**Звіт до комп’ютерного практикуму №2.**

**Класичні методи пошуку рішень у просторах станів**

**ПІБ: Головня О.Р.**

**Група: ІП-11**

**Мета:** Розглянути та дослідити алгоритми неінформативного та інформативного пошуку в моделях на основі станів. Провести порівняльний аналіз ефективності використання алгоритмів.

***З*авдання:** Реалізувати програму, яка розв’язує поставлену задачу 8-ферзів за допомогою алгоритму неінформативного пошуку DFS та алгоритму інформативного пошуку RBFS, що використовує задану евристичну функцію F1. Провести серію експериментів для вивчення ефективності роботи алгоритмів.

**Номер варіанту: 15**

**Завдання для варіанту:**

****

**Формалізація задачі:**

Стан: розташування 8 ферзів на шахівниці.

Дія: перестановка одного з ферзів в одній віссі

Початковий стан: Усі 8 ферзів розміщені на шахівниці так, що кожен ферзь не може побити іншого.

Функція визначення наступника:

* Вибирається один ферзь та переставляється у нову позицію в тому ж стовпці.
* Неіноформативний пошук: перша позиція за списком
* Інформативний пошук: екрестична функція

Еврестична функція: кількість пар ферзів, які б’ють один одного з урахуванням видимості

**Дерево пошуку:** На верхньому рівні знаходиться початковий стан, у якому всі 8 ферзів розташовані на шахівниці. На другому рівні знаходяться всі можливі стани, які з’являються при переміщенні одного з ферзів на шахівниці. Кожен наступний рівень – це нові стани, які можна отримати з попереднього рівня

У випадку DFS, наступник вибирається одним із можливих наступників, який ще не був відвіданий, і рекурсивно викликається функція DFS для цього наступного стану.

У випадку RBFS, наступник обирається на основі оцінки **f** (значення, що представляє інформацію про оцінку вартості шляху від поточного стану до кінцевого). RBFS обирає наступника зі списку наступників, який має найменше значення **f** серед всіх наступників, але не менше, ніж поточне обмеження **f\_limit**.

**Алгоритм неінформованого пошуку:**   
Алгоритм неінформованого пошуку в глибину (англ. Depth-first search**,** DFS) дозволяє побудувати обхід орієнтованого або неорієнтованого графа, при якому відвідуються всі вершини, які доступні з початкової вершини. Результатом алгоритму пошуку в глибину є деякий маршрут, рухаючись по якому можна обійти послідовно всі вершини графа, які доступні з початкової вершини. Робота алгоритму починається з кореня дерева і здійснюється обхід в максимально можливу глибину до переходу на наступну вершину.

Алгоритм пошуку в глибину (DFS) не є повним. Повний алгоритм пошуку повинен гарантувати знаходження розв'язку, якщо такий існує. Однак DFS може не знаходити розв'язок навіть у випадку, коли він існує, якщо алгоритм "застрягає" у гілці графа і не може продовжити пошук у глибину.

Властивості:  
Повний? – Ні, потрапляє до нескінченних просторів, просторів із циклами  
(з обеженням глибини теж ні, якщо depth<d  
Час? – O(b^D), при обмежені глибини - O(b^depth)

Простір? – O(bD) – Лінійний простір  
Оптимальний? – Ні!

**Програмна реалізація алгоритму неінформованого пошуку:**

**search(board)**: Ця процедура є вхідною точкою для алгоритму DFS. Вона створює кореневий вузол для початкового стану (шахова дошка) і додає його до стеку. Потім вона викликає **performSearch** для початку процесу пошуку.

*procedure search(board)*

*root = createNode(board)*

*stack.push(root)*

*performSearch(stack)*

*end procedure*

**performSearch(stack)**: Ця процедура виконує основний цикл пошуку в глибину. Вона продовжує виконання, доки стек не стане пустим. На кожній ітерації вона видаляє верхній елемент зі стеку і перевіряє, чи є він цільовим станом. Якщо ні, то генерує наступники поточного стану, перевіряє їх на повторення та додає до стеку для подальшого розгляду. Якщо ж поточний стан є цільовим, вона викликає **printSolutionPath** для виведення шляху до розв'язку і виводить інформацію про вартість шляху та кількість згенерованих і збережених вузлів.

