МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ І.І. МЕЧНИКОВА
Факультет математики, фізики та інформаційних технологій

Звіт
з дисципліни: «Комп’ютерні системи штучного інтелекту»
Лабораторна робота № 1
за темою: «Евристичний пошук»

Виконав: студент денної форми навчання
Лавров Владислав
спеціальності: Комп’ютерна інженерія
кафедри: КСТ
курс: 4
Викладач:
Гожий О.П.

Одеса – 2024

Лабораторна робота №1. Евристичний пошук

Завдання: Створити дерево пошуку для гри “8”. Реалізувати в програмному коді алгоритм вирішення задачі з використанням однієї з стратегій пошуку. Навести кількість кроків, необхідних для вирішення задачі та вивести їх послідовність. Передбачити можливість відсутності рішення та можливість зміни цільового стану.

Варіант № 7

1	8	7
6	0	5
2	4	3

Теоретичні відомості:

Пошук та організація даних у сфері штучного інтелекту є ключовою темою. Часто виникають завдання, де необхідно знайти правильне рішення серед великої кількості варіантів. У деяких випадках ця кількість може бути настільки великою, що стає неможливим розробити алгоритм, який знайде єдине вірне рішення. Тут на допомогу приходять евристичні методи, які дозволяють прискорити процес пошуку та наблизитися до оптимального рішення.

Існують різні види пошуку, такі як DFS, BFS і UCS, які часто використовуються при роботі з графами. Ці методи відносяться до неінформованого пошуку, де всі можливі шляхи перевіряються, і оптимальний вибирається без заздалегідь відомої інформації.

З іншого боку, інформований пошук, що використовує апріорну інформацію, є евристичним. Він дозволяє виключити зайві шляхи та прискорює процес пошуку рішення. Евристичні методи є ефективними для швидкого отримання розумного рішення в обставинах, коли інші методи займали б занадто багато часу.

Хід роботи:

Програма використовує алгоритм A* для пошуку шляху вирішення головоломки, де пріоритетність визначається на основі евристичної функції (Манхеттенська відстань). Алгоритм пошуку вирішення завдання складається з наступних етапів:

1. Створення колекції унікальних станів гри: використовується map visited, для відстеження вже відвіданих станів гри.

2. Створення колекції пріоритетної черги: використовується структура PriorityQueue, яка реалізує пріоритетну чергу для вибору наступного кроку.

3. Додавання початкового стану в чергу: початковий стан гри додається до пріоритетної черги з пріоритетом 0.

4. Цикл обробки станів:

- Цикл працює доти, поки пріоритетна черга не стане порожньою або час не вийде за межі обмеження.
- На кожній ітерації з черги вилучається (і видаляється) елемент з найвищим пріоритетом.
- Перевіряється, чи є поточний стан кінцевим. Якщо так, повертається шлях до вирішення задачі.
- Перевіряється, чи вже відвіданий поточний стан. Якщо ні, додається до відвіданих станів.
- Для поточного стану обчислюються можливі допустимі ходи, та для кожного з них обчислюється значення евристичної функції.
- Нові стани додаються до поля маршруту поточного елемента разом з поточним станом, та вставляються у пріоритетну чергу.

5. Виведення результатів: після закінчення обробки пріоритетної черги виводиться шлях до вирішення задачі та час роботи алгоритму.

6. Введення початкової головоломки:

- Користувачеві пропонується ввести початкову головоломку розміром 3x3 з клавіатури.
- Кожен рядок вводиться окремо, числа в рядку розділені пробілами.

7. Ініціалізація алгоритму та його запуск:

- Після введення головоломки, створюється об'єкт алгоритму AStarAlgorithm.
- Запускається метод Run(), який виконує алгоритм A* та виводить результат.

У консолі користувач вводить початковий стан, а також можемо побачити маршрут вирішення задачі та час роботи алгоритму. Початок та кінець роботи програми наведено на рисунку 1 та рисунку 2.

