Харківський національний університет радіоелектроніки

Методи пошуку найкоротшого шляху проходження лабіринтів за допомогою засобів обчислювального інтелекту

Виконав:
магістрант групи КСМзм-20-1 Солонцевой Д.М.

Науковий керівник:
доцент каф. ЕОМ Іващенко Г.С.

Харківський національний університет радіоелектроніки
Кафедра ЕОМ, тел. 702-13-54

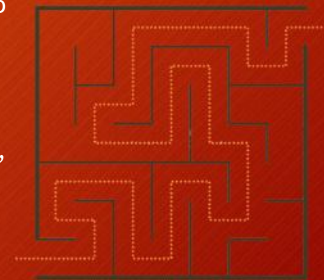
Актуальність проблеми

2

Однією з основних задач, які розв'язуються при проектуванні різноманітних систем (наприклад, комп'ютерних мереж, ігрового світу), є забезпечення можливості пошуку найкоротшого шляху між вузлами цієї системи. Для цього використовуються спеціальні алгоритми, але вибір не є очевидним.

Одним із видів представлення системи, на якій виконується пошук шляху, є лабіринт. Дана структура може мати безліч умов, що ускладнюють пошук рішення: динамічні зміні у схемі лабіринту, часові обмеження, умовна «вартість» переходу між певними точками.

Тому актуальною є проблема вибору алгоритму пошуку у лабіринтах, який забезпечує оптимальний шлях.



Існуючі методи пошуку шляху

3

Жадібні алгоритми:
- Дейкстри
- пошук A*
- жадібний пошук
- хвильовий алгоритм (Лі)

Засоби обчислювального інтелекту:
- генетичний алгоритм
- мурашина колонія
- імітація відпалу

Постановка задачі

4

Метою роботи є проведення аналізу існуючих алгоритмів пошуку шляхів у контексті вирішення лабіринтів.

Для виявлення переваг та недоліків алгоритмів були враховані наступні умови:
- наявність «кімнат» (ділянок вільного простору без «стін»)
- наявність пасток
- наявність тупиків та їх кількість
- наявність різниці між ребрами у вазі
- інші додаткові умови

Перелік використаних технологій

5

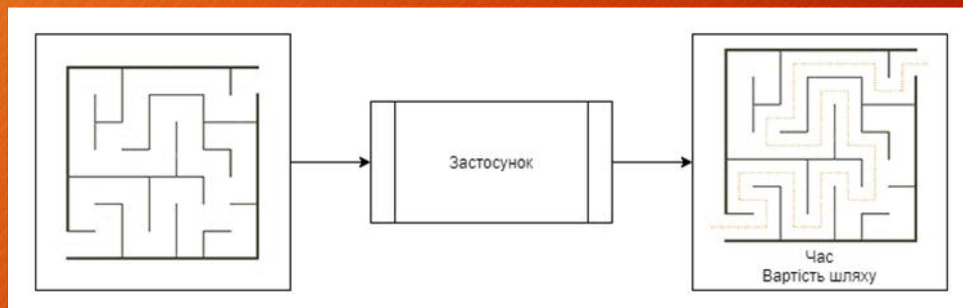
Задля проведення ретельного аналізу було вирішено розробити застосунок, який приймає у якості вхідних даних певний лабіринт, а на виході показує час обробки та обраний шлях.

Для написання застосунку були використані наступні технології та інструментальні засоби:

- мова програмування Java
- середовище розробки IntelliJIDEA Community 2020.3.3
- Apache Maven

Процес роботи застосунку

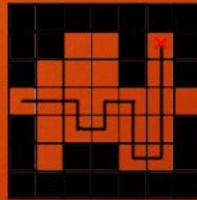
6

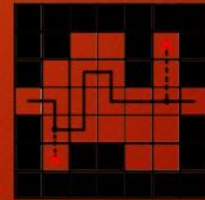


Стратегія дослідження лабіринту

7

Випадковий пошук


Пошук із списком табу

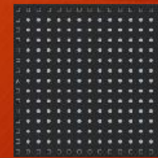

Модифікований пошук

Тестові дані

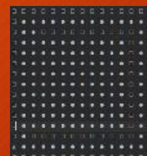
8

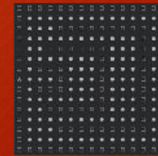
Коридор

Сніжинка


Вільне поле

Зал колон


Кут


Пастка

9

Результати тесту «Коридор»

10

Назва алгоритму	Шлях (незв.)	Час (незв.)	Шлях (зв.)	Час (зв.)
Алгоритм Дейкстри	15	0.181	15	0.164
Пошук A*	15	0.28	15	0.233
Жадібний пошук	15	0.376	15	0.265
Алгоритм Лі	15	0.158	15	0.141
Генетичний алгоритм	15	0.073	15	0.084
Мурашиний алгоритм	15	0.103	15	0.092
Імітація відпалу	15	0.122	15	0.119

Результати тесту «Сніжинка»

11

Назва алгоритму	Шлях (незв.)	Час (незв.)	Шлях (зв.)	Час (зв.)
Алгоритм Дейкстри	15	0.888	15	0.627
Пошук A*	15	0.29	15	0.614
Жадібний пошук	15	0.485	15	0.375
Алгоритм Лі	15	0.611	15	0.625
Генетичний алгоритм	15	0.708	15	0.485
Мурашиний алгоритм	15	0.98	15	0.654
Імітація відпалу	15	0.822	15	0.305

Результати тесту «Вільне поле»

12

Назва алгоритму	Шлях (незв.)	Час (незв.)	Шлях (зв.)	Час (зв.)
Алгоритм Дейкстри	15	2.587	48	2.577
Пошук A*	15	0.39	48	2.05
Жадібний пошук	15	0.564	51	0.423
Алгоритм Лі	15	2.009	59	2.243
Генетичний алгоритм	19	69.743	73.6	51.144
Мурашиний алгоритм	38.2	31.164	146.4	31.069
Імітація відпалу	33.4	52.553	132.2	38.434

Результати тесту «Зал колон»

13

Назва алгоритму	Шлях (незв.)	Час (незв.)	Шлях (зв.)	Час (зв.)
Алгоритм Дейкстри	15	1.636	67	1.718
Пошук А*	15	0.373	67	2.344
Жадібний пошук	15	0.533	78	0.3
Алгоритм Лі	15	1.95	72	1.529
Генетичний алгоритм	15.8	40.674	74.6	37.274
Мурашиний алгоритм	23.8	30.088	97.8	24.404
Імітація відпалу	23	41.969	100	35.085

Результати тесту «Кут»

