Tema 1 – Căutare în spațiul stărilor

Deadline: 11.11.2019, 23:59

Mihai Nan

Inteligență artificială (Anul universitar 2019-2020)

1 Obiective

Scopul acestei teme este rezolvarea unei probleme de optimizare, modelată ca o problemă de căutare în spațiul stărilor, prin implementarea unor strategii de căutare.

Aspectele urmărite sunt:

- alegerea unei metode de reprezentare a datelor problemei;
- găsirea unor funcții euristice pentru evaluarea stărilor problemei;
- analizarea particularităților diverselor strategii de căutare;
- determinarea eficienței soluției determinate pentru fiecare tip de strategie analizată;
- abstractizarea procesului de rezolvare;
- realizarea unei analize comparative între proprietățile strategiilor implementate.

2 Problema propusă

2.1 Enunțul problemei

Definim următoarea problemă de căutare în spațiul stărilor: Avem un grid ce conține N linii și M coloane. În cadrul acestui grid sunt poziționate K persoane și un taxi, pentru fiecare dintre acestea cunoscându-se celula din grid în care sunt plasate. Pentru fiecare din cele K persoane cunoaștem destinația în care acestea trebuie să ajungă și suma pe care o oferă pentru a ajunge la destinație. Taximetristul are la dispoziție o cantitate limită de benzină și dorește să strângă cât mai mulți bani din transportarea persoanelor. Considerăm că, de fiecare dată când trece dintr-o celulă în alta, taximetristul consumă un litru de benzină. El primește suma de bani de la o persoană doar dacă aceasta este lăsată la destinație. La orice moment de timp, taximetristul nu poate avea mai mult de un client în mașină.

Taximetristul poate alege, la fiecare moment de timp, una dintre următoarele acțiuni:

- `south` – se va deplasa o poziție la sud, dacă acest lucru este posibil;
- `north` – se va deplasa o poziție la nord, dacă acest lucru este posibil;
- `east` – se va deplasa o poziție la est, dacă acest lucru este posibil;
- `west` – se va deplasa o poziție la vest, dacă acest lucru este posibil;
- `pickup` – dacă se află într-o celulă în care există o persoană, va prelua clientul respectiv;
- `dropoff` – dacă are un client și a ajuns în poziția destinație a acestuia, atunci îl va lăsa pe client și va primi banii de la acesta.

După cum se poate observa în Figura 1, există celule în care setul de acțiuni este restrâns. În cazul configurației din exemplul prezentat, taximetristul poate selecta una dintre următoarele acțiuni posibile: `north`, `east` sau `south`.

