Tema 1 – Strategii de căutare

Deadline: X.Y.2021, 23:59

Mihai Nan

Inteligență artificială (Anul universitar 2021-2022)

1 Obiective

Scopul acestei teme este rezolvarea unei probleme în două variante:

- modelată ca o problemă de căutare în spațiul stărilor, prin implementarea unor strategii de căutare;
- modelată ca o problemă de satisfacere a constrângerilor.

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;
- abstractizarea procesului de rezolvare;
- realizarea unei analize comparative între proprietățile strategiilor implementate.

2 Problema propusă

2.1 Enunțul problemei

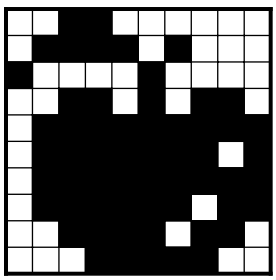
Nonogramele sunt puzzle-uri logice care în urma rezolvării dezvăluie diferite imagini. Soluția este dată de către colorarea sau nu a unor celule dintr-un grilă pe baza interpretării logice a numerelor prezente la începutul fiecărui rând și coloane.

Pentru a rezolva un astfel de puzzle, pornim de la următoarele:

Definiția problemei

Se dă o grilă formată din pătrate care trebuie să fie înnegrite sau lăsate libere. Lângă fiecare linie se află lungimile șirurilor de pătrate negre de pe linia respectivă. Deasupra fiecărei coloane se află lungimile șirurilor de pătrate negre de pe acea coloană.

	1	1 4	2 6	2 7	1 6	8	1 4 1	4 2	2 3	4
2										
4 1										
1 1										
2 1 2										
9										
7 1										
9										
6 2										
4 2										
5										



2.2 Date de intrare

Datele de intrare ale problemei vor fi furnizate într-un fișier `json`, având următoarele câmpuri:

- `name` – numele testului;
- `height` – înălțimea grilei (ca număr de celule);
- `width` – lățimea grilei (ca număr de celule);
- `rows` – vom avea o listă ce conține ca elemente liste cu valorile numerice corespunzătoarea fiecărei linii;
- `columns` – vom avea o listă ce conține ca elemente liste cu valorile numerice corespunzătoarea fiecărei coloane.

```
{
    "name": "test1",
    "height": 10,
    "width": 10,
    "rows": [[2], [4, 1], [1, 1], [2, 1, 2], [9], [7, 1], [9], [6, 2], [4, 2], [5]],
    "columns": [[1], [1, 4], [2, 6], [2, 7], [1, 6], [8], [1, 4, 1], [4, 2], [2, 3], [4]]
}
```

2.3 Date de ieșire

Datele de ieșire vor fi furnizate tot în cadrul unui fișier `json`, având următoarele câmpuri:

- `strategy` – strategia folosită pentru determinarea soluției;
- `nodes generated` – numărul de noduri generate în timpul căutării soluției;
- `nodes expanded` – numărul de noduri expandate complet în timpul căutării soluției;
- `time` – timpul necesar determinării soluției;
- `solution` – soluția problemei reprezentată sub forma unei liste de liste (celule colorate vor avea valoarea 1, iar cele albe vor avea valoarea 0).

3 Cerințe

3.1 Problema de căutare în spațiul stărilor

3.1.1 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.

3.1.2 Cerința 2 – 3 puncte

Implementați următoarele strategii de căutare neinformată: **Breadth First Search** (4.2.1), **Depth First Search** (4.2.2) și **Iterative deepening search** (4.2.4). Aplicați aceste strategii pentru rezolvarea problemei date.

3.1.3 Cerința 3 – 2 puncte

Implementați strategia de căutare informată **A* Search** (4.3.1) împreună cu două funcții euristice adecvate. Aplicați această strategie pentru rezolvarea problemei date.

3.2 Problema satisfacerii restricțiilor

Probleme de satisfacere a restricțiilor sunt probleme cu o mare aplicabilitate practică, deoarece multe situații din viața reală pot fi modelate folosind acest tip.