14

Назва алгоритму	Шлях (незв.)	Час (незв.)	Шлях (зв.)	Час (зв.)
Алгоритм Дейкстри	21	2.002	88	2.614
Пошук А*	21	1.317	88	1.905
Жадібний пошук	21	0.377	88	0.282
Алгоритм Лі	21	2.006	88	2.028
Генетичний алгоритм	21	2.937	88	2.205
Мурашиний алгоритм	21	4.82	88	1.782
Імітація відпалу	21	0.356	88	1.998

Результати тесту «Пастка»

15

Назва алгоритму	Шлях (незв.)	Час (незв.)	Шлях (зв.)	Час (зв.)
Алгоритм Дейкстри	27	1.237	109	1.28
Пошук A*	27	1.202	109	1.362
Жадібний пошук	41	1.158	177	1.569
Алгоритм Лі	27	1.107	113	1.177
Генетичний алгоритм	27	5.969	109	29.838
Мурашиний алгоритм	27	8.215	115.4	21.589
Імітація відпалу	27	10.88	112.8	22.048

Результати тесту на великому лабіринті

16

Назва алгоритму	Шлях	Час
Алгоритм Дейкстри	100	27.793
Пошук A*	100	7.369
Жадібний пошук	116	2.509
Алгоритм Лі	100	22.827
Генетичний алгоритм	105.6	1174.659
Мурашиний алгоритм	130.4	658.94
Імітація відпалу	126.8	1006.667

Висновки

17

Згідно отриманих результатів можна зробити наступні висновки:

- засоби обчислювального інтелекту підходять для ситуацій, коли можливо віднайти шлях випадковими спробами, а не виконувати повний перебір
- на відкритих просторах засоби обчислювального інтелекту досить довго «блукають» в пошуках найліпшого шляху, тому в даних ситуаціях жадібні методи є більш доречним рішенням
- засоби обчислювального інтелекту також підходять у тих ситуаціях, коли пошук шляху необхідно проводити в реальному часі

Результати роботи представлені у рамках Дев’ятої міжнародної науково-технічної конференції "Проблеми інформатизації".

.1

.1.1

```
package pathfinders.greedy;

import dto.ShortestPathDTO;
import graph.Graph;
import dto.PathLinkDTO;
import graph.Transition;

import java.util.*;
import java.util.function.BiFunction;

public class DijkstraBasic implements BiFunction<Graph, Integer,
ShortestPathDTO> {

    @Override
    public ShortestPathDTO apply(Graph graph, Integer optimalCost) {
        var valueLabels = new HashMap<String, PathLinkDTO>();
        valueLabels.put(graph.getEntranceVertexLabel(), new
PathLinkDTO(graph.getEntranceVertexLabel(), 0));

        var settledValueVertices = new HashSet<String>();

        while (settledValueVertices.size() < graph.getAllVertices().size()) {
            String currentNodeLabel;
            try {
                currentNodeLabel = valueLabels.entrySet().stream()
                        .filter(entry ->
!settledValueVertices.contains(entry.getKey()))
                        .min(Comparator.comparingInt(value ->
valueLabels.get(value.getKey()).getValue()))
                        .get()
                        .getKey();
            } catch (Exception e) {
                break;
            }

            var transitions =
graph.getTransitionsByVertexLabel(currentNodeLabel);

            for (Transition transition : transitions) {
                Integer value = Integer.MAX_VALUE;

                if (valueLabels.containsKey(transition.getDestination())) {
                    value =
valueLabels.get(transition.getDestination()).getValue();
                } else {
                    valueLabels.put(
                            transition.getDestination(),
```

```
                            new PathLinkDTO(currentNodeLabel,
valueLabels.get(currentNodeLabel).getValue() + transition.getWeight())
                    );
                }

                if (valueLabels.get(currentNodeLabel).getValue() +
transition.getWeight() < value) {
                    if (value != Integer.MAX_VALUE) {

settledValueVertices.remove(transition.getDestination());
                    }

                    value = valueLabels.get(currentNodeLabel).getValue() +
transition.getWeight();
                    valueLabels.put(
                            transition.getDestination(),
                            new PathLinkDTO(currentNodeLabel, value)
                    );
                }
            }

            settledValueVertices.add(currentNodeLabel);
        }

        var pathDTO = new ShortestPathDTO();

pathDTO.setCost(valueLabels.get(graph.getExitVertexLabel()).getValue());

        var shortestPath = new ArrayList<String>();
        shortestPath.add(graph.getEntranceVertexLabel());

        var backtrackCursor = graph.getExitVertexLabel();
        while (!backtrackCursor.equals(graph.getEntranceVertexLabel())) {
            shortestPath.add(backtrackCursor);
            backtrackCursor =
valueLabels.get(backtrackCursor).getLabelFrom();
        }

        pathDTO.setPath(shortestPath);

        return pathDTO;
    }
}
```

.1.2 *

```
package pathfinders.greedy;

import dto.HeuristicNodeDTO;
import dto.ShortestPathDTO;
import graph.Graph;
import graph.Transition;

import java.util.*;
import java.util.function.BiFunction;
import java.util.function.Function;

public class AStarBasic implements BiFunction<Graph, Integer,
ShortestPathDTO> {
```

```
    @Override
    public ShortestPathDTO apply(Graph graph, Integer optimalCost) {
        var valueLabels = new HashMap<String, HeuristicNodeDTO>();
        var openSet = new HashSet<String>();

        openSet.add(graph.getEntranceVertexLabel());
        valueLabels.put(graph.getEntranceVertexLabel(), new
HeuristicNodeDTO(graph.getEntranceVertexLabel(), 0, 0));

        var closedSet = new HashSet<String>();

        while (!openSet.isEmpty()) {
            var currentNodeLabel = openSet.stream()
                    .min(Comparator.comparingInt(x ->
valueLabels.get(x).getTotalValue()))
                    .get();

            openSet.remove(currentNodeLabel);

            var transitions =
graph.getTransitionsByVertexLabel(currentNodeLabel);

            for (Transition transition : transitions) {
                var neighbourParams = new HeuristicNodeDTO();
                neighbourParams.setParentLabel(currentNodeLabel);

neighbourParams.setFromSourceCostValue(valueLabels.get(currentNodeLabel).getF
romSourceCostValue() + transition.getWeight());

neighbourParams.setHeuristicGoalDistance(calculateHeuristics(transition.getDe
stination(), graph.getExitVertexLabel()));

                if (openSet.contains(transition.getDestination())
                        &&
valueLabels.get(transition.getDestination()).getTotalValue()
                        <= neighbourParams.getTotalValue()) {
                    continue;
                }

                if (closedSet.contains(transition.getDestination())
                        &&
valueLabels.get(transition.getDestination()).getTotalValue()
                        <= neighbourParams.getTotalValue()) {
                    continue;
                } else {
                    openSet.add(transition.getDestination());
                }

                valueLabels.put(transition.getDestination(),
neighbourParams);

                if
(transition.getDestination().equals(graph.getExitVertexLabel())) {
                    openSet.clear();
                    break;
                }
            }

            closedSet.add(currentNodeLabel);
        }

        var pathDTO = new ShortestPathDTO();

pathDTO.setCost(valueLabels.get(graph.getExitVertexLabel()).getFromSourceCost
```

```
Value());

        var shortestPath = new ArrayList<String>();
        shortestPath.add(graph.getEntranceVertexLabel());

        var backtrackCursor = graph.getExitVertexLabel();
        while (!backtrackCursor.equals(graph.getEntranceVertexLabel())) {
            shortestPath.add(backtrackCursor);
            backtrackCursor =
valueLabels.get(backtrackCursor).getParentLabel();
        }

        pathDTO.setPath(shortestPath);

        return pathDTO;
    }

    private Integer calculateHeuristics(String source, String goal) {
        var sourceCoordinates = source.split("_");
        var goalCoordinates = goal.split("_");

        var sourceX = Integer.parseInt(sourceCoordinates[0]);
        var sourceY = Integer.parseInt(sourceCoordinates[1]);
        var goalX = Integer.parseInt(goalCoordinates[0]);
        var goalY = Integer.parseInt(goalCoordinates[1]);

        return Math.abs(sourceX - goalX) + Math.abs(sourceY - goalY);
    }
}
```

