Artificial

Escola Politècnica Superior

Índex

| | |
|---|----------|
| 1. Introducció | 1 |
| 2. Implementació Algorismes | 1 |
| 2.1. <i>Depth-First Search</i> (DFS) | 1 |
| 2.2. <i>Breadth-First Search</i> (BFS) | 1 |
| 2.3. <i>Uniform Cost Search</i> (UCS) | 2 |
| 2.4. <i>A-Star</i> | 2 |
| 3. Heurístiques | 3 |
| 3.1. <i>Corners Problem</i> | 4 |
| 3.2. <i>Food Search Problem</i> | 4 |
| 3.3. Comparativa heurístiques | 5 |
| 3.3.1. Anàlisis <i>cornersHeuristic</i> | 5 |
| 3.3.2. Anàlisis <i>foodHeuristic</i> | 6 |
| 4. Conclusions | 6 |

Índex de Taules

| | | |
|---------|--|---|
| Taula 1 | Comparativa de costos y nodes <i>expanded</i> entre UCS i A* | 5 |
| Taula 2 | Comparativa de temps i nodes <i>expanded</i> entre UCS i A* | 6 |

1. Introducció

En aquest informe, es detallen diferents algorismes de cerca implementats per al joc del *Pac-Man*. A través de la implementació d'aquests algorismes (DFS, BFS, UCS i A*) es tracta de comprendre el seu funcionament i poder comparar-los entre si. D'altra banda, es proporciona una descripció i comparativa de rendiment de les heurístiques desenvolupades per a estimar el cost restant per arribar a l'objectiu.

2. Implementació Algorismes

2.1. *Depth-First Search* (DFS)

Per a la implementació del DFS s'han considerat els següents punts de disseny:

- S'utilitza una estructura LIFO (*Last In First Out*) com a *fringe*, això permet explorar en profunditat, ja que sempre s'explora l'últim node introduït en l'estructura quan s'ha acabat d'expandir un.
- Abans d'entrar al bucle es comprova si el node inicial és ja solució.
- Quan obtenim els successors del node que estem explorant, es comprova que no estigui ni a *expanded* ni a *fringe*.
- Es comprova si un node és solució en el moment que el veiem per primer cop, és a dir, quan el crea el node pare.

2.2. *Breadth-First Search* (BFS)

Per a la implementació del BFS s'han considerat els següents punts de disseny:

- S'utilitza una estructura FIFO (*First In First Out*) com a *fringe*, això permet explorar en amplada ja es va explorant per nivells i no s'explora un node del nivell següent fins que s'ha explorat el actual.
- Abans d'entrar al bucle es comprova si el node inicial és ja solució.

- Quan obtenim els successors del node que estem explorant, es comprova que no estigui ni a *expanded* ni a *fringe*.
- Es comprova si un node és solució en el moment que el veiem per primer cop, és a dir, quan el crea el node pare.

2.3. *Uniform Cost Search* (UCS)

Per a la implementació del UCS s'han considerat els següents punts de disseny:

- S'utilitza una estructura *Priority Queue* com a *fringe*, ja que permet que sempre explorem el node amb millor cost acumulat.
- Es comprova si un node és solució just al fer *fringe.pop()* per assegurar que sigui la solució òptima (millor cost).
- Es comprova que el node a explorar no estigui en *expanded*.
- Es comprova que els nodes fill no estiguin en *expanded* quan es creen, si no ho estan s'introdueixen a *fringe*.
- La funció *update* de *fringe* s'encarrega de reemplaçar el mateix node si està a *fringe* i ara el visitem amb un cost menor.

2.4. *A-Star*

Per a la implementació del A-Star s'han considerat els següents punts de disseny:

- S'utilitza una estructura *Priority Queue* com a *fringe*, ja que permet que sempre explorem el node amb millor cost acumulat (cost del node + cost de l'heurística per a aquell node).
- Es comprova si un node és solució just al fer *fringe.pop()* per assegurar que sigui la solució òptima (millor cost).
- Es comprova que el node a explorar no estigui en *expanded*.

- Es comprova que els nodes fill no estiguin en *expanded* quan es creen, si no ho estan s'introdueixen a *fringe*.
- La funció *update* de *fringe* s'encarrega de reemplaçar el mateix node si està a *fringe* i ara el visitem amb un cost menor.

3. Heurístiques

La implementació de l'algorisme de A^* requereix funcions que calculin l'heurística per a cada node. Per defecte, usen una heurística ja implementada al projecte que és trivial, però l'objectiu és implementar una heurística no trivial, admissible i monòtona, és a dir, consistent.

Una heurística és admissible si per a cada node l'heurística no sobreestima el cost del camí òptim a l'objectiu. Per aquest motiu, l'estratègia d'una heurística admissible és la d'eliminar restriccions al problema original, com en aquest cas, eliminar la restricció que les parets no poden ser travessades.

D'altra banda, perquè l'heurística sigui consistent ha de ser monòtona. Això implica que per a tot parell de nodes el cost del camí d'un node al seu successor més l'heurística d'aquest successor ha de ser major igual a l'heurística del primer node. En una expressió més formal, la condició que ha de complir és la següent:

$$h(n) \leq c(n, a, n') + h(n')$$

- n : Node actual
- n' : Node successor
- a : Cost de n a n'

Finalment, per a implementar aquesta heurística s'ha implementat en dues funcions *cornersHeuristic* i *FoodSearchProblem* en el fitxer `searchAgents.py`.

