СОФИЙСКИ УНИВЕРСИТЕТ „СВ. КЛИМЕНТ ОХРИДСКИ“

Факултет по математика и информатика

Дисциплина: СОЗ (3ти курс ИС, зимен семестър 2020/2021)

ЗАДАНИЕ ЗА ДОМАШНА РАБОТА № 1

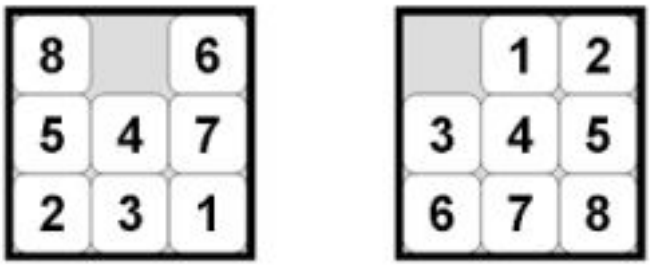
Кристина Георгиева Цекова, ФН: 71852

Евристично търсене в пространство на състояния

1. Описание на задачата.

Имплементирайте метода за търсене с минимизиране на общата цена на пътя (A*) и метода на най-доброто спускане (best-first search), които да намират решение на играта с плъзгащите плочки с размерност на дъската 3x3 (8 - puzzle). Целта на играта е да се преместят плочките от началото до крайното състояние, с възможно най-малко ходове. Генерирайте случайно начално разположение на плочките.

пример: начално състояние целево състояние



Възможните ходове са плъзгане нагоре, надолу, наляво и надясно на плочка, съседна на празния квадрат, водещо до размяна на мястото на плочката с това на празния квадрат.

Примерни евристични функции h(n):

✓ Манхатънско разстояние (Manhattan distance) – сумата на разстоянията на всички плочки от оценяваното състояние до правилните им положения в целевото състояние.

✓ Броят на плочките в оценяваното състояние, които не са на местата си в целевото състояние.

Необходимо е да се направят експерименти с поне две различни евристични функции и да се анализират получените резултати.

2. Описание на използваните методи за решението на задачата.

Играта с плъзгащите плочки се състои от 3 реда и 3 колони, като дъската с размерност 3x3 се състои от общо 9 плочки – на осем от тях можем да поставяме цифри от 0 до 8, а оставащата плочка е за празно пространство, на което можем да местим тези 8 плочки. Играта може да се реши чрез местене на плочките една по една в празното пространство, докато не стигнем до целевото състояние. Но тя не винаги има решение. В случай че броят на плочките е четен, то играта има решение, ако:

- празното място е на четен ред, броейки отдолу и инверсиите са нечетен брой;

- празното място е на нечетен ред, броейки отдолу, а броят на инверсиите е четен. Във всички останали случаи, задачата няма решение.

За реализацията на решението на задачата съм направила общо 4 класа заедно с 4 тестови класа. Първите два от тях са class Table и class State, които представят самата дъска и нейните състояния.

Class Table се състои от 9 плочки. Самите плочки организирам в едномерен масив с размер 9. Добавям и индекс, който показва на коя позиция се намира празното място. Създавам си конструктори, функции за достъп до елементите на класа, както и функции за промяна на стойностите на член-данните. Добавям и метод toString(), който извежда на екрана информация за данните. Освен тези методи създавам 4 метода за местене на плочките нагоре, надолу, наляво или надясно, въвеждайки проверки за това кога не могат да се местят на съответната позиция. Имам и два допълнителни метода. Методът isGoal() е от тип boolean и връща истина, когато целта е достигната. А другият - isSolvable() показва дали задачата има решение. И това става като се броя инверсиите между елементите. Ако инверсиите са четен брой, то има решение. В противен случай - няма,

Class State представя състоянието на възлите. Той се състои от 5 член-данни: дъска, родител на текущото състояние, оператор, цена на пътя и евристична функция. За да може да се сравняват отделните състояния, той имплементира interface Comparable, заедно с метода му compareTo(state). В него също добавям метод за извеждане на информация за съответните данни, както и конструктор за общо ползване, сет и гет функции, както и метод isGoal(), който връща метода isGoal() на съответната дъска.

За тези два класа съм създала тестови класове – TestTable и TestState , чрез които просто тествам функционалността им и не показват решението на задачата.