.1.3

```
package pathfinders.greedy;

import dto.HeuristicNodeDTO;
import dto.ShortestPathDTO;
import graph.Graph;

import java.util.*;
import java.util.function.BiFunction;

public class GreedySearchBasic implements BiFunction<Graph, Integer,
ShortestPathDTO> {

    @Override
    public ShortestPathDTO apply(Graph graph, Integer optimalCost) {
        var valueLabels = new HashMap<String, HeuristicNodeDTO>();
        var openSet = new HashSet<String>();
        openSet.add(graph.getEntranceVertexLabel());
        valueLabels.put(graph.getEntranceVertexLabel(), new
HeuristicNodeDTO(graph.getEntranceVertexLabel(), 0, 0));

        var closedSet = new HashSet<String>();

        while (!openSet.isEmpty()) {
            var currentNodeLabel = openSet.stream()
                    .min(Comparator.comparingInt(x ->
valueLabels.get(x).getHeuristicGoalDistance()))
                    .get();
```

```
            openSet.remove(currentNodeLabel);

            var transitions =
graph.getTransitionsByVertexLabel(currentNodeLabel);

            if (transitions.stream().allMatch(transition ->
closedSet.contains(transition.getDestination()))) {
                closedSet.add(currentNodeLabel);
                continue;
            }

            transitions.forEach(transition -> {
                if (!closedSet.contains(transition.getDestination())) {
                    var neighbourParams = new HeuristicNodeDTO();
                    neighbourParams.setParentLabel(currentNodeLabel);

neighbourParams.setFromSourceCostValue(valueLabels.get(currentNodeLabel).getF
romSourceCostValue() + transition.getWeight());

neighbourParams.setHeuristicGoalDistance(calculateHeuristics(transition.getDe
stination(), graph.getExitVertexLabel()));

                    valueLabels.put(transition.getDestination(),
neighbourParams);

                    openSet.add(transition.getDestination());
                }
            });

            var bestOfAllTransition = transitions.stream()
                    .filter(transition ->
!closedSet.contains(transition.getDestination()))
                    .min(Comparator.comparingInt(x ->
calculateHeuristics(x.getDestination(), graph.getExitVertexLabel())))
                    .get();
            if
(bestOfAllTransition.getDestination().equals(graph.getExitVertexLabel())) {
                openSet.clear();
                break;
            }

            closedSet.add(currentNodeLabel);
        }

        var pathDTO = new ShortestPathDTO();

pathDTO.setCost(valueLabels.get(graph.getExitVertexLabel()).getFromSourceCost
Value());

        var shortestPath = new ArrayList<String>();
        shortestPath.add(graph.getEntranceVertexLabel());

        var backtrackCursor = graph.getExitVertexLabel();
        while (!backtrackCursor.equals(graph.getEntranceVertexLabel())) {
            shortestPath.add(backtrackCursor);
            backtrackCursor =
valueLabels.get(backtrackCursor).getParentLabel();
        }

        pathDTO.setPath(shortestPath);

        return pathDTO;
    }
```

```
    private Integer calculateHeuristics(String source, String goal) {
        var sourceCoordinates = source.split("_");
        var goalCoordinates = goal.split("_");

        var sourceX = Integer.parseInt(sourceCoordinates[0]);
        var sourceY = Integer.parseInt(sourceCoordinates[1]);
        var goalX = Integer.parseInt(goalCoordinates[0]);
        var goalY = Integer.parseInt(goalCoordinates[1]);

        return Math.abs(sourceX - goalX) + Math.abs(sourceY - goalY);
    }
}
```

.1.4

```
package pathfinders.greedy;

import dto.PathLinkDTO;
import dto.ShortestPathDTO;
import graph.Graph;
import graph.Transition;
import graph.Vertex;

import java.util.*;
import java.util.function.BiFunction;
import java.util.stream.Collectors;

public class LeeBasic implements BiFunction<Graph, Integer, ShortestPathDTO>
{

    @Override
    public ShortestPathDTO apply(Graph graph, Integer optimalCost) {
        var valueLabels = new HashMap<String, PathLinkDTO>();
        valueLabels.put(graph.getEntranceVertexLabel(), new
PathLinkDTO(graph.getEntranceVertexLabel(), 0));

        var unsettledValueVertices = graph.getAllVertices().stream()
                .map(Vertex::getLabel)
                .collect(Collectors.toCollection(HashSet::new));

        while (!unsettledValueVertices.isEmpty()) {
            var currentNodeLabel = unsettledValueVertices.stream()
                    .filter(valueLabels::containsKey)
                    .min(Comparator.comparingInt(x ->
valueLabels.get(x).getValue()))
                    .get();

            var transitions =
graph.getTransitionsByVertexLabel(currentNodeLabel);

            for (Transition transition : transitions) {
                if
(unsettledValueVertices.contains(transition.getDestination())) {
                    valueLabels.put(transition.getDestination(),
                            new PathLinkDTO(currentNodeLabel,
valueLabels.get(currentNodeLabel).getValue() + transition.getWeight()));
                    if
(transition.getDestination().equals(graph.getExitVertexLabel())) {
                        unsettledValueVertices.clear();
                        break;
```

```
                }
            }
        }

        unsettledValueVertices.remove(currentNodeLabel);
    }

    var pathDTO = new ShortestPathDTO();

pathDTO.setCost(valueLabels.get(graph.getExitVertexLabel()).getValue());

    var shortestPath = new ArrayList<String>();
    shortestPath.add(graph.getEntranceVertexLabel());

    var backtrackCursor = graph.getExitVertexLabel();
    while (!backtrackCursor.equals(graph.getEntranceVertexLabel())) {
        shortestPath.add(backtrackCursor);
        backtrackCursor =
valueLabels.get(backtrackCursor).getLabelFrom();
    }

    pathDTO.setPath(shortestPath);

    return pathDTO;
  }
}
```