Нижче наведено лістинг програми на мові програмування Go:

Лістинг коду:

```
package main

import (
   "bufio"
   "container/heap"
   "errors"
   "fmt"
   "os"
   "strconv"
   "strings"
   "time"
)

// AStarAlgorithm представляє алгоритм A* для вирішення головоломки.
type AStarAlgorithm struct {
   initArr   [][]int // Початковий стан головоломки
   targetArr [][]int // Цільовий стан головоломки
}

// State представляє стан головоломки для алгоритму A*.
type State struct {
   priorityNum int    // Пріоритет (використовується для пріоритетної черги)
   arr         [][]int   // Поточний стан головоломки
   path        [][][]int // Шлях, який призвів до цього стану
}

// NewAStarAlgorithm ініціалізує новий екземпляр AStarAlgorithm.
func NewAStarAlgorithm(initArr [][]int) (*AStarAlgorithm, error) {
   // Перевірка на валідність початкової головоломки
   if len(initArr) != 3 || len(initArr[0]) != 3 {
```

```
      return nil, errors.New("Початковий масив повинен бути матрицею розміром
3x3.")
   }

   // Цільовий стан головоломки
   targetArr := [][]int{
      {1, 2, 3},
      {8, 0, 4},
      {7, 6, 5},
   }

   return &AStarAlgorithm{
      initArr:   initArr,
      targetArr: targetArr,
   }, nil
}

// getZeroPos знаходить позицію нульового елемента в головоломці.
func (a *AStarAlgorithm) getZeroPos(currentArr [][]int) []int {
   for i, row := range currentArr {
      for j, val := range row {
         if val == 0 {
            return []int{i, j}
         }
      }
   }
   panic("Неправильна головоломка: порожній простір (нуль) не знайдено.")
}

// move міняє місцями елементи в головоломці.
func (a *AStarAlgorithm) move(currentArr [][]int, zeroPos, newZeroPos []int)
[][]int {
   updatedArr := a.copyArr(currentArr)
   updatedArr[zeroPos[0]][zeroPos[1]] =
currentArr[newZeroPos[0]][newZeroPos[1]]
   updatedArr[newZeroPos[0]][newZeroPos[1]] = 0
   return updatedArr
}

// isPosValid перевіряє, чи нова позиція знаходиться в межах матриці.
func (a *AStarAlgorithm) isPosValid(position []int) bool {
   return position[0] >= 0 && position[0] < 3 && position[1] >= 0 &&
position[1] < 3
}

// getPossibleMoves повертає список можливих наступних матриць.
func (a *AStarAlgorithm) getPossibleMoves(currentArr [][]int) [][][]int {
   zeroPos := a.getZeroPos(currentArr)
   offsets := [][]int{{0, 1}, {1, 0}, {0, -1}, {-1, 0}} // Всі можливі зсуви
в горизонтальному та вертикальному напрямках
   var possibleMoves [][][]int
```

```
   for _, offset := range offsets {
      newZeroPos := []int{zeroPos[0] + offset[0], zeroPos[1] + offset[1]}
      if a.isPosValid(newZeroPos) {
         possibleMoves = append(possibleMoves, a.move(currentArr, zeroPos,
newZeroPos))
      }
   }

   return possibleMoves
}

// calcManhattanDistance обчислює Манхеттенську відстань до цільової матриці.
func (a *AStarAlgorithm) calcManhattanDistance(currentArr [][]int) int {
   distance := 0
   for i, row := range currentArr {
      for j, val := range row {
         if val != 0 {
            distance += abs(i-(val-1)/3) + abs(j-(val-1)%3)
         }
      }
   }
   return distance
}

// algorithm знаходить шлях вирішення або генерує помилку.
func (a *AStarAlgorithm) algorithm() ([][][]int, error) {
   visited :=
make(map[string]bool)
    // відстеження відвіданих станів
   priorityQueue := make(PriorityQueue,0) // Пріоритетна черга для вибору
наступного кроку
   heap.Init(&priorityQueue)  // Ініціалізація пріоритетної черги
   heap.Push(&priorityQueue, &State{priorityNum: 0, arr: a.initArr, path:
make([][][]int, 0)}) // Додавання початкового стану в чергу

   startTime := time.Now()
   for priorityQueue.Len() > 0 && time.Since(startTime).Milliseconds() <
10000 { // Виконуємо цикл, доки черга не пуста або не вичерпано час
      current := heap.Pop(&priorityQueue).(*State) // Беремо поточний стан з
пріоритетної черги

      // Якщо поточний масив дорівнює цільовому, повертаємо шлях вирішення
      if a.deepEquals(current.arr, a.targetArr) {