*procedure performSearch(stack)*

*while stack is not empty do*

*tempNode = stack.pop()*

*if tempNode is not goal state then*

*successors = generateSuccessors(tempNode)*

*for each successor in successors do*

*newNode = createNode(successor.state, tempNode.cost + successor.cost)*

*generatedStatesCount++*

*if not checkRepeats(newNode) then*

*stack.push(newNode)*

*end if*

*end for*

*storedStatesCount = max(storedStatesCount, stack.size())*

*else*

*printSolutionPath(tempNode)*

*print "The cost was: " + tempNode.cost*

*print "Generated states count: " + generatedStatesCount*

*print "Stored states count: " + storedStatesCount*

*exit program*

*end if*

*end while*

*print "Error! No solution found!"*

*end procedure*

**checkRepeats(node)**: Ця функція перевіряє, чи був вузол з таким станом вже доданий до стеку. Вона переглядає всіх батьків вузла, щоб переконатися, що вузол ще не був оброблений.

**printSolutionPath(node)**: Ця процедура виводить шлях до розв'язку, починаючи з кінцевого вузла і переходячи до початкового. Вона використовує стек для збереження шляху та виводить кожен стан шляху на екран

**Алгоритм інформованого пошуку:**

Рекурсивний пошук по першому найкращому збігу*— RBFS* (англ. *Recursive Best-First Search*)  є вдосконаленням алгоритму пошуку з першим кращим вибором (Best-First Search), який дозволяє враховувати обмеження на глибину пошуку.

Основна ідея RBFS полягає в тому, що він використовує рекурсію для обходу графа. Він здійснює пошук по графу, розширюючи кращий вузол, що не перевищує поточне обмеження на f-значення. Якщо найкращий вузол має f-значення, яке перевищує обмеження, алгоритм вибирає наступний кращий вузол у списку. Якщо ж всі вузли мають f-значення, які перевищують обмеження, алгоритм розглядає вузли знову, збільшуючи обмеження на f-значення.

Алгоритм не є повним, оскільки він не гарантує знаходження розв'язку, якщо такий існує. Він може застрягти в локальних мінімумах або вийти за межі обмежень, якщо відсутній шлях до розв'язку в межах заданих обмежень.

Властивості:  
Повний? – Так  
Час? – Залежить від евристики та того, наскільки часто відбувались зміни найкращого шляху

Простір? – O(bd)   
Оптимальний? – Так, якщо h(n) допустима

**Евристична функція оцінки:**

F1 – кількість пар ферзів, які б’ють один одного з урахуванням видимості

(ферзь А може стояти на одній лінії з ферзем В, проте між ними стоїть

ферзь С; тому А не б’є В)  
(Буде вираховуватись у функції **calculateAttacks()**)

**Програмна реалізація алгоритму інформованого пошуку:**

Основні процедури алгоритму Recursive Best-First Search (RBFS) можна навести у вигляді псевдокоду наступним чином:

*Procedure RBFS(node, fLimit)*

*If node is goal state*

*Print solution path and exit*

*Generate successors of node*

*If no successors*

*Return infinity*

*For each successor of node*

*If successor not in path to node*

*Add successor to list of successors*

*While successors exist*

*Sort successors by f-cost*

*If f-cost of best successor > fLimit*

*Return f-cost of best successor*

*Alternative = fLimit if there is only one successor, otherwise f-cost of second best successor*

*newFCost = RBFS(best successor, min(fLimit, Alternative))*

*If newFCost < infinity*

*Update f-cost of best successor to newFCost*

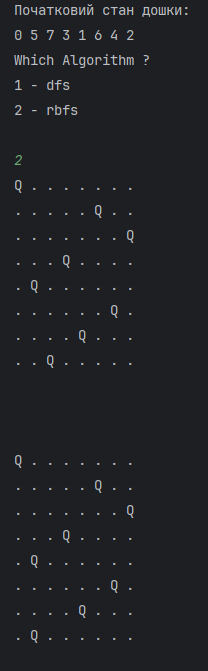
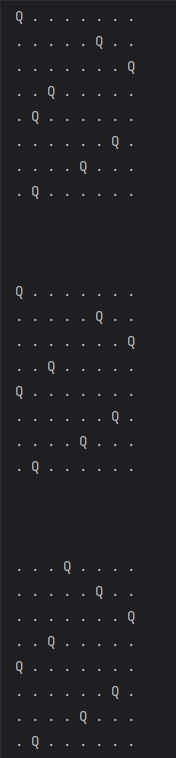
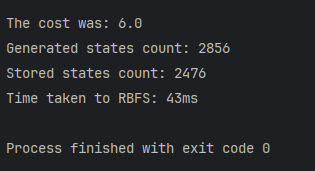
*Remove best successor from path*