.1.5

```
package pathfinders.computational;

import dto.ShortestPathDTO;
import graph.Graph;
import graph.Vertex;

import java.util.*;
import java.util.function.BiFunction;
import java.util.stream.Collectors;

public class GeneticBasic implements BiFunction<Graph, Integer,
ShortestPathDTO> {

    @Override
    public ShortestPathDTO apply(Graph graph, Integer optimalCost) {
        var initialGeneration = new ArrayList<ShortestPathDTO>();

        for (int i = 0; i < 50; i++) {
            var foundBest = tryToAddNewCreatureTo(initialGeneration, graph,
optimalCost);
            if (foundBest != null) {
                return foundBest;
            }
        }

        ShortestPathDTO result = new ShortestPathDTO(new ArrayList<>(),
Integer.MAX_VALUE);

        List<ShortestPathDTO> generation = new
ArrayList<>(initialGeneration);
        var numOfGens = 0;
```

```
        while (numOfGens < 35) {
            crossingCycle(generation, graph);

            generation = generation.stream()

.sorted(Comparator.comparingInt(ShortestPathDTO::getCost))
                    .limit(25)
                    .collect(Collectors.toList());

            result =
generation.stream().min(Comparator.comparingInt(ShortestPathDTO::getCost)).ge
t();
            if (result.getCost() <= optimalCost) {
                return result;
            }

            numOfGens++;
        }

        return result;
    }

    private void crossingCycle(List<ShortestPathDTO> generation, Graph graph)
{
        for (int i = 0; i < 25; i++) {
            var seedM = Double.valueOf(Math.random() *
generation.size()).intValue();
            var seedF = Double.valueOf(Math.random() *
generation.size()).intValue();

            while (seedF == seedM) {
                seedF = Double.valueOf(Math.random() *
generation.size()).intValue();
            }

            var paternalSpecimen = generation.get(seedM);
            var maternalSpecimen = generation.get(seedF);

            if (paternalSpecimen.getPath().size() > 5 &&
maternalSpecimen.getPath().size() > 5) {
                var crossMPos = -1;
                var crossFPos = -1;

                var zoomMFactor = Double.valueOf(Math.random() *
paternalSpecimen.getPath().size() / 4 + 2).intValue();
                var zoomFFactor = Double.valueOf(Math.random() *
maternalSpecimen.getPath().size() / 4 + 2).intValue();

                for (int j = zoomMFactor; j <
paternalSpecimen.getPath().size() - zoomMFactor; j++) {
                    if (maternalSpecimen.getPath().subList(zoomFFactor,
maternalSpecimen.getPath().size() -
zoomFFactor).contains(paternalSpecimen.getPath().get(j))) {
                        crossMPos = j;
                        crossFPos =
maternalSpecimen.getPath().indexOf(paternalSpecimen.getPath().get(j));

                        break;
                    }
                }

                if (crossMPos > -1) {
                    var childSpecimen = new ShortestPathDTO();
                    childSpecimen.setCost(0);
```

```
                    childSpecimen.setPath(new ArrayList<>());

childSpecimen.getPath().add(paternalSpecimen.getPath().get(0));

                    for (int j = 0; j < crossMPos; j++) {
                        var current = paternalSpecimen.getPath().get(j);
                        var next = paternalSpecimen.getPath().get(j+1);
                        var costOfTransition =
graph.getTransitionsByVertexLabel(current).stream()
                                .filter(x -> x.getDestination().equals(next))
                                .findFirst()
                                .get()
                                .getWeight();
                        childSpecimen.getPath().add(next);
                        childSpecimen.addCost(costOfTransition);
                    }

                    for (int j = crossFPos; j <
maternalSpecimen.getPath().size() - 1; j++) {
                        var current = maternalSpecimen.getPath().get(j);
                        var next = maternalSpecimen.getPath().get(j+1);
                        var costOfTransition =
graph.getTransitionsByVertexLabel(current).stream()
                                .filter(x -> x.getDestination().equals(next))
                                .findFirst()
                                .get()
                                .getWeight();
                        childSpecimen.getPath().add(next);
                        childSpecimen.addCost(costOfTransition);
                    }

                    removeKnots(childSpecimen, graph);
                    if (childSpecimen.getCost() < paternalSpecimen.getCost()
&& childSpecimen.getCost() < maternalSpecimen.getCost()) {
                        generation.add(childSpecimen);
                    }
                }
            }
        }
    }

    private ShortestPathDTO tryToAddNewCreatureTo(ArrayList<ShortestPathDTO>
initialGeneration, Graph graph, Integer optimalCost) {
        var specimen = new ShortestPathDTO();
        specimen.setPath(new ArrayList<>());
        specimen.setCost(0);

        var visitedMap = new HashMap<String, Integer>();
        for (Vertex vertex : graph.getAllVertices()) {
            visitedMap.put(vertex.getLabel(), -1);
        }
        visitedMap.put(graph.getEntranceVertexLabel(), 0);

        var cursor = graph.getEntranceVertexLabel();
        specimen.getPath().add(graph.getEntranceVertexLabel());

        while (!cursor.equals(graph.getExitVertexLabel())) {
            var possibleTransitions =
graph.getTransitionsByVertexLabel(cursor).stream()
                    .filter(transition ->
visitedMap.get(transition.getDestination()) < 0)
                    .collect(Collectors.toList());

            if (possibleTransitions.size() < 1) {
```

```
                var resetTo = cursor;
                var found = false;
                var pathIterator = specimen.getPath().size() - 2;

                while (!found) {
                    resetTo = specimen.getPath().get(pathIterator);
                    var possiblePreviousTransitions =
graph.getTransitionsByVertexLabel(resetTo).stream()
                            .filter(transition ->
visitedMap.get(transition.getDestination()) < 0)
                            .collect(Collectors.toList());
                    if (possiblePreviousTransitions.size() > 0) {
                        found = true;
                    }
                    pathIterator--;
                }

                var prevCursor =
specimen.getPath().get(specimen.getPath().size() - 2);
                var backtickCursor =
specimen.getPath().get(specimen.getPath().size() - 1);
                while (!backtickCursor.equals(resetTo)) {
                    var finalBacktickCursor = backtickCursor;
                    var costOfTransition =
graph.getTransitionsByVertexLabel(prevCursor).stream()
                            .filter(x ->
x.getDestination().equals(finalBacktickCursor))
                            .findFirst()
                            .get()
                            .getWeight();
                    specimen.addCost(-costOfTransition);
                    specimen.getPath().remove(backtickCursor);

                    prevCursor =
specimen.getPath().get(specimen.getPath().size() - 2);
                    backtickCursor =
specimen.getPath().get(specimen.getPath().size() - 1);
                }
                cursor = resetTo;
            } else {
                var seed = 0;
                if (possibleTransitions.size() > 1) {
                    seed = Double.valueOf(Math.random() *
possibleTransitions.size()).intValue();
                }

                var chosenTransition = possibleTransitions.get(seed);

                cursor = chosenTransition.getDestination();
                visitedMap.put(cursor, visitedMap.get(cursor) + 1);

                specimen.getPath().add(chosenTransition.getDestination());
                specimen.addCost(chosenTransition.getWeight());
            }
        }

        if (specimen.getCost() <= optimalCost) {
            return specimen;
        } else {
            initialGeneration.add(specimen);
            return null;
        }
    }
```

```
    private void removeKnots(ShortestPathDTO result, Graph graph) {
        var knotDetected = true;

        while (knotDetected) {
            var checkList = new LinkedHashSet<String>();

            for (int i = 0; i < result.getPath().size(); i++) {
                var element = result.getPath().get(i);
                if (!checkList.add(element)) {
                    var knotStart = result.getPath().get(new
ArrayList<>(checkList).indexOf(element));
                    var knotEnd = element;

                    var prevCursor = result.getPath().get(i - 1);
                    var backtickCursor = element;
                    while (!backtickCursor.equals(knotStart)) {
                        var finalBacktickCursor = backtickCursor;
                        var costOfTransition =
graph.getTransitionsByVertexLabel(prevCursor).stream()
                                .filter(x ->
x.getDestination().equals(finalBacktickCursor))
                                .findFirst()
                                .get()
                                .getWeight();
                        result.addCost(-costOfTransition);
                        result.getPath().remove(backtickCursor);

                        i--;

                        prevCursor = result.getPath().get(i - 1);
                        backtickCursor = result.getPath().get(i);
                    }

                    break;
                }
            }

            knotDetected = false;
        }
    }
}
```

