Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Робототехники и комплексной автоматизации

КАФЕДРА Системы автоматизированного проектирования (РК-6)

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

по дисциплине: «Интеллектуальные системы»

Студент	Макаров Тимофей Геннадьевич
Группа	РК6-22М
Тип задания	Лабораторная работа
Тема	Алгоритм A*

Студент _____ **Макаров Т.Г.**
подпись, дата *фамилия, и.о.*

Преподаватель _____ **Божко А.Н.**
подпись, дата *фамилия, и.о.*

Оценка _____

Москва, 2023 г.

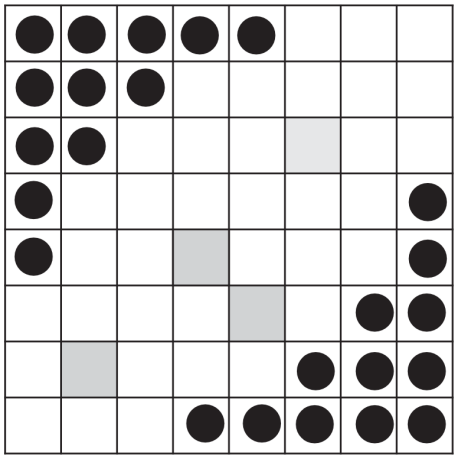
Оглавление

Алгоритм A*

Алгоритм A* (англ. A star) — алгоритм поиска, который находит во взвешенном графе маршрут наименьшей стоимости от начальной вершины до выбранной конечной. Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обозначаемой как f(x)). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обозначается как g(x)), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

Описание задачи

Вариант 17



Переместить все фишки в противоположный угол (нижний правый). Фишка может перепрыгивать через другую фишку на свободное поле по вертикали или горизонтали. Движение по диагонали запрещено. Фишки не снимаются. Серое поле запрещено для посещения.

Используемая эвристика

Для решения поставленной задачи в качестве эвристики было решено взять сумму манхэттенских расстояний фишек до правого нижнего угла:

$$h(x) = \sum_{i=1}^{N} \|x_i - t\|, \qquad (1)$$

где || || означает манхэттенскую норму, x_i – координаты i -й фишки на доске, t – координаты правого нижнего угла доски.

Манхэттенское расстояние – это сумма модулей разностей координат.

Более наглядно пример вычисления данной эвристики показан на рисунке 1.

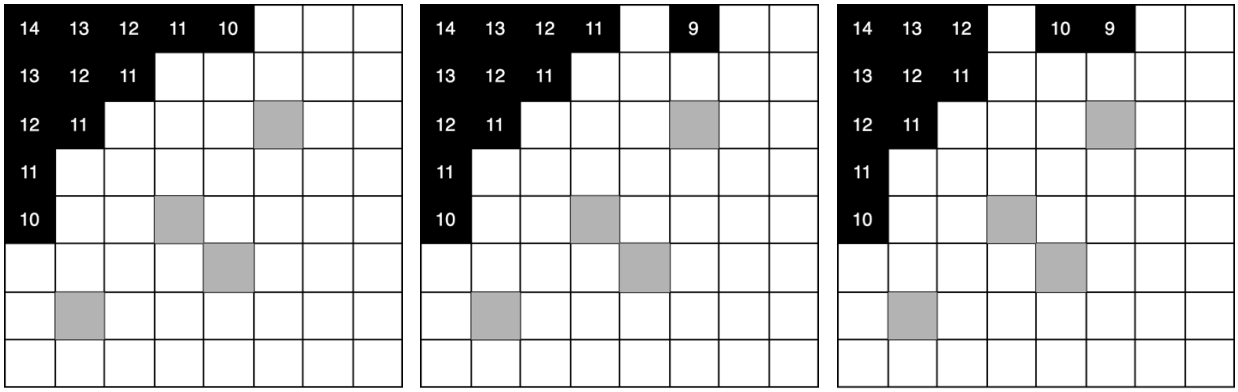


Рисунок 1 – Пример значений эвристики для различных конфигураций. Цифрами на доске обозначены манхэттенские расстояния от каждой фишки до правого нижнего угла

Особенности реализации

Для написания программной реализации решения поставленной задачи с помощью алгоритма A был использован язык программирования Go и интегрированная среда разработки Goland.

Для хранения списка open в алгоритме A была использована очередь с приоритетом, реализованная с помощью двоичной кучи. В Go данная структура является встроенным типом container.Heap.

Начальная и конечная конфигурации доски передаются в формате csv. Пример начально состояния в данном формате приведён в листинге 1.