*Return infinity*

1. Перевірка, чи поточний вузол є цільовим. Якщо так, виведення шляху до цілі та завершення алгоритму.
2. Генерація нащадків поточного вузла і перевірка, чи є вони унікальними.
3. Пошук нащадків, які не зустрічалися на шляху, та додавання їх до списку нащадків.
4. Сортування нащадків за значенням f-функції.
5. Якщо f-значення найкращого нащадка перевищує ліміт **fLimit**, повернення значення f-функції цього нащадка.
6. Обчислення альтернативного ліміту **Alternative** для випадку, коли є два або більше нащадків.
7. Рекурсивний виклик RBFS для найкращого нащадка з новим лімітом **min(fLimit, Alternative)**.
8. Оновлення f-значення найкращого нащадка на нове значення, отримане під час рекурсивного виклику.
9. Видалення найкращого нащадка зі шляху.

**Приклад роботи розробленої програми:**

На рисункаъ нище наведений скріни з результатами для прикладу використання RBFS:

**Результати використання алгоритмів:**

Результати для RBFSearch:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Експери-мент | Початковий стан | Час, мілі-секунди | Загальна кількість згенерованих вузлів | Максимальна кількість вузлів, що зберігалися в пам'яті |
| 1 | 7 5 5 7 3 7 5 4 | 21ms | 224 | 189 |
| 2 | 4 4 5 6 2 7 2 4 | 36ms | 4872 | 4083 |
| 3 | 0 3 5 4 6 3 7 5 | 31ms | 1456 | 1249 |
| 4 | 6 1 7 1 4 4 6 4 | 39ms | 280 | 247 |
| 5 | 3 4 2 2 0 6 0 2 | 31ms | 4648 | 3954 |
| 6 | 3 0 3 0 6 7 7 3 | 46ms | 6048 | 5077 |
| 7 | 5 6 3 0 1 7 2 2 | 16ms | 280 | 236 |
| 8 | 1 1 3 1 3 5 4 1 | 38ms | 784 | 683 |
| 9 | 1 2 5 4 6 1 2 6 | 92ms | 4872 | 4090 |
| 10 | 7 3 1 6 1 1 6 7 | 36ms | 5432 | 4703 |

Результати для DFSearch: при обмеженні в глибину на 6 (Також було обмеження при використанні пам’яті 512Мб і 10хв по часу виконання)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Експери-мент | Початковий стан | Час, мілі-секунди | Загальна кількість згенерованих вузлів | Максимальна кількість вузлів, що зберігалися в пам'яті |
| 1 | 1 6 4 5 3 7 0 2 | 2381ms | 146496 | 92311 |
| 2 | 4 0 2 5 6 3 7 1 | Time\_limit | - | - |
| 3 | 0 1 4 0 3 3 5 7 | Time\_limit | - | - |
| 4 | 2 5 7 0 0 1 1 0 | 1181ms | 83496 | 54433 |
| 5 | 4 1 5 3 6 0 2 0 | 84083ms | 991256 | 600500 |
| 6 | 4 1 6 7 5 0 2 3 | 99075ms | 1125584 | 609618 |
| 7 | 3 4 1 7 3 3 4 7 | Time\_limit | - | - |
| 8 | 6 1 2 3 5 0 7 4 | Time\_limit | - | - |
| 9 | 5 2 6 4 3 0 7 1 | 61464ms | 699944 | 412629 |
| 10 | 2 4 3 1 0 6 5 7 | Time\_limit | - | - |

**Оцінка результатів навчання:**

Під час виконання лабораторної роботи, було використано алгоритми неінформативного пошуку DFS та інформативного пошуку RBFS з використанням еврестичної функції F1.

Використання DFS виявилось не найкращим алгоритмом для вирішення поставленої задачі, бо він є дуже неефективним для великих задач, так як складність алгоритму зростає експоненційно.

На відміну від DFS, алгоритм RBFS з використанням еврестичної функції F1 показав значно кращі результати. Складність алгоритму залежить від еврестичної функції.

Отже, практичні результати підтверджують теоретичні твердження про алгоритми.