.1.6

```
package pathfinders.computational;

import dto.ShortestPathDTO;
import graph.Graph;
import graph.Transition;
import graph.Vertex;

import java.util.*;
import java.util.function.BiFunction;
import java.util.stream.Collectors;

public class AntBasic implements BiFunction<Graph, Integer, ShortestPathDTO>
{

    @Override
    public ShortestPathDTO apply(Graph graph, Integer optimalCost) {
```

```
        var a = 1;
        var b = 3;

        var pheromoneValues = new HashMap<String, Double>();
        initPheromone(pheromoneValues, graph);

        ShortestPathDTO result = new ShortestPathDTO(new ArrayList<>(),
Integer.MAX_VALUE);

        var numberOfIterations = 0;
        while (numberOfIterations < 5) {
            var antPaths = new ArrayList<ShortestPathDTO>();

            for (int i = 0; i < 10; i++) {
                var antPath = newAnt(graph, pheromoneValues, a, b);
                if (antPath.getCost() <= optimalCost) {
                    return antPath;
                }

                for (String vertex : antPath.getPath()) {
                    pheromoneValues.put(vertex, pheromoneValues.get(vertex) +
optimalCost/antPath.getCost());
                }
                antPaths.add(antPath);
            }

            updatePheromone(pheromoneValues, graph);

            result =
antPaths.stream().min(Comparator.comparingInt(ShortestPathDTO::getCost)).get(
);
            if (result.getCost() <= optimalCost) {
                break;
            }

            numberOfIterations++;
        }

        return result;
    }

    private void initPheromone(HashMap<String, Double> pheromoneValues, Graph
graph) {
        for (Vertex vertex : graph.getAllVertices()) {
            pheromoneValues.put(vertex.getLabel(), 1D);
        }
    }

    private void updatePheromone(HashMap<String, Double> pheromoneValues,
Graph graph) {
        var evaporationRate = 0.6;

        for (Vertex vertex : graph.getAllVertices()) {
            pheromoneValues.put(vertex.getLabel(),
pheromoneValues.get(vertex.getLabel()) * evaporationRate);
        }
    }

    private ShortestPathDTO newAnt(Graph graph, HashMap<String, Double>
pheromoneValues, Integer a, Integer b) {
        var visitedMap = new HashMap<String, Integer>();
        for (Vertex vertex : graph.getAllVertices()) {
            visitedMap.put(vertex.getLabel(), -1);
        }
```

```
        visitedMap.put(graph.getEntranceVertexLabel(), 0);

        var antPath = new ShortestPathDTO();
        antPath.setCost(0);
        antPath.setPath(new ArrayList<>());
        antPath.getPath().add(graph.getEntranceVertexLabel());
        var ant = graph.getEntranceVertexLabel();

        while (!ant.equals(graph.getExitVertexLabel())) {
            var possibleTransitions =
graph.getTransitionsByVertexLabel(ant).stream()
                    .filter(transition ->
visitedMap.get(transition.getDestination()) == -1)
                    .collect(Collectors.toList());

            if (possibleTransitions.size() < 1) {
                var resetTo = ant;
                var found = false;
                var pathIterator = antPath.getPath().size() - 2;

                while (!found) {
                    resetTo = antPath.getPath().get(pathIterator);
                    var possiblePreviousTransitions =
graph.getTransitionsByVertexLabel(resetTo).stream()
                            .filter(transition ->
visitedMap.get(transition.getDestination()) < 0)
                            .collect(Collectors.toList());
                    if (possiblePreviousTransitions.size() > 0) {
                        found = true;
                    }
                    pathIterator--;
                }

                var prevCursor =
antPath.getPath().get(antPath.getPath().size() - 2);
                var backtickCursor =
antPath.getPath().get(antPath.getPath().size() - 1);
                while (!backtickCursor.equals(resetTo)) {
                    var finalBacktickCursor = backtickCursor;
                    var costOfTransition =
graph.getTransitionsByVertexLabel(prevCursor).stream()
                            .filter(x ->
x.getDestination().equals(finalBacktickCursor))
                            .findFirst()
                            .get()
                            .getWeight();
                    antPath.addCost(-costOfTransition);
                    antPath.getPath().remove(backtickCursor);

                    prevCursor =
antPath.getPath().get(antPath.getPath().size() - 2);
                    backtickCursor =
antPath.getPath().get(antPath.getPath().size() - 1);
                }
                ant = resetTo;
            } else {
                if (possibleTransitions.size() > 1) {
                    var totalAttraction = 0D;
                    var attractionMap = new HashMap<String, Double>();
                    for (Transition transition : possibleTransitions) {
                        var cost =
Math.pow(pheromoneValues.get(transition.getDestination()), a) + 1 /
Math.pow(transition.getWeight(), b);
                        totalAttraction += cost;
```

```
                    attractionMap.put(transition.getDestination(),
totalAttraction);
                }

                var comparators = attractionMap.entrySet().stream()

.sorted(Comparator.comparingDouble(Map.Entry::getValue))
                        .collect(Collectors.toList());
                var seed = Math.random() * totalAttraction;

                for (Map.Entry<String, Double> entry : comparators) {
                    if (seed <= entry.getValue()) {
                        ant = entry.getKey();
                        visitedMap.put(ant, visitedMap.get(ant) + 1);
                        antPath.getPath().add(ant);
                        var weight = possibleTransitions.stream()
                                .filter(transition ->
transition.getDestination().equals(entry.getKey()))
                                .findFirst()
                                .get()
                                .getWeight();
                        antPath.addCost(weight);
                        break;
                    }
                }
            } else {
                var transition = possibleTransitions.get(0);

                ant = transition.getDestination();
                visitedMap.put(ant, visitedMap.get(ant) + 1);

                antPath.getPath().add(ant);
                antPath.addCost(transition.getWeight());
            }
        }
    }

    return antPath;
  }
}
```