Другите два класа са – class HeuristicFunctions и class MethodsForSearching.

Първият от тях се състои от две функции, които представят реализацията на двете евристични функции – Манхатънското разстояние и броят на плочките, които не са наредени в целевото състояние. И двата метода връщат резултат от тип цяло число.

Вторият клас – class MethodsForSearching има като член-данна състояние (state) и спрямо него се изпълняват двата метода за търсене, за които съм създала две отделни функции.

За тези два класа също имам тестови класове – TestHeuristics и TestMethods, като първият от тях тества работата на класа HeuristicFunctions, а вторият – реализира решението на цялата задача, използвайки двата метода.

2.1. Метод за търсене с минимизиране на общата цена на пътя (A*)

Ключова характеристика на метода е, че той пази запис на всеки посетен възел, който помага в игнорирането на възлите, които вече са посетени, спестявайки огромно количество време.

В моята реализация използвам опашка (Priority queue), която съхранява сортирани посетените възли (фронт). И докато фронтът не е празен, т.е. имаме възли в опашката, местим плочките в зависимост от текущото състояние на празното място, прилагайки евристична функция за всяко състояние. Така разширяваме текущото състояние. Най – подходящото състояние се избира и разширява отново. Този процес продължава, докато състоянието на целта настъпи като текущо състояние.

Бележка: Ако сме преместили плочката нагоре, то няма смисъл тя да се връща отново надолу. Както и ако местим надясно, то няма смисъл да я местим наляво. И обратно. В случай, че опашката остане празна, се извежда съобщение за грешка.

2.2 Метод на най – доброто спускане. (Best-first search)

Този метод доста прилича на метода A*. При него също определяме възел за търсене. Първо, вмъкваме първоначално избрания възел в приоритетната опашка. След това изтриваме от опашката възела с минимален приоритет и вмъкваме всички негови наследници, които могат да бъдат достигнати с едно преместване от декларирания възел. Тази процедура се повтаря, докато възелът за търсене, отхвърлен в съответствие, отговаря на целевото състояние.

3. Описание на реализацията с псевдокод.

3.1. Псевдокод на реализацията на методите - Манхатънско разстояние и броят на ненаредените плочки в целевото състояние.

- Методите се различават единствено по изчисляването на резултата.

function manhattanDistance/numberOdUnorderedBlocks(table) returns a solution of failure

    declare and initialize an array temp[] which get the tiles from the table

    declare and initialize the result with 0

    for each index = 0 till the length of temp

        if current tile is not equal to 0 AND current tile is not equal to index

        then calculate the result

return result

3.2. Псевдокод на реализацията на метода A* и на метода на най - доброто спускане.

- Двата метода се различават единствено по цената на пътя, което е отразено в кода.

```
function bestFirstSearch/aStar(originalState, hfunc) returns a solution or failure
    declare frontier of type PriorityQueue
    add the originalState to the frontier
    loop while frontier is not empty
        take the first minimum state
        if it is the goal
            return it
        else declare new state and take it from the frontier
            declare four tables to be moved up, down, left or right
            generate its acyclic nodes and take an action
            switch action
                case up -> move all tables different from down
                case down -> move all tables different from up
                case left -> move all tables different from right
                case right -> move all tables different from left
                default -> the first state -> move it up, down, left, right
            end switch
            if we can move the table up
                add it to the frontier with action up and apply the heuristic function
            if we can move the table left
              add it to the frontier with action left and apply the heuristic function
            if we can move the table right
              add it to the frontier with action right and apply the heuristic function
            if we can move the table down
              add it to the frontier with action down and apply the heuristic function
    if the queue is empty -> returns an error
end function
```

4. Заключение.

Задачата за плъзгащите плочки не винаги има решение и това се доказва чрез броят на инверсиите между елементите. Двата алгоритъма, с които се търси решение, са пълни, но и експоненциални по памет. Но за разлика от A*, който **винаги** намира решение - и то най - оптималното решение - при методът за бързото спускане не винаги е така. Това става ясно и от множеството тестове, които направих, тъй като в повечето случаи елементите, с които работи този метод са прекалено много и се стига до изчерпване на паметта. Следователно, можем да направим извод, че A* е много по-бърз алгоритъм, който е за предпочитане пред Best-First search.