Для хранения конфигураций доски была разработана структура Board, состоящая из размера доски size и информации о конфигурации доски board. Для данной структуры определены следующие методы:

- NewBoardFromFile – получение экземпляра доски из конфигурационного csv-файла;

- GetNeighbours – получение ходов, доступных в текущей конфигурации;
- getHorizontalMoves, getVerticalMoves – получение ходов по горизонтали и вертикали соответственно;
- isOnBoard – проверка того, лежит ли некоторая координата в пределах доски;
- getMove – изменение поля board для совершения хода.
- Heuristic – эвристика описанная формулой (1);
- Print – вывод в терминал псевдографического изображения доски.

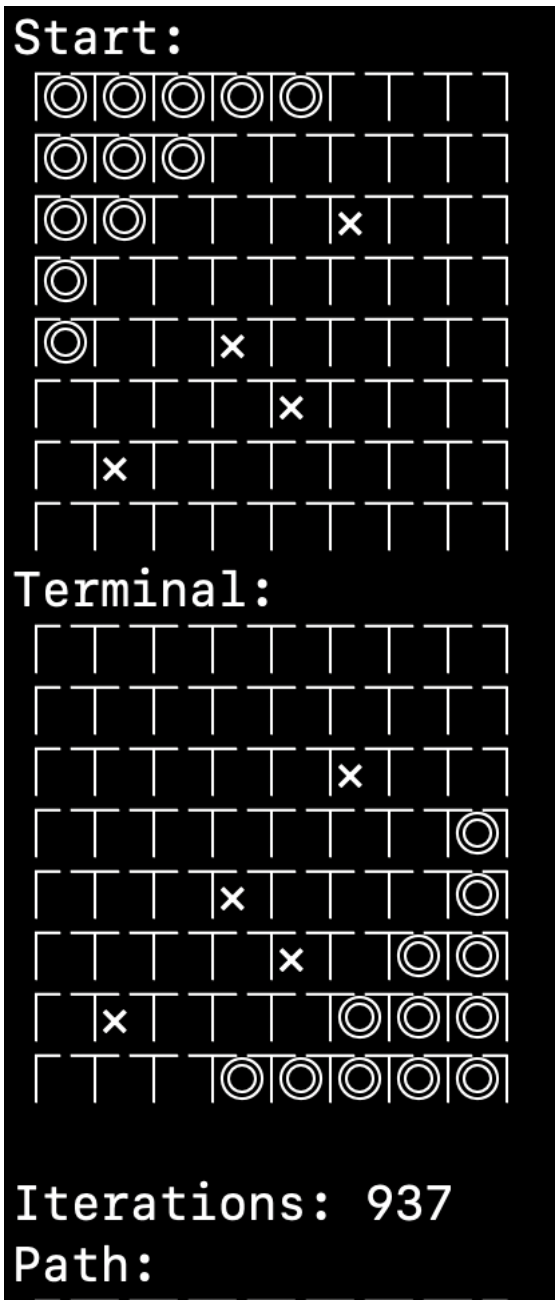
Для ввода начальной и конечной конфигурации используется файл в формате csv. Пример конфигурационного файла представлен в листинге 1.

Листинг 1. Представление конфигурации доски, представленной в задании, в формате csv. Первой строкой передаётся размер квадратной доски, далее записаны значения каждой клетки: 0 – свободная, 1 – занята, 2 – запрещена для посещения.

```
8
1 1 1 1 1 0 0 0
1 1 1 0 0 0 0 0
1 1 0 0 0 2 0 0
1 0 0 0 0 0 0 0
1 0 0 2 0 0 0 0
0 0 0 0 2 0 0 0
0 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Результат работы программы

В ходе решения программой были произведены 937 итераций. Результатом стала последовательность из 65 шагов, следуя которой можно передвинуть все фишки в правый нижний угол. Пример работы программы представлен на рисунке 2.



…

Рисунок 2 – Пример работы программы для доски размером 8 на 8

Приложение

Листинг 2. Исходный код программы.

main.go

```
package main

import (
	"fmt"
	"os"
	"src/a_search"
	"src/board"
	"time"
)

func main() {
	if len(os.Args) < 3 {
		println("usage: ./main <path_to_csv>")
		return
	}
	graphPathStart := os.Args[1]
	graphPathTerm := os.Args[2]

	start := board.NewBoardFromFile(graphPathStart)
	terminal := board.NewBoardFromFile(graphPathTerm)

	fmt.Println("Start:")
	start.Print()
	fmt.Println("Terminal:")
	terminal.Print()
	fmt.Println()

	s := time.Now()
	path, dur, ok := a_search.A(*start, *terminal)
	f := time.Since(s)

	if ok {
		fmt.Println("Path:")
		for i := len(path) - 1; i >= 0; i-- {
			path[i].Print()
			fmt.Println()
		}
		fmt.Printf("Len: %d\n", dur)
		fmt.Printf("Elapsed time: %s\n", f)
	} else {
		fmt.Println("Can't find solution")
	}
}
```

board/board.go