.1.7

```
package pathfinders.computational;

import dto.ShortestPathDTO;
import graph.Graph;
import graph.Vertex;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.function.BiFunction;
import java.util.stream.Collectors;

public class AnnealingBasic implements BiFunction<Graph, Integer,
ShortestPathDTO> {

    @Override
    public ShortestPathDTO apply(Graph graph, Integer optimalCost) {
        var temperature = 100;
```

```
        var result = generateInitialResult(graph);

        while (temperature > 0 && result.getCost() > optimalCost) {
            ShortestPathDTO newResult;
            if (result.getPath().size() > 10) {
                newResult = tryImprovingResult(result, graph);
            } else {
                break;
            }

            var delta = Double.valueOf(newResult.getCost() -
result.getCost());

            var doTransition = false;
            if (delta < 0) {
                doTransition = true;
            } else {
                var transitionChance = Math.exp(-delta/temperature);
                var seed = Math.random();
                if (seed <= transitionChance) {
                    doTransition = true;
                }
            }

            if (doTransition) {
                result = newResult;
            }

            if (result.getCost() <= optimalCost) {
                return result;
            }

            temperature--;
        }

        return result;
    }

    private ShortestPathDTO tryImprovingResult(ShortestPathDTO result, Graph
graph) {
        var newResult = new ShortestPathDTO(result);

        var intersections = newResult.getPath().stream()
                .filter(vertex ->
graph.getTransitionsByVertexLabel(vertex).size() > 2)
                .collect(Collectors.toList());

        var seed = Double.valueOf(Math.random() *
intersections.size()).intValue();
        var chosenIntersection = intersections.get(seed);
        var chosenIndex = newResult.getPath().indexOf(chosenIntersection);
        var bannedTransition =
newResult.getPath().get(newResult.getPath().indexOf(chosenIntersection) + 1);

        var success = retryPathFromTo(
                newResult,
                graph,
                chosenIntersection,
                chosenIndex,
                bannedTransition
        );

        if (success) {
            return newResult;
```

```
        } else {
            return result;
        }
    }

    private boolean retryPathFromTo(ShortestPathDTO path, Graph graph, String
chosenIntersection, Integer chosenIndex, String bannedTransition) {
        for (int i = chosenIndex; i < path.getPath().size() - 1; i++) {
            var current = path.getPath().get(i);
            var next = path.getPath().get(i + 1);
            var costOfTransition =
graph.getTransitionsByVertexLabel(current).stream()
                    .filter(x -> x.getDestination().equals(next))
                    .findFirst()
                    .get()
                    .getWeight();
            path.addCost(-costOfTransition);
        }

        path.getPath().subList(chosenIndex + 1, path.getPath().size() -
1).clear();

        var visitedMap = new HashMap<String, Integer>();
        for (Vertex vertex : graph.getAllVertices()) {
            if (path.getPath().contains(vertex.getLabel()) &&
!vertex.getLabel().equals(graph.getExitVertexLabel())) {
                visitedMap.put(vertex.getLabel(), 0);
            } else {
                visitedMap.put(vertex.getLabel(), -1);
            }
        }
        visitedMap.put(bannedTransition, 0);
        visitedMap.put(graph.getEntranceVertexLabel(), 0);

        var cursor = chosenIntersection;
        while (!cursor.equals(graph.getExitVertexLabel())) {
            var possibleTransitions =
graph.getTransitionsByVertexLabel(cursor).stream()
                    .filter(transition ->
visitedMap.get(transition.getDestination()) < 0)
                    .collect(Collectors.toList());

            if (possibleTransitions.size() < 1) {
                var resetTo = cursor;
                var found = false;
                var pathIterator = path.getPath().size() - 2;

                while (!found) {
                    if (pathIterator < 0) {
                        return false;
                    }

                    resetTo = path.getPath().get(pathIterator);
                    var possiblePreviousTransitions =
graph.getTransitionsByVertexLabel(resetTo).stream()
                            .filter(transition ->
visitedMap.get(transition.getDestination()) < 0)
                            .collect(Collectors.toList());
                    if (possiblePreviousTransitions.size() > 0) {
                        found = true;
                    }
                    pathIterator--;
                }
```

```
            var prevCursor = path.getPath().get(path.getPath().size() - 3);
            var backtickCursor = path.getPath().get(path.getPath().size() - 2);
                while (!backtickCursor.equals(resetTo)) {
                    var finalBacktickCursor = backtickCursor;

                    var costOfTransition =
graph.getTransitionsByVertexLabel(prevCursor).stream()
                            .filter(x ->
x.getDestination().equals(finalBacktickCursor))
                            .findFirst()
                            .get()
                            .getWeight();
                    path.addCost(-costOfTransition);
                    path.getPath().remove(backtickCursor);

                  prevCursor = path.getPath().get(path.getPath().size() - 3);
                    backtickCursor = path.getPath().get(path.getPath().size()
- 2);
                }
                cursor = resetTo;
            } else {
                var seed = 0;
                if (possibleTransitions.size() > 1) {
                    seed = Double.valueOf(Math.random() *
possibleTransitions.size()).intValue();
                }

                var chosenTransition = possibleTransitions.get(seed);

                cursor = chosenTransition.getDestination();
                visitedMap.put(cursor, visitedMap.get(cursor) + 1);

                if (!cursor.equals(graph.getExitVertexLabel())) {
                    path.getPath().add(path.getPath().size() - 1,
chosenTransition.getDestination());
                }
                path.addCost(chosenTransition.getWeight());
            }
        }

        return true;
    }

    private ShortestPathDTO generateInitialResult(Graph graph) {
        var solution = new ShortestPathDTO();
        solution.setPath(new ArrayList<>());
        solution.setCost(0);

        var visitedMap = new HashMap<String, Integer>();
        for (Vertex vertex : graph.getAllVertices()) {
            visitedMap.put(vertex.getLabel(), -1);
        }
        visitedMap.put(graph.getEntranceVertexLabel(), 0);

        var cursor = graph.getEntranceVertexLabel();
        solution.getPath().add(graph.getEntranceVertexLabel());

        while (!cursor.equals(graph.getExitVertexLabel())) {
            var possibleTransitions =
graph.getTransitionsByVertexLabel(cursor).stream()
                    .filter(transition ->
visitedMap.get(transition.getDestination()) < 0)
                    .collect(Collectors.toList());
```

```
            if (possibleTransitions.size() < 1) {
                var resetTo = cursor;
                var found = false;
                var pathIterator = solution.getPath().size() - 2;

                while (!found) {
                    resetTo = solution.getPath().get(pathIterator);
                    var possiblePreviousTransitions =
graph.getTransitionsByVertexLabel(resetTo).stream()
                            .filter(transition ->
visitedMap.get(transition.getDestination()) < 0)
                            .collect(Collectors.toList());

                    if (possiblePreviousTransitions.size() > 0) {
                        found = true;
                    }
                    pathIterator--;
                }

                var prevCursor =
solution.getPath().get(solution.getPath().size() - 2);
                var backtickCursor =
solution.getPath().get(solution.getPath().size() - 1);

                while (!backtickCursor.equals(resetTo)) {
                    var finalBacktickCursor = backtickCursor;
                    var costOfTransition =
graph.getTransitionsByVertexLabel(prevCursor).stream()
                            .filter(x ->
x.getDestination().equals(finalBacktickCursor))
                            .findFirst()
                            .get()
                            .getWeight();
                    solution.addCost(-costOfTransition);
                    solution.getPath().remove(backtickCursor);

                    prevCursor =
solution.getPath().get(solution.getPath().size() - 2);
                    backtickCursor =
solution.getPath().get(solution.getPath().size() - 1);
                }
                cursor = resetTo;
            } else {
                var seed = 0;
                if (possibleTransitions.size() > 1) {
                    seed = Double.valueOf(Math.random() *
possibleTransitions.size()).intValue();
                }

                var chosenTransition = possibleTransitions.get(seed);

                cursor = chosenTransition.getDestination();
                visitedMap.put(cursor, visitedMap.get(cursor) + 1);

                solution.getPath().add(chosenTransition.getDestination());
                solution.addCost(chosenTransition.getWeight());
            }
        }

        return solution;
    }
}
```