3.1. *Corners Problem*

En el cas de l'heurística per a les cantonades (*cornersHeuristic*), l'objectiu és proporcionar una estimació del cost restant per a arribar a la meta del problema. Més específicament, la meta és visitar totes les cantonades del mapa i retornar l'estimació d'aquest cost.

En primer lloc, s'han d'identificar les cantonades no visitades fent ús del segon paràmetre de l'estat. En cas que una cantonada no estigui visitada dins de la llista de *corners*, aquesta cantonada es guarda en una llista de les cantonades no visitades. En el cas que no existeixin cantonades no visitades, llavors totes les cantonades han estat visitades i l'heurística retorna 0.

Seguidament, es calculen les distàncies des de la posició actual a totes les cantonades no visitades emprant la funció de la distància *Manhattan* definida en l'arxiu `util.py`. D'aquesta manera, es troba la cantonada més propera i la seva distància.

Finalment, des de la cantonada més propera es calculen les distàncies cap a les altres cantonades no visitades, les quals es van acumulant en la variable `corners_heuristic` fins que es retorna aquest valor. Aquesta heurística es pot provar amb la següent comanda:

```
$ python3 pacman.py -l <layout> -p AStarCornersAgent -z 0.5
```

3.2. *Food Search Problem*

L'objectiu de l'heurística de *foodHeuristic* és estimar la distància des de la posició actual fins a tots els punts de menjar restants en el mapa. Primerament, es revisa si tot el menjar ha estat ja menjat, és a dir, no queda menjar i, per tant, es retorna 0. Seguidament, es calcula la distància de *Manhattan* des de la posició actual fins a cada punt de menjar. A continuació, es troba el menjar més proper i es calcula la distància acumulada des d'aquest menjar fins a altres menjars, de forma similar a la implementació de *cornersHeuristic*. Finalment, se suma la distància en la variable *food_heuristic* per a cada iteració i es retorna al final de les iteracions.

Per a provar aquesta heurística es pot fer mitjançant la següent comanda:

```
$ python3 pacman.py -l <layout> -p AStarFoodSearchAgent
```

3.3. Comparativa heurístiques

Per a comprovar les heurístiques s'han realitzat una sèrie de proves amb l'heurística trivial i s'ha fet una comparació dels nodes expandits i dels costos.

3.3.1. Anàlisi *cornersHeuristic*

Per la heurística de les cantonades s'ha provat amb els mapes de *tinyCorners*, *mediumCorners* i *bigCorners*. Per a UCS, A* i *cornersHeuristic*. Per tant, s'han introduït les següents comandes per a cada *layout* (mapa):

```
$ python3 pacman.py -l <layout> -p SearchAgent -a fn=ucs
```

```
$ python3 pacman.py -l <layout> -p AStarCornersAgent
```

A partir de l'execució d'aquestes comandes, s'ha elaborat la següent taula:

| Algorismes | Mapes | Cost | Nodes <i>Expanded</i> |
|------------|----------------------|------|-----------------------|
| UCS | <i>tinyCorners</i> | 7 | 20 |
| | <i>mediumCorners</i> | 18 | 69 |
| | <i>bigCorners</i> | 36 | 383 |
| A* | <i>tinyCorners</i> | 28 | 154 |
| | <i>mediumCorners</i> | 106 | 692 |
| | <i>bigCorners</i> | 162 | 1725 |

Taula 1: Comparativa de costos y nodes *expanded* entre UCS i A*

A partir d'aquesta taula, es pot veure que amb el *AStarCornersAgent* tant el cost com els nodes expandits han estat considerablement superiors als de l'heurística trivial. D'altra banda, val a dir que aquest algorisme ha estat l'únic que ha acabat els mapes, llavors és lògic que generi més nodes i el cost sigui major.

3.3.2. Anàlisis *foodHeuristic*

Per la heurística del menjar s'ha provat amb els mapes que no tenen cap fantasma i hi ha més d'un menjar. Per a mostrar el temps d'execució s'ha modificat una línia de `searchAgents.py` (`registerInitialState`) per a tenir més precisió del temps (`%.1f` -> `%.3f`).

Les comandes que s'han introduït són les següents:

```
$ python3 pacman.py -l <layout> -p SearchAgent -a fn=ucs
$ python3 pacman.py -l <layout> -p AStarFoodSearchAgent
```

A partir de l'execució d'aquestes comandes, s'ha elaborat la següent taula:

| Mapes | Temps | | Nodes expanded | |
|---------------------|-------|--------|----------------|-------|
| | UCS | A* | UCS | A* |
| <i>smallSearch</i> | 0,003 | 3,541 | 38 | 33020 |
| <i>tinySearch</i> | 0,013 | 0,131 | 24 | 3226 |
| <i>testSearch</i> | 0,002 | 0,0003 | 6 | 13 |
| <i>trickySearch</i> | 0,067 | 1,269 | 59 | 12259 |

Taula 2: Comparativa de temps i nodes *expanded* entre UCS i A*

4. Conclusions

Per a realitzar la implementació dels algorismes primerament descrits, s'ha hagut de superar una sèrie d'obstacles, però el principal ha estat compenmdre el context del joc i quines eines de les que es disposen. Amb l'ajuda del pseudocodi de les transparències s'ha pogut realitzar una estimació de la sol·lució per a cada algorisme, i a base de realitzar iteracions sobre el problema s'ha aconseguit implementar aquests algorismes.