```
package board

import (
	"encoding/csv"
	"fmt"
	"os"
	"strconv"
	"strings"
)
```

```
const (
	FREE  rune = '0'
	BLACK rune = '1'
	TABOO rune = '2'
)

type Board struct {
	size  int
	Board string
}

func NewBoardFromFile(path string) *Board {
	csvConf, err := os.Open(path)
	if err != nil {
		panic(err)
	}
	defer func() {
		if err := csvConf.Close(); err != nil {
			panic(err)
		}
	}()

	reader := csv.NewReader(csvConf)
	reader.Comma = ' '
	reader.FieldsPerRecord = -1
	data, err := reader.ReadAll()
	if err != nil {
		panic(err)
	}

	var board Board
	board.size, _ = strconv.Atoi(data[0][0])
	data = data[1:]
	sb := strings.Builder{}
	for _, row := range data {
		for _, el := range row {
			sb.WriteString(el)
		}
	}

	board.Board = sb.String()

	return &board
}

func (b Board) GetNeighbours() []Board {
	var moves []Board
	for j := 0; j < b.size; j++ {
		for i := 0; i < b.size; i++ {
			if h := b.getHorizontalMoves(i, j); len(h) != 0 {
				moves = append(moves, h...)
			}
			if v := b.getVerticalMoves(i, j); len(v) != 0 {
				moves = append(moves, v...)
			}
		}
	}

	if len(moves) == 0 {
```

```
                return nil
        }

        return moves
}

func (b Board) getHorizontalMoves(x, y int) []Board {
        var moves []Board
        // right
        if b.isOnBoard(x+1, y) {
                if rune(b.Board[y*b.size+x+1]) == FREE {
                        moves = append(moves, b.getMove(x, y, x+1, y))
                }
        }
        if b.isOnBoard(x+2, y) &&
                rune(b.Board[y*b.size+x+1]) != FREE &&
                rune(b.Board[y*b.size+x+1]) != TABOO {
                if rune(b.Board[y*b.size+x+2]) == FREE {
                        moves = append(moves, b.getMove(x, y, x+2, y))
                }
        }
        // left
        if b.isOnBoard(x-1, y) {
                if rune(b.Board[y*b.size+x-1]) == FREE {
                        moves = append(moves, b.getMove(x, y, x-1, y))
                }
        }
        if b.isOnBoard(x-2, y) {
                if rune(b.Board[y*b.size+x-2]) == FREE &&
                        rune(b.Board[y*b.size+x-1]) != FREE &&
                        rune(b.Board[y*b.size+x-1]) != TABOO {
                        moves = append(moves, b.getMove(x, y, x-2, y))
                }
        }

        return moves
}

func (b Board) getVerticalMoves(x, y int) []Board {
        var moves []Board
        // top
        if b.isOnBoard(x, y+1) {
                if rune(b.Board[(y+1)*b.size+x]) == FREE {
                        moves = append(moves, b.getMove(x, y, x, y+1))
                }
        }
        if b.isOnBoard(x, y+2) &&
                rune(b.Board[(y+1)*b.size+x]) != FREE &&
                rune(b.Board[(y+1)*b.size+x]) != TABOO {
                if rune(b.Board[(y+2)*b.size+x]) == FREE {
                        moves = append(moves, b.getMove(x, y, x, y+2))
                }
        }
        // bottom
        if b.isOnBoard(x, y-1) {
                if rune(b.Board[(y-1)*b.size+x]) == FREE {
                        moves = append(moves, b.getMove(x, y, x, y-1))
                }
        }
        if b.isOnBoard(x, y-2) {
```