.2

.2.1

```
package parser;

import lombok.SneakyThrows;

import java.io.File;
import java.util.Scanner;

public class InputMazeParser {

    @SneakyThrows
    public char[][] parseNotWeightedNotOrientedMaze() {
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\_maze_large"));

        var scanner = new Scanner(new
File(".\\src\\main\\resources\\straight_line_maze"));
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\corner_maze"));
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\open_field_maze"));
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\net_trap_maze"));
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\snowflake_maze"));
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\pillar_room_maze"));

        var size = scanner.nextLine().split("\\s");

        var readArray = new
char[Integer.parseInt(size[0])][Integer.parseInt(size[1])*3];

        for (int i = 0; i < Integer.parseInt(size[0]); i++) {
            readArray[i] = scanner.nextLine().toCharArray();
        }

        var array = new
char[Integer.parseInt(size[0])][Integer.parseInt(size[1])];

        for (int i = 0; i < Integer.parseInt(size[0]); i++) {
            for (int j = 0; j < Integer.parseInt(size[1]); j++) {
                array[i][j] = readArray[i][j*3];
            }
        }

        return array;
    }

    @SneakyThrows
    public char[][] parseWeightedNotOrientedMaze() {
        var scanner = new Scanner(new
File(".\\src\\main\\resources\\straight_line_maze_w"));
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\corner_maze_w"));
//        var scanner = new Scanner(new
```

```
File(".\\src\\main\\resources\\open_field_maze_w"));
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\net_trap_maze_w"));
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\snowflake_maze_w"));
//        var scanner = new Scanner(new
File(".\\src\\main\\resources\\pillar_room_maze_w"));
        var size = scanner.nextLine().split("\\s");

        var readArray = new char[Integer.parseInt(size[0])*2 -
1][Integer.parseInt(size[1])*6];

        for (int i = 0; i < Integer.parseInt(size[0])*2 - 1; i++) {
            readArray[i] = scanner.nextLine().toCharArray();
        }

        var array = new char[Integer.parseInt(size[0])*2 -
1][Integer.parseInt(size[1])*2];

        for (int i = 0; i < Integer.parseInt(size[0])*2 - 1; i++) {
            for (int j = 0; j < Integer.parseInt(size[1])*2 - 1; j++) {
                array[i][j] = readArray[i][j*3];
            }
        }

        return array;
    }
}
```

