Compte rendu Projet-TP Intelligence Artificielle

BUI Van Tuan

3A STI

1. Exploration en profondeur-d'abord

Démarche : Implémentation de l'algorithme se base sur le pseudo-code **BREATH-FIRST-SEARCH** dans le poly Partie 1 du cours

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Pour l'exploration en profondeur-d'abord, son pseudo-code est comme l'exploration en largeur-d'abord, sauf qu'il utilise LIFO queue

Problèmes rencontrés :

- Implémentation de la fonction **SOLUTION**(node) quand l'état **node** est l'état but.
  Résoudre : Ajoute de la variable **solution** qui est un dictionnaire. Cette variable stocke des actions et ses noeuds correspondants qui ont exploré. Et après l'état but est atteint, on peut retrouver les actions pour y arriver dans la variable **solution**

- La variable **solution** n'est pas mise à jour quand un successeur de nœud d'exploration est déjà dans la frontière.
  Plus précisément, quand je teste la correction de l'algorithme : **python autograder.py**
  Mon algorithme ne marche pas pour le graphe

```
diagram: """
/-- B
|    ^
|    |
|   *A -->[G]
|    |     ^
|    V     |
\-->D ----/
```

Ma solution est A → G, mais la solution correcte est A → D → G ou A → B → D → G.

```
for child in problem.getSuccessors(node): #Les successeurs de noeud node
    #print("Successors: ", problem.getSuccessors(node))

    if child[0] not in explored and child not in frontier.list: #Si un successeur n'est pas dans la co
        solution[child[0]] = (child[1], node) #On stocke ce successeur, son parent et
                                    # l'action du parent vers ce successeur qui vient d'être exploré
```

Après A est exploré, D, G, B sont les frontières. Ensuite, le nœud D est exploré et son successeur G est l'état but et aussi une frontière, donc l'action D → G n'est pas stockée dans la variable **solution**.

Résoudre : Mise à jour la variable avant de tester le nœud dans la frontière.

```
if child[0] not in explored: #Si un successeur n'est pas dans la collection explored
    solution[child[0]] = (child[1], node) #On stocke ce successeur, son parent et
                                # l'action du parent vers ce successeur qui vient d'être exploré
    if problem.isGoalState(child): #Si ce noeud est l'état but
        goal = child #Affecter ce noeud à l'état but
        break
    if(child[0] not in frontier.list): # Si le successeur n'est pas dans la pile frontier
        frontier.push(child[0]) # On empile ce successeur à la pile
```

<u>Vérification de l'algorithme</u> : **python autograder.py**

```
Provisional grades
==================
Question q1: 3/3
```

2. Exploration par A*

<u>Démarche</u> : Implémentation de l'algorithme se base sur le pseudo-code UNIFORM-COST-SEARCH dans le livre « Artificial Intelligence_ A Modern Approach »

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Pour l'exploration par A^* , son pseudo-code est comme UNIFORM-COST-SEARCH, sauf qu'il utilise la fonction d'évaluation $f(n) = g(n) + h(n)$ au lieu de PATH_COST
Utilisation des mêmes variables du code de l'algorithme Exploration en profondeur-d'abord pour cette partie.

<u>Problèmes recontrés</u> :

- La fonction $h(n)$ est déjà implémentée et est passée en paramètre dans la fonction **aStarSearch** mais $g(n)$ n'est pas implémentée.
  Résoudre : Pour trouver $g(n)$, j'ai appliqué la formule $g(n) = g(p_n) + stepCost$, avec p_n est le nœud parent du nœud n, $stepCost$ est le coût pour arriver au nœud n du nœud p_n

```
for child in problem.getSuccessors(node): #les successeurs de noeud node
    if child[0] not in explored: #Si un successeur n'est pas dans la collecti
        path_cost[child[0]] = path_cost[node] + child[2] #Ajout le coût de ch
                                                         #La valeur est égale
                                                         #plus stepCost pour a
```