3.2.1 Cerința 4 – 3 puncte

Modelați puzzle-ul logic ca pe o problemă de satisfacere a restricțiilor și determinați o soluție pentru această problemă folosind algoritmul **Maintaining Arc Consistency (MAC)**. Problemele de acest tip se pot rezolva folosind tehnica Backtracking, acesta fiind un algoritm complet care găsește soluțiile problemei, dacă acestea există. Deoarece, în general, spațiul de căutare este unul foarte mare, ne vom propune să optimizăm algoritmul Backtracking prin reducerea acestui spațiu de căutare chiar în timpul rulării algoritmului. În cazul algoritmului **MAC**, acest lucru se realizează prin impunerea și menținerea arc-consistenței la fiecare pas al algoritmului (atât timp cât există variabile neinstanțiate).

Înainte de începerea căutării se aplică un algoritm pentru obținerea arc-consistenței verificându-se toate hiperarcele posibile. Apoi, la fiecare nod al arborelui de căutare, se impune arc-consistența pentru acele restricții corespunzătoare variabilei instanțiate în acel nod. Mai precis, dacă variabila x este cea insanțiată la pasul curent, se va impune arc-consistența pentru toate hiperarcele (y, c), unde $c \in \mathbf{C}$ este o constrângere cu $x \in \mathbf{Vars}(c)$, iar $y \in \mathbf{Vars}(c) \setminus \{x\}$.

Pentru reducerea spațiului de căutare (a domeniilor variabilelor) la rularea algoritmului Backtracking se pot impune restricții locale de consistență mai puternice decât arc-consistența, dar, în general, **MAC** reprezintă un compromis bun între costul propagării restricțiilor și dimensiunea spațiului efectiv explorat.

3.3 Analiză comparativă – 1 punct

Realizați o analiză comparativă între strategiile implementate, specificând **succint** pentru fiecare algoritm în parte care sunt avantajele identificate și limitările sale.

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;
- descrierea celor două funcții euristice considerate pentru algoritmul **A* Search**;
- tabel cu rezultatele obținute pentru fiecare strategie în parte pentru testele propuse și analizarea acestora. În acest tabel trebuie incluse informațiile legate de numărul de noduri generate, numărul de noduri expandate complet și timpul necesar determinării soluției.

3.4 Bonus – 2 puncte

Se acordă maximum 2 puncte bonus pentru:

- implementarea strategiei de căutare **Learning Real-Time A*** [1], aplicarea algoritmului pentru problema propusă și compararea acesteia cu strategia **A* Search** (**1 punct**);
- utilizarea euristicilor în rezolvarea problemei satisfacerii restricțiilor: ordonarea variabilelor în vederea atribuirii sau ordonarea valorilor în vederea atribuirii. (**1 punct**).

4 Noțiuni introductive

4.1 Probleme de căutare în spațiul stărilor

Problemele de căutare în spațiul stărilor 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
```

4.2 Strategii de căutare neinformată

4.2.1 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
```

4.2.2 Depth-first search (DFS)

Algorithm 3 Algoritmul Depth-first search (DFS)

```
procedure DEPTHFIRSTSEARCH(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
            insertFront(next, open)
    return fail
```

4.2.3 Depth-limited search

Algorithm 4 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

4.2.4 Iterative deepening search

Algorithm 5 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 $result$

4.3 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:

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

4.3.1 A* search

Algorithm 6 Algoritmul A* search

1: **procedure** A*SEARCH($s0, succ, goal$, h)
2: $open \leftarrow \{s_0\}$
3: $\pi(s_0) \leftarrow Nil$
4: $closed \leftarrow \emptyset$
5: **while** $open \neq \emptyset$ **do**
6: $current \leftarrow extractHead(open)$
7: **if** $goal(current)$ **then return** $(current, \pi)$
8: $closed \leftarrow closed \cup \{current\}$
9: **for** $next \in succ(current)$ **do**
10: $cost \leftarrow h(\text{next}) + g(\text{next})$
11: **if** $\exists next' \in closed \cup open$ ***such that*** $state(next) = state(next')$ **then**
12: **if** $g(next') < g(next)$ **then**
13: **continue**
14: **else**
15: $remove(next', open \cup closed)$
16: $\pi(next) \leftarrow current$
17: $insertSortedBy(cost, next, open)$
18: **return fail**

4.4 Probleme de satisfacere a restricțiilor [3]

O problemă de satisfacere a restricțiilor este descrisă printr-un set de variabile $\mathbf{X}$, o mulțime de domenii finite de valori pentru acestea $\mathbf{D}$ și un set de constrângeri $\mathbf{C}$. Vom nota $\mathbf{D}(\mathbf{x})$ domeniul variabilei $x \in \mathbf{X}$. O constrângere c va fi reprezentată printr-o relație între una sau mai multe variabile din $\mathbf{X}$.