.2.2

```
package converters;

import graph.Graph;

import java.util.function.Function;

public class ArrayToGraphConverter implements Function<char[][], Graph> {

    @Override
    public Graph apply(char[][] array) {
        var graph = new Graph();

        for (int i = 0; i < array.length; i++) {
            for (int j = 0; j < array[i].length; j++) {
                var symbol = array[i][j];

                if (symbol != ’ ’) {
                    if (symbol == ’ ’ || symbol == ’A’ || symbol == ’B’) {
                        var label = String.format("%d_%d", i, j);
                        if (symbol == ’A’) {
                            graph.setEntranceVertexLabel(label);
                        } else if (symbol == ’B’) {
                            graph.setExitVertexLabel(label);
                        }
                        graph.addNewVertex(label);
                        searchForAdjacentVertices(graph, array, i, j);
                    }
                }
            }
```

```
        }

        return graph;
    }

    private void searchForAdjacentVertices(Graph graph, char[][] array, int
rowIndex, int columnIndex) {
        if (rowIndex > 0
                && rowIndex < array.length - 1
                && (array[rowIndex + 1][columnIndex] == 'A'
                || array[rowIndex + 1][columnIndex] == 'B'
                || array[rowIndex + 1][columnIndex] == ' ')) {

            graph.addNewVertex(String.format("%d_%d", rowIndex + 1,
columnIndex));
            graph.addTwoWayTransition(
                    String.format("%d_%d", rowIndex, columnIndex),
                    String.format("%d_%d", rowIndex + 1, columnIndex),
                    1,
                    1
            );
        }

        if (rowIndex < array.length - 1
                && rowIndex > 0
                && (array[rowIndex - 1][columnIndex] == 'A'
                || array[rowIndex - 1][columnIndex] == 'B'
                || array[rowIndex - 1][columnIndex] == ' ')) {

            graph.addNewVertex(String.format("%d_%d", rowIndex - 1,
columnIndex));
            graph.addTwoWayTransition(
                    String.format("%d_%d", rowIndex, columnIndex),
                    String.format("%d_%d", rowIndex - 1, columnIndex),
                    1,
                    1
            );
        }

        if (columnIndex > 0
                && columnIndex < array[rowIndex].length - 1
                && (array[rowIndex][columnIndex + 1] == 'A'
                || array[rowIndex][columnIndex + 1] == 'B'
                || array[rowIndex][columnIndex + 1] == ' ')) {

            graph.addNewVertex(String.format("%d_%d", rowIndex, columnIndex +
1));
            graph.addTwoWayTransition(
                    String.format("%d_%d", rowIndex, columnIndex),
                    String.format("%d_%d", rowIndex, columnIndex + 1),
                    1,
                    1
            );
        }

        if (columnIndex < array[rowIndex].length - 1
                && columnIndex > 0
                && (array[rowIndex][columnIndex - 1] == 'A'
                || array[rowIndex][columnIndex - 1] == 'B'
                || array[rowIndex][columnIndex - 1] == ' ')) {

            graph.addNewVertex(String.format("%d_%d", rowIndex, columnIndex -
1));
            graph.addTwoWayTransition(
```

```
                String.format("%d_%d", rowIndex, columnIndex),
                String.format("%d_%d", rowIndex, columnIndex - 1),
                1,
                1
            );
        }
    }
}

package converters;

import graph.Graph;

import java.util.function.Function;

public class ArrayToWeightedGraphConverter implements Function<char[][],
Graph> {

    @Override
    public Graph apply(char[][] array) {
        var graph = new Graph();

        for (int i = 0; i < array.length; i++) {
            for (int j = 0; j < array[i].length; j++) {
                var symbol = array[i][j];

                if (symbol != ' ') {
                    if (symbol == ' ' || symbol == 'A' || symbol == 'B') {
                        var label = String.format("%d_%d", i/2, j/2);
                        if (symbol == 'A') {
                            graph.setEntranceVertexLabel(label);
                        } else if (symbol == 'B') {
                            graph.setExitVertexLabel(label);
                        }
                        graph.addNewVertex(label);
                        searchForAdjacentVertices(graph, array, i, j);
                    }
                }
            }
        }

        return graph;
    }

    private void searchForAdjacentVertices(Graph graph, char[][] array, int
rowIndex, int columnIndex) {
        if (rowIndex > 1
                && rowIndex < array.length - 2
                && (array[rowIndex + 2][columnIndex] == 'A'
                || array[rowIndex + 2][columnIndex] == 'B'
                || array[rowIndex + 2][columnIndex] == ' ')) {

            graph.addNewVertex(String.format("%d_%d", rowIndex/2 + 1,
columnIndex/2));
            graph.addTwoWayTransition(
                    String.format("%d_%d", rowIndex/2, columnIndex/2),
                    String.format("%d_%d", rowIndex/2 + 1, columnIndex/2),
                    Character.getNumericValue(array[rowIndex +
1][columnIndex]),
                    Character.getNumericValue(array[rowIndex +
1][columnIndex])
            );
        }
```

```
        if (rowIndex < array.length - 2
                && rowIndex > 1
                && (array[rowIndex - 2][columnIndex] == 'A'
                || array[rowIndex - 2][columnIndex] == 'B'
                || array[rowIndex - 2][columnIndex] == ' ')) {

            graph.addNewVertex(String.format("%d_%d", rowIndex/2 - 1,
columnIndex/2));
            graph.addTwoWayTransition(
                    String.format("%d_%d", rowIndex/2, columnIndex/2),
                    String.format("%d_%d", rowIndex/2 - 1, columnIndex/2),
                    Character.getNumericValue(array[rowIndex -
1][columnIndex]),
                    Character.getNumericValue(array[rowIndex -
1][columnIndex])
            );
        }

        if (columnIndex > 1
                && columnIndex < array[rowIndex].length - 2
                && (array[rowIndex][columnIndex + 2] == 'A'
                || array[rowIndex][columnIndex + 2] == 'B'
                || array[rowIndex][columnIndex + 2] == ' ')) {

            graph.addNewVertex(String.format("%d_%d", rowIndex/2,
columnIndex/2 + 1));
            graph.addTwoWayTransition(
                    String.format("%d_%d", rowIndex/2, columnIndex/2),
                    String.format("%d_%d", rowIndex/2, columnIndex/2 + 1),
                    Character.getNumericValue(array[rowIndex][columnIndex +
1]),
                    Character.getNumericValue(array[rowIndex][columnIndex +
1])
            );
        }

        if (columnIndex < array[rowIndex].length - 2
                && columnIndex > 1
                && (array[rowIndex][columnIndex - 2] == 'A'
                || array[rowIndex][columnIndex - 2] == 'B'
                || array[rowIndex][columnIndex - 2] == ' ')) {

            graph.addNewVertex(String.format("%d_%d", rowIndex/2,
columnIndex/2 - 1));
            graph.addTwoWayTransition(
                    String.format("%d_%d", rowIndex/2, columnIndex/2),
                    String.format("%d_%d", rowIndex/2, columnIndex/2 - 1),
                 Character.getNumericValue(array[rowIndex][columnIndex - 1]),
                 Character.getNumericValue(array[rowIndex][columnIndex - 1])
            );
        }
    }
}
```