- La variable **solution** n'est pas mise à jour quand la valeur $f(n)$ du successeur n est plus petite que la valeur $f(n)$ précédente du nœud n.
  Résoudre :

```
if child[0] not in solution.keys(): #Si le successeur n'est pas stocké dans la dictionnaire solution
    solution[child[0]] = (child[1], node, f_n) #On stocke ce successeur, son parent et
                                               #l'action du parent vers ce successeur qui vient d'être
                                               #la fonction d'évaluation de ce successeur
else: #Si le successeur est déjà stocké dans la dictionnaire solution
    if f_n <= solution[child[0]][2]: #Si f_n est plus petite que la valeur f_n précédente
        solution[child[0]] = (child[1], node, f_n) #Mise à jour avec cette nouvelle valeur et
                                                   #l'action, son parent qui vient d'être explorée
```

Vérification de l'algorithme : **python autograder.py**

```
Provisional grades
==================
Question q1: 3/3
Question q2: 0/3
Question q3: 0/3
Question q4: 3/3
```

3. Manger toute la nourriture de Pacman en un minimum de temps

Pour minimiser le temps pour manger toute la nourriture, c'est équivalent à minimiser le nombre des actions pour manger toute la nourriture. Donc le coût de la solution est le nombre des actions.

Démarche : Génération d'heuristiques admissibles à partir de « **relaxed problems** ». « Relaxed problem » est un problème avec moins de restrictions sur les actions de l'agent. Le coût d'une solution optimale à un « relaxed problem » est une heuristique admissible pour le problème d'origine. Donc on va définir le « relaxed problems » en supprimant quelques conditions sur le problème d'origine :

- On supprime les conditions du mouvement de l'agent(Pacman). Il ne se déplace plus que les cases adjacentes, il peut se déplacer vers n'importe où dans son environnement. Donc le coût de la solution optimale de ce problème est le nombre de nourritures car Pacman a besoin seulement d'une action pour manger une nourriture. En vrai, $h_1(n) =$ le nombre de nourriture est admissible car Pacman doit déplacer au moins une fois s'il y a encore de nourritures.

  Tester la performance : **python autograder.py**

  ```
  *** PASS: test_cases\q7\food_heuristic_7.test
  *** PASS: test_cases\q7\food_heuristic_8.test
  *** PASS: test_cases\q7\food_heuristic_9.test
  *** FAIL: test_cases\q7\food_heuristic_grade_tricky.test
  ***     expanded nodes: 12517
  ***     thresholds: [15000, 12000, 9000, 7000]

  ### Question q7: 2/4 ###
  ```

  12517 nœuds ont été explorés pour l'environnement **trickySearch**, donc cette heuristique n'est pas la meilleure.

- On supprime les murs, la distance de Pacman à une nourriture est ainsi la distance manhattan. Cette distance est aussi le nombre d'actions pour arriver à la nourriture. Il y a plusieurs de nourritures, on définit la heuristique $h_2(n)$ = le maximum des distances

manhattan de Pacman à des nourritures. En effet, cette heuristique est admissible car Pacman doit déplacer au moins $h_2(n)$ pas pour manger de la nourriture.

Tester la performance : python autograder.py

```
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** FAIL: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 9409
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 3/4 ###
```

9409 nœuds ont été explorés pour l'environnement **trickySearch**, donc cette heuristique n'est pas la meilleure.

Remarque : h_2 domine h_1 donc h_2 est plus performante que h_1

Cette méthode a donné deux heuristiques admissibles mais ces deux heuristiques ne sont pas efficaces, donc on va appliquer la nouvelle méthode : génération d'heuristiques admissibles à partir de **sous-problèmes**. Clairement, le coût de la solution optimale de sous-problèmes est une borne inférieure du coût du problème d'origine. Donc ce coût est une heuristique admissible du problème d'origine. Les sous-problèmes qu'on définit dans le problème d'origine sont de trouver le chemin pour manger une seule nourriture dans les nourritures restantes. Pour trouver le chemin, on utilise l'algorithme qu'on a implémenté : **exploration en profondeur-d'abord**. Heureusement, le fichier searchAgents.py a une fonction **mazeDistance**() qui y sert.

Donc la heuristique $h_3(n)$ = le nombre des actions de chemin trouvé dans le sous-problème de l'état n.

Tester la performance : **python autograder.py**

```
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** PASS: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 5549
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 5/4 ###
```

Quelle surprise ! Seulement **5549** nœuds sont explorés, plus performants que les deux heuristiques précédente.

Remarque : Car il existe des murs dans l'environnement, donc la longueur de chemin trouvé dans les sous-probèmes est plus grand que la distance manhattan pour la plupart des cas. Donc c'est raisonnable qu'elle soit le plus performante.