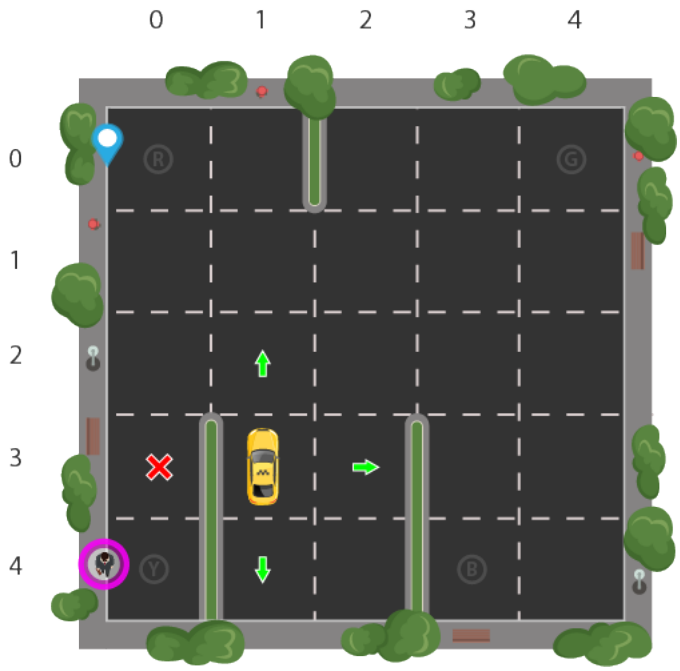

Figure 1: Exemplu de configurație pentru problema propusă

2.2 Evaluarea soluției

Soluția problemei este reprezentată sub forma unei liste de acțiuni. Pentru a evalua o astfel de soluție, pornim din starea inițială și aplicăm lista respectivă de acțiuni. După aplicarea unei acțiuni care presupune modificarea celulei în care se află mașina, este consumată o cantitate de 1 litru de benzină din rezervorul mașinii. De fiecare dată când un client este lăsat în poziția destinație, este colectată suma de bani pe care acesta o oferea pentru cursă. Astfel, pentru determinarea profitului obținut pentru soluția propusă, vom calcula suma banilor încasați de la clienții lăsați la destinație la care vom aduna cantitatea de benzină rămasă în rezervorul mașinii.

2.3 Date de intrare

Datele de intrare ale problemei vor fi furnizate într-un fișier text, având următoarea formă:

- pe prima linie se vor găsi trei numere naturale, separate printr-un spațiu, semnificând dimensiunile gridului (`height`, `width`) și cantitatea limită de benzină a rezervorului mașinii (`capacity`);
- a doua linie conține două numere naturale (`carPositionY, carPositionX`), separate printr-un spațiu, reprezentând poziția de start a mașinii;
- a treia linie conține numărul de posibili clienți, `N`;
- pe următoarele `N` linii se găsesc câte 5 numere naturale, separate printr-un spațiu, având următoarea semnificație: `startY, startX` – poziția de start a clientului, `destinationY, destinationX` – poziția destinație în care dorește să ajungă clientul, `budget` – suma pe care este dispus clientul să o ofere pentru cursă;
- în finalul fișierului se va oferi configurația gridului dată sub următoarea formă (exemplu de codificare pentru gridul din Figura 1):

```
+----------+
| : | : : |
| : : : : |
| : : : : |
| | : | : |
| | : | : |
+----------+
```

Observație

Obstacolul vertical este reprezentat folosind caracterul `|`, iar cel orizontal apare **doar** pe marginea gridului și este codificat folosind caracterul `-`.

2.4 Date de ieșire

Date de ieșire vor fi furnizate tot în cadrul unui fișier text, sub următorul format:

- pe prima linie se va preciza numele strategiei aplicate pentru rezolvarea problemei;
- pe a doua linie se va afla timpul de execuție necesar pentru determinarea soluției;
- pe a treia linie se va scrie numărul de stări vizitate;
- pe a patra linie se va găsi costul asociat soluției obținute;
- în continuare se va scrie soluția identificată sub forma unei liste de acțiuni.

3 Cerințe

3.1 Cerința 1 – 1 punct

Pentru această cerință, trebuie să citiți informațiile furnizate, să vă alegeți o reprezentare pe care să o folosiți pentru starea problemei propuse și să implementați funcțiile de care aveți nevoie pentru evaluarea unei stări, tranziția între stări, determinarea acțiunilor posibile care se pot aplica într-o stare și tot ceea ce mai considerați că ar fi necesar pentru reprezentarea și prelucrarea datelor problemei.

3.2 Cerința 2 – 2 puncte

Implementați următoarele strategii de căutare neinformată: **Breadth First Search** (5.1), **Uniform Cost Search** (5.2), **Depth First Search** (5.3), **Depth-limited Search** (5.4) și **Iterative deepening search** (5.5). Aplicați aceste strategii pentru rezolvarea problemei date.

3.3 Cerința 3 – 1 punct

Implementați o funcție care primește ca argumente rezultatul furnizat de strategia de căutare și determină o soluție a problemei sub forma unei secvențe de acțiuni care conduc la atingerea stării scop găsită de strategia de căutare. De asemenea, funcția va trebui să determine costul soluției.