```
                if rune(b.Board[(y-2)*b.size+x]) == FREE &&
                        rune(b.Board[(y-1)*b.size+x]) != FREE &&
                        rune(b.Board[(y-1)*b.size+x]) != TABOO {
                        moves = append(moves, b.getMove(x, y, x, y-2))
                }
        }

        return moves
}

func (b Board) isOnBoard(x, y int) bool {
        if x < 0 || y < 0 || x > b.size-1 || y > b.size-1 {
                return false
        }

        return true
}

func (b Board) getMove(oldX, oldY, newX, newY int) Board {
        m := Board{size: b.size}
        r := []rune(b.Board)
        r[oldY*m.size+oldX], r[newY*m.size+newX] = FREE,
r[oldY*m.size+oldX]
        m.Board = string(r)

        return m
}

// Sum of manh distances of each checker to corner
func (b Board) Heuristic(to Board) int {
        res := 0
        for j := 0; j < b.size; j++ {
                for i := 0; i < b.size; i++ {
                        switch rune(b.Board[j*b.size+i]) {
                        case BLACK:
                                res += (b.size - 1 - i) + (b.size - 1 - j)
                        }
                }
        }

        return res
}

// Print pseudographic of board
var Symbols = map[rune]string{BLACK: "◎", TABOO: "╳", FREE: " "}

func (b Board) Print() {
        // First row
        fmt.Print("┌" + Symbols[rune(b.Board[0])])
        for i := 1; i < b.size; i++ {
                fmt.Print("┬" + Symbols[rune(b.Board[i])])
        }
        fmt.Print("┐\n")
        // Middle
        for j := 1; j < b.size-1; j++ {
                fmt.Print("├" + Symbols[rune(b.Board[j*b.size])])
                for i := 1; i < b.size; i++ {
                        fmt.Print("┼" + Symbols[rune(b.Board[j*b.size+i])])
                }
```

```
                fmt.Print("┐\n")
        }
        // Last row
        fmt.Print("└" + Symbols[rune(b.Board[b.size*(b.size-1)])])
        for i := 1; i < b.size; i++ {
                fmt.Print("┴" + Symbols[rune(b.Board[b.size*(b.size-
1)+i])])
        }
        fmt.Print("┘\n")
}
```

a_search/a.go

```
package a_search

import (
        "container/heap"
        "src/board"
)

func A(start, terminal board.Board) ([]board.Board, int, bool) {
        allNodes := nodeMap{}
        openedList := &priorityQueue{}
        heap.Init(openedList)

        // Init OPENED list
        fromNode := allNodes.get(start)
        fromNode.opened = true
        heap.Push(openedList, fromNode)

        for {
                // If there's no path -> failure
                if openedList.Len() == 0 {
                        return nil, 0, false
                }

                // Close best node
                current := heap.Pop(openedList).(*node)
                current.opened = false
                current.closed = true

                // If found end -> trace back and return path
                if current == allNodes.get(terminal) {
                        var p []board.Board
                        curr := current
                        for curr != nil {
                                p = append(p, curr.board)
                                curr = curr.parent
                        }
                        return p, current.cost, true
                }

                for _, neighbour := range current.board.GetNeighbours() {
                        cost := current.cost + 1
                        neighborNode := allNodes.get(neighbour)
                        // If already in OPENED -> check if cost is lower
                        if cost < neighborNode.cost {
                                if neighborNode.opened {
                                        heap.Remove(openedList,
neighborNode.index)
                                }
```

```
                                neighborNode.opened = false
                                neighborNode.closed = false
                        }
                        // If completely new node -> add to OPENED
                        if !neighborNode.opened && !neighborNode.closed {
                                neighborNode.cost = cost
                                neighborNode.opened = true
                                neighborNode.rank = cost +
neighbour.Heuristic(terminal)
                                neighborNode.parent = current
                                heap.Push(openedList, neighborNode)
                        }
                }
        }
}
```

a_search/node.go

```
package a_search

import "src/board"

type node struct {
        board  board.Board
        parent *node
        cost   int
        opened bool
        closed bool
        index  int
        rank   int
}

type nodeMap map[board.Board]*node

func (nm nodeMap) get(p board.Board) *node {
        n, ok := nm[p]
        if !ok {
                n = &node{
                        board: p,
                }
                nm[p] = n
        }
        return n
}
```

a_search/priorityQueue.go

```
package a_search

type priorityQueue []*node

func (pq priorityQueue) Len() int {
        return len(pq)
}

func (pq priorityQueue) Less(i, j int) bool {
        return pq[i].rank < pq[j].rank
}

func (pq priorityQueue) Swap(i, j int) {
        pq[i], pq[j] = pq[j], pq[i]
```

```
        pq[i].index = i
        pq[j].index = j
}

func (pq *priorityQueue) Push(x interface{}) {
        n := len(*pq)
        no := x.(*node)
        no.index = n
        *pq = append(*pq, no)
}

func (pq *priorityQueue) Pop() interface{} {
        old := *pq
        n := len(old)
        no := old[n-1]
        no.index = -1
        *pq = old[0 : n-1]

        return no
}
```