O soluție pentru o astfel de problemă este reprezentată printr-o instanțiere a variabilelor din $\mathbf{X}$ cu valori ce satisfac toate restricțiile din $\mathbf{C}$.

4.4.1 Algoritmul GAC3

Arc-consistența reprezintă o metodă pentru propagarea restricțiilor (eliminarea din domeniile variabilelor a acelor valori care nu pot face parte dintr-o soluție a problemei). Arc-consistența este obținută atunci când pentru fiecare valoare din domeniul unei variabile și pentru orice restricție care implică acea variabilă există o instanțiere a tuturor variabilelor implicate care conține acea valoare astfel încât restricția să fie satisfăcută.

AC3 este un algoritm pentru impunerea arc-consistenței asupra domeniilor de valori ale variabilelor unei probleme descrise prin restricții. AC3 a fost ulterior generalizat pentru hiperarce, descrise pentru relații între mai mult de 2 variabile, această variantă fiind numită GAC3.

Algorithm 7 Algoritmul GAC3

1: **procedure GAC3**$(\mathbf{X}, \mathbf{D}, \mathbf{H}, \mathbf{C})$
2: **while** $\mathbf{H} \neq \emptyset$ **do**
3: $(x, c) \leftarrow first(\mathbf{H})$
4: $\mathbf{H} \leftarrow \mathbf{H} \setminus (x, c)$
5: $D_x^* \leftarrow$ **Revise**$(x, \mathbf{D}, c)$
6: **if** $\mathbf{D}(x) \neq$ **then**
7: $\mathbf{D}(x) \leftarrow \mathbf{D}_x^*$
8: $\mathbf{H} \leftarrow \mathbf{H} \cup \{(y, c') | c' \in \mathbf{C} \wedge c' \neq c \wedge x \in \mathbf{Vars}(c')\}$

Algoritmul GAC primește variabilele problemei $\mathbf{X}$, domeniile acestora $\mathbf{D}$, un set de hiperarce ce trebuie verificate și mulțimea tuturor constrângerilor problemei $\mathbf{C}$. GAC3 consideră la fiecare pas un hiperarc care corespunde unei restricții c și unei variabile x. Ceea ce se urmărește la un ciclu este eliminarea tuturor valorilor din domeniul lui x pentru care restricția c nu poate fi satisfăcută. Verificarea se face cu ajutorul funcției **Revise** care pentru fiecare valoare v din domeniul variabilei x caută un set de valori de suport τ pentru care este satisfăcută restricția. Dacă un astfel de set suport nu este găsit, valoarea v este eliminată din domeniul lui x.

Algorithm 8 Algoritmul Revise

1: **procedure Revise**$(x, \mathbf{D}, c)$
2: $\mathbf{D_x} \leftarrow \mathbf{D}(x)$
3: **for** $v \in \mathbf{D_x}$ **do**
4: **if** $\nexists \tau, \tau \in \times_{x_i \in \mathbf{Vars}(c)} \mathbf{D}(x_i), \tau$ satisface $c \wedge \tau(x) = v$ **then**
5: $\mathbf{D_x} \leftarrow \mathbf{D_x} \setminus \{v\}$
6: **return** $\mathbf{D_x}$

Mulțimea suport τ conține o instanțiere a tuturor variabilelor implicate în restricția c în care x are valoarea v. De aceea, pentru hiperarce cu un număr mare de variabile implicate, căutarea acestei mulțimi poate reprezenta o operație foarte costisitoare.

Dacă, după aplicarea funcției **Revise** pentru un hiperarc domeniul variabilei x este redus, atunci se verifică domeniile tuturor variabilelor care apar împreună cu x într-o constrângere (alta decât c). Drept urmare, pentru orice constrângere c' care implică variabila x se adaugă câte un hiperarc pentru fiecare altă variabilă $y \in \mathbf{Vars}(c'), y \neq x$.

Algoritmul se oprește atunci când nu mai există hiperarce de verificat sau când cel puțin unul dintre domeniile de valori este vid (în acest caz nu există soluție).

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] Tudor Berariu. Paradigme de Programare – Tema Prolog. `https://elf.cs.pub.ro/pp/15/teme/prolog-csp`, 2015. [Online; accesat pe 20-Oct-2021].