3.4 Cerința 4 – 3 puncte

Implementați următoarele strategii de căutare informată: **Greedy Best-First Search** (6.1), **A* Search** (6.2) și **Hill-Climbing Search** (6.3). Aplicați aceste strategii pentru rezolvarea problemei date.

3.5 Cerința 5 – 1 punct

Alegeți una dintre cele două variante de optimizare propuse (**Prevenirea explorării căilor care conțin cicluri** (7.1) sau **Prevenirea revizitării unei stări**) (7.2) și optimizați toți algoritmii de căutare implementați folosind această variantă.

3.6 Cerința 6 – 2 puncte

Realizați o analiză comparativă între strategiile implementate, specificând pentru fiecare algoritm în parte care sunt avantajele identificate și limitările sale. De asemenea, specificați cum a influențat tehnica de optimizare aleasă performanțele algoritmului (timp de execuție / valoarea soluției). Definiți cel puțin două funcții euristice de evaluare a unei stări și explicați cum influențează acestea rezultatele algoritmilor euristici.

Pentru rezolvarea acestei cerințe veți realiza un fișier `README`, în format `pdf`, în care veți prezenta în mod explicit:

- avantajele și dezavantajele identificate pentru fiecare strategie în parte;

- cum a afectat aplicarea optimizării modul de rulare și rezultatul obținut pentru fiecare strategie de căutare implementată;
- descrierea celor două funcții euristice considerate și ce rezultate au fost obținute (pentru fiecare strategie se va realiza câte un grafic pentru timpul de execuție și unul pentru calitatea soluției obținute în care se vor evidenția rezultatele obținute pentru scenariile propuse de varianta cu prima funcție euristică și cea care folosește a doua funcție euristică);
- grafice în care să prezentați cum variază timpul de execuție pentru fiecare scenariu propus, în funcție de strategia de căutare folosită;
- grafice în care să prezentați cum variază calitatea soluției obținute (calculată după cum este precizat în 2.2) pentru fiecare scenariu propus, în funcție de strategia de căutare folosită;
- pentru fiecare strategie se va realiza un grafic în care se va prezenta cum diferă numărul de stări vizitate în funcție de variantă (fără optimizare vs cu optimizare) în cazul fiecărui scenariu propus.

3.7 Bonus – 2 puncte

Se acordă maximum 2 puncte bonus pentru:

- implementarea strategiei de căutare **Learning Real-Time A*** [1], aplicarea acesteia pentru problema propusă și compararea acesteia cu strategia **A* Search** (1 punct);
- implementarea strategiei de căutare **Recursive Best-First Search** [2], [3], aplicarea acesteia pentru problema propusă și compararea acesteia cu strategia **Greedy Best-First Search** (1 punct).

4 Noțiuni introductive

Una dintre clasele de probleme cel mai întâlnite în cazul problemelor cu o mare aplicabilitate practică este reprezentată de problemele de optimizare. Problemele din această categorie pot fi caracterizate prin: spațiul stărilor, criteriul de optimizarea și restricțiile impuse de problemă.

Dacă dorim să modelăm o problemă de optimizare sub forma unei probleme de căutare în spațiul stărilor, trebuie să specificăm următoarele:

- **starea inițială**: $s_0 \in S$, unde S este mulțimea stărilor posibile ale problemei;
- **funcția de tranziție**: $succ : S \to \mathcal{P}(S)$, unde $\mathcal{P}(S)$ reprezintă mulțimea părților mulțimii S;
- **o aserțiune care verifică dacă o stare este sau nu finală**: $goal : S \to \{0, 1\}$

De obicei, atunci când realizăm o căutare pentru o astfel de problemă construim un **arbore de căutare**. Pornim din starea inițială care o să reprezinte rădăcina arborelui și continuăm expandând nodurile, folosind funcția *succ* pentru a general descendenții unui nod. Nodurile din acest arbore, pot fi de două tipuri:

1. **noduri Open**: nodurile care au fost generate și adăugate în arborele de căutare, dar nu au fost încă expandate complet;
2. **noduri Closed**: nodurile care au fost adăugate în arborele de căutare și sunt expandate complet.