         res := append(current.path, current.arr)
         return res, nil
      }

      stateHash := a.getStateHash(current) // Отримуємо хеш-рядок поточного
стану
      if !visited[stateHash] { // Перевіряємо, чи ми ще не відвідали цей стан
```

```
            visited[stateHash] = true       // Позначаємо стан як відвіданий
            possibleMoves := a.getPossibleMoves(current.arr) // Отримуємо
можливі наступні стани
            for _, arr := range possibleMoves {
               heuristic := a.calcManhattanDistance(arr)
               currentPath := append(current.path, current.arr)   // Додаємо
поточний стан до шляху
               heap.Push(&priorityQueue, &State{priorityNum: heuristic, arr:
arr, path: currentPath}) // Додаємо наступний стан в чергу з його пріоритетом
            }
         }
      }
      return nil, errors.New("Час виконання алгоритму перевищено!")
}

// Run виконує алгоритм A* та виводить результат.
func (a *AStarAlgorithm) Run() {
      start := time.Now()
      arr, err := a.algorithm()

      // Вимір часу роботи
      fmt.Printf("Час вирішення: %d мс.\nШлях:\n",
time.Since(start).Milliseconds())

      // Виведення шляху вирішення
      if err != nil {
         fmt.Println("Щось пішло не так!")
         fmt.Println(err.Error())
         return
      }

      numOfStep := 0
      for _, subArr := range arr {
         fmt.Printf("Крок #%d\n", numOfStep)
         for _, i := range subArr {
            fmt.Println(i)
         }
         fmt.Println()
         numOfStep++
      }
}

// copyArr повертає копію заданого масиву.
func (a *AStarAlgorithm) copyArr(arr [][]int) [][]int {
      copyArr := make([][]int, len(arr))
      for i, row := range arr {
         copyArr[i] = make([]int, len(row))
         copy(copyArr[i], row)
      }
      return copyArr
}
```

```go
// getStateHash повертає хеш-рядок для State.
func (a *AStarAlgorithm) getStateHash(s *State) string {
   return fmt.Sprintf("%v", s.arr)
}

// deepEquals перевіряє, чи дві матриці глибоко рівні.
func (a *AStarAlgorithm) deepEquals(arr1, arr2 [][]int) bool {
   return fmt.Sprintf("%v", arr1) == fmt.Sprintf("%v", arr2)
}

// abs повертає абсолютне значення цілого числа.
func abs(x int) int {
   if x < 0 {
      return -x
   }
   return x
}

// main - головна функція для запуску алгоритму A*.
func main() {
   fmt.Println("Введіть початкову головоломку розміром 3x3 (цифри від 0 до 8,
розділені пробілом, рядок за рядком):")
   initArr := make([][]int, 3)
   reader := bufio.NewReader(os.Stdin)
   for i := 0; i < 3; i++ {
      fmt.Printf("Рядок #%d: ", i+1)
      input, _ := reader.ReadString('\n')
      input = strings.TrimSpace(input)
      nums := strings.Split(input, " ")
      for _, numStr := range nums {
         num, _ := strconv.Atoi(numStr)
         initArr[i] = append(initArr[i], num)
      }
   }

   astar, err := NewAStarAlgorithm(initArr)
   if err != nil {
      fmt.Println("Щось пішло не так!")
      fmt.Println(err.Error())
      return
   }

   astar.Run()
}

// PriorityQueue реалізує пріоритетну чергу для структури State.
type PriorityQueue []*State

func (pq PriorityQueue) Len() int           { return len(pq) }
```

```go
func (pq PriorityQueue) Less(i, j int) bool { return pq[i].priorityNum <
pq[j].priorityNum }
func (pq PriorityQueue) Swap(i, j int)      { pq[i], pq[j] = pq[j], pq[i] }

// Push та Pop - функції для heap.Interface
func (pq *PriorityQueue) Push(x interface{}) {
   item := x.(*State)
   *pq = append(*pq, item)
}

func (pq *PriorityQueue) Pop() interface{} {
   old := *pq
   n := len(old)
   item := old[n-1]
   *pq = old[0 : n-1]
   return item
}
```

```
Введіть початкову головоломку розміром 3x3 (цифри від 0 до 8, розділені пробілом, рядок за ряд
ком):
Рядок #1: 1 8 7
Рядок #2: 6 0 5
Рядок #3: 2 4 3
Час вирішення: 14 мс.
Шлях:
Крок #0
[1 8 7]
[6 0 5]
[2 4 3]

Крок #1
[1 8 7]
[6 4 5]
[2 0 3]

Крок #2
[1 8 7]
[6 4 5]
[0 2 3]

Крок #3
[1 8 7]
[0 4 5]
[6 2 3]
```