Strategia de căutare reprezintă modalitatea prin care nodurile din arborele de căutare sunt expandate. Strategiile de căutare se împart în două mari categorii:

1. **strategii de căutare neinformată (*căutare oarbă*)**;
2. **strategii de căutare informată (*căutare euristică*)**.

Unul dintre algoritmii cei mai naivi care pot fi folosiți pentru rezolvarea unei astfel de probleme este următorul, el putând fi considerat drept un algoritm ce implementează o strategie generală de căutare:

Algorithm 1 Algoritmul general de căutare [2]

```
procedure SEARCH(s0, succ, goal)
    open ← {s_0}
    π(s_0) ← Nil
    while open ≠ ∅ do
        current ← extract(open)
        if goal(current) then return (current, π)
        for next ∈ succ(current) do
            π(next) ← current
            insert(next, open)
    return fail
```

5 Strategii de căutare neinformată

5.1 Breadth-first search (BFS)

Algorithm 2 Algoritmul Breadth-first search (BFS)

```
procedure BREADTHFIRSTSEARCH(s0, succ, goal)
    open ← {s_0}
    π(s_0) ← Nil
    while open ≠ ∅ do
        current ← extractHead(open)
        if goal(current) then return (current, π)
        for next ∈ succ(current) do
            π(next) ← current
            insertBack(next, open)
    return fail
```

5.2 Uniform cost search

Algorithm 3 Algoritmul Uniform cost search

```
procedure UNIFORMCOSTSEARCH(s0, succ, goal)
    open ← {s_0}
    π(s_0) ← Nil
    while open ≠ ∅ do
        current ← extractHead(open)
        if goal(current) then return (current, π)
        for next ∈ succ(current) do
            π(next) ← current
            cost ← g(next)
            insertSortedBy(cost, next, open)
    return fail
```

5.3 Depth-first search (DFS)

Algorithm 4 Algoritmul Depth-first search (DFS)

procedure DepthFirstSearch($s0, succ, goal$)
  $open \leftarrow \{s_0\}$
  $\pi(s_0) \leftarrow Nil$
  while $open \neq \emptyset$ **do**
    $current \leftarrow extractHead(open)$
    if $goal(current)$ **then return** $(current, \pi)$
    for $next \in succ(current)$ **do**
      $\pi(next) \leftarrow current$
      $insertFront(next, open)$
  return fail

5.4 Depth-limited search

Algorithm 5 Algoritmul Depth-limited search

procedure DepthLimitedSearch($s0, succ, goal$, k)
  $open \leftarrow \{s_0\}$
  $\pi(s_0) \leftarrow Nil$
  while $open \neq \emptyset$ **do**
    $current \leftarrow extractHead(open)$
    if $goal(current)$ **then return** $(current, \pi)$
    if $depth(current) \geq k$ **then continue**
    for $next \in succ(current)$ **do**
      $\pi(next) \leftarrow current$
      $insertFront(next, open)$
  return fail

5.5 Iterative deepening search

Algorithm 6 Algoritmul Iterative deepening search

procedure IterativeDeepeningSearch($s0, succ, goal$)
  for $k \leftarrow 0$ **to** ∞ **do**
    $result \leftarrow$ depthLimitedSearch$(s_0, succ, goal, k)$
    if $result \neq fail$ **then**
      return $result$

Important

Pentru strategia de căutare neinformată care ține cont de cost, veți folosi o funcție de cost `g(state)` care reprezintă costul drumului de la starea inițială până la nodul corespunzător lui `state`.

6 Strategii de căutare informată

Important

Algoritmii de căutare euristică utilizează o informație suplimentară pentru găsirea soluției problemei. Această informație este reprezentată sub forma unei funcții `h`, unde `h(state)` reprezintă costul drumului estimat de la nodul corespunzător stării `state` până la cea mai apropiată soluție. Funcția `h` poate fi definită în orice mod, existând o singură constrângere:

$$h(state) = 0, \forall state, isGoal(state) == 1$$

6.1 Greedy best-first search

Algorithm 7 Algoritmul Greedy best-first search

```
procedure GreedyBestFirstSearch(s0, succ, goal, h)
    open ← {s_0}
    π(s_0) ← Nil
    while open ≠ ∅ do
        current ← extractHead(open)
        if goal(current) then return (current, π)
        for next ∈ succ(current) do
            cost ← h(current)
            π(next) ← current
            insertSortedBy(cost, next, open)
    return fail
```

6.2 A* search

Algorithm 8 Algoritmul A* search

```
1:  procedure A*Search(s0, succ, goal, h)
2:      open ← {s_0}
3:      π(s_0) ← Nil
4:      closed ← ∅
5:      while open ≠ ∅ do
6:          current ← extractHead(open)
7:          if goal(current) then return (current, π)
8:          closed ← closed ∪ {current}
9:          for next ∈ succ(current) do
10:             cost ← h(current) + g(current)
11:             if ∃next' ∈ closed ∪ open such that state(next) = state(next') then
12:                 if g(next') < g(next) then
13:                     continue
14:                 else
15:                     remove(next', open ∪ closed)
16:             π(next) ← current
17:             insertSortedBy(cost, next, open)
18:     return fail
```

6.3 Hill-climbing search

Algorithm 9 Algoritmul Hill-climbing search

```
procedure HillClimbingSearch(s0, succ, h)
    current ← s0
    π(s0) ← Nil
    done ← False
    currentCost ← g(current) + h(current)
    while done = False do
        maxim ← currentCost
        nextState ← Nil
        for next ∈ succ(current) do
            cost = g(next) + h(next)
            if cost > maxim then
                maxim ← cost
                nextState ← next
        if nextState = Nil then
            done ← True
        else
            π(nextState) ← current
            current ← nextState
            currentCost ← maxim
    return (current, π)
```

Acest algoritm corespunde unei strategii de căutare care se orientează permanent în direcția creșterii valorii soluției, terminându-se atunci când se ajunge într-o stare care corespunde unui maxim. Cu alte cuvinte, niciun succesor nu are valoare mai mare.

7 Optimizări

7.1 Prevenirea explorării căilor care conțin cicluri

Algorithm 10 Algoritmul general de căutare optimizat (varianta 1)

```
procedure Path(state, π)
    parent ← π(state)
    if parent = Nil then
        return {state}
    return Path(parent, π) ∪ {state}
procedure Search(s0, succ, goal)
    open ← {s0}
    π(s0) ← Nil
    while open ≠ ∅ do
        current ← extract(open)
        if goal(current) then return (current, π)
        for next ∈ succ(current) do
            if next ∈ Path(current, π) then continue
            π(next) ← current
            insert(next, open)
    return fail
```

7.2 Prevenirea revizitării unei stări

Algorithm 11 Algoritmul general de căutare optimizat (varianta 2)

```
procedure SEARCH(s0, succ, goal)
    open ← {s_0}
    π(s_0) ← Nil
    visited ← ∅
    while open ≠ ∅ do
        current ← extract(open)
        if goal(current) then return (current, π)
        visited ← visited ∪ {current}
        for next ∈ succ(current) do
            if next ∈ visited then continue
            π(next) ← current
            insert(next, open)
    return fail
```

Referințe

[1] Richard E Korf. Real-time heuristic search. *Artificial intelligence*, 42(2-3):189–211, 1990. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.1955&rep=rep1&type=pdf`.

[2] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[3] Matthew Hatem, Scott Kiesel, and Wheeler Ruml. Recursive best-first search with bounded overhead. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.