Рисунок 1 – Початок виводу результату роботи програми

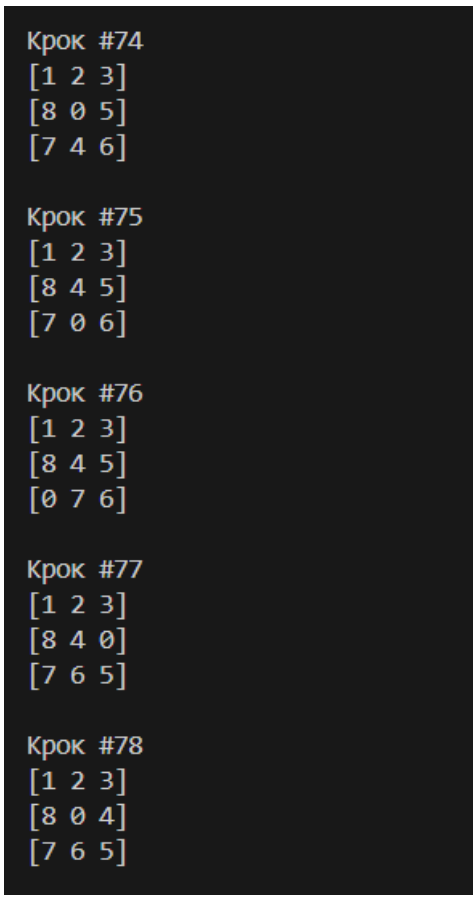

Рисунок 2 – Кінець виводу результату роботи програми

Висновок

Під час цієї лабораторної роботи я ознайомився з алгоритмом A* та використанням Манхеттенської відстані для вирішення головоломки "8". На основі цього я розробив програму, що може вирішувати головоломку "8" для різних початкових станів.

Контрольні запитання:

1. Евристика – поняття та визначення.
2. Формування дерева пошуку.
3. Стратегії сліпого пошуку.
4. Евристичний пошук. Особливості.
5. Евристична оцінювальна функція.

Відповіді:

1). Евристика – це підхід до розв'язання проблем, що ґрунтується на досвіді та інтуїції. Зазвичай вона застосовується тоді, коли повний аналіз всіх можливих варіантів є недоцільним або неможливим. Евристичні методи дозволяють швидко знаходити приблизні рішення, опираючись на експертний досвід або здоровий глузд. У багатьох випадках вони використовуються в алгоритмах штучного інтелекту та оптимізації для покращення продуктивності та ефективності.

2). Формування дерева пошуку – процес побудови структури даних у вигляді дерева, де кожен вузол представляє можливий стан або конфігурацію, а ребра відображають можливі переходи між станами. Це використовується для організації пошукових алгоритмів, таких як алгоритм A*.

3). Стратегії сліпого пошуку – це методи пошуку, які не використовують інформацію про ціль. Вони базуються на систематичному переборі можливих варіантів без заздалегідь заданих евристик або оцінок. Такі методи включають пошук в ширину, пошук в глибину та їх варіації.

4). Евристичний пошук – це метод, який використовує спеціальні правила або припущення, щоб швидше знаходити рішення в складних задачах. Він дозволяє скоротити обсяг пошуку і прискорити процес прийняття рішень. Такий підхід часто використовується в оптимізаційних задачах та при створенні алгоритмів штучного інтелекту.

5). Евристична оцінювальна функція – це функція, яка використовується в евристичних алгоритмах для оцінки потенційності кожної альтернативи або стану системи. Вона допомагає визначити, які варіанти є більш перспективними для подальшого розгляду або вибору.