ДОДАТОК А

Графічний матеріал атестаційної роботи

Міністерство освіти та науки України

Харківський національний університет радіоелектроніки
Кафедра «Електронних обчислювальних машин»

Атестаційна робота
на тему:
«Алгоритми проходження лабіринту в умовах, що змінюються»

Виконав: ст. гр. СПм-19-1 Павленко О. С.
Керівник: доц. Іващенко Г.С.

Актуальність проблеми

У сучасній ігровій індустрії поширені ігри з відкритим світом, який наповнений неігровими персонажами, програмування переміщення яких потребує врахування можливостей змін ігрового світу під час пересування ним. Якщо ігровий світ нестатичний, тобто навколо персонажу щось відбувається, то це має впливати на дії персонажу. Тим паче повинно відображатися на пошуку шляху, який повинен бути динамічно змінний.

Пошук шляху у без урахування особливостей, які можуть виникнути під час проходження, можна реалізувати за допомогою рішення задачі пошуку найкоротшого шляху. Але враховуючи особливості побудови маршруту, які характерні під час проходження у режимі реального часу, стає актуальною задача пошуку найкоротшого шляху у лабіринтах у змінних умовах.

Аналіз існуючих рішень

Ігрові світи які можуть бути описані за допомогою теорії графів, що дозволяє використовувати для вирішення завдань побудови маршрутів типові алгоритми пошуку найкоротшого шляху у графах. У розробці ігор часто застосовують алгоритм A*, що є модифікацією алгоритму Дейкстри.

Одним із таких рішень є компонент Navmesh у платформі для розробки ігор Unity 3D.

У теперішній час існує багато досліджень з приводу вирішення задачі пошуку найкоротшого шляху у розробці ігрових програмних застосунків.

Постановка задачі

Метою роботи є дослідження ефективності та якості роботи алгоритмів пошуку найкоротшого шляху у змінних умовах, для застосування їх як імітацію інтелекту у неігрових персонажів в розробці ігрових програмних застосунків.

Для тестування алгоритмів пошуку найкоротшого шляху у змінних умовах потрібно створити ігровий світ, який може бути представлений у вигляді лабіринту, який, в свою чергу, описується за допомогою теорії графів.

Тестове програмне забезпечення повинно враховувати можливість зміни умов під час проходження маршруту у режимі реального часу.

При виборі інтерпретації змінних умов, що аналізуються у даній роботі, серед розглянутих варіантів обрана саме необхідність враховувати при побудові маршруту можливості зміни позиції кінцевої точки маршруту під час його проходження.

Потрібно розробити застосунок, який допоможе у тестуванні алгоритмів пошуку найкоротшого шляху у змінних умовах.

Використані технології

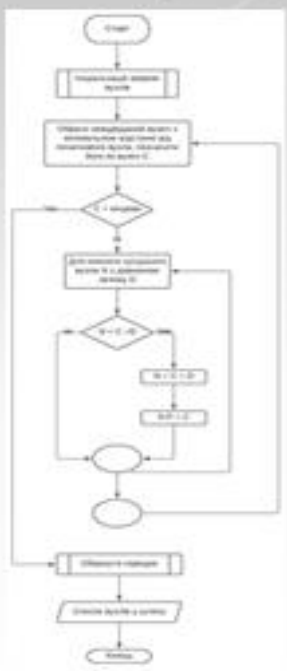
5

- Unity 3D;
- Visual Studio 2019;
- NavMesh.

Алгоритм Дейкстри

6

Алгоритм Дейкстри названий на честь голландського вченого Едсгер Дейкстри (Edsger Dijkstra). Алгоритм був запропонований в 1959 році для знаходження найкоротших шляхів від однієї вершини до всіх інших в орієнтованому зваженому графі, за умови, що всі ребра в графі мають невід'ємні ваги. Алгоритм Дейкстри відноситься до так званих «жадібних» алгоритмів.



Алгоритм A*

7

Алгоритм A * – один з найпопулярніших методів вирішення завдань на пошук найкоротшого маршруту. Його відносять до проінформованих алгоритмів пошуку, так як для вирішення завдань використовуються дані про вартість шляху і принципи евристики.

У 1964 році Нільс Нільсон винайшов евристичний підхід до збільшення швидкості алгоритму Дейкстри. Цей алгоритм був названий A1. У 1967 році Бертрам Рафаель зробив значні поліпшення до цього алгоритму, але йому не вдалося досягти оптимальності. Він назвав цей алгоритм A2. Тоді в 1968 році Пітер Е. Харт представив аргументи, які доводили, що A2 був оптимальним при використанні послідовної евристики лише з незначними змінами. У його доказ алгоритму також включений розділ, який показував, що новий алгоритм A2 був можливо найкращим алгоритмом, враховуючи умови. Таким чином, він позначив новий алгоритм в синтаксисі зірочкою, оскільки він починається на A і включав в себе всі можливі номери версій.

Генетичний алгоритм

8

Генетичний алгоритм (ГА) – це алгоритм, який дозволяє виявити задовільне рішення до аналітично нерозв'язних проблем через послідовний підбір і комбінування шуканих параметрів з використанням механізмів, що схожі на процес біологічної еволюції. Генетичний алгоритм є різновидом еволюційних обчислень. Природний відбір – основний механізм еволюції. Суть відбору полягає в тому, що особини, які більш пристосовані, мають більше можливостей для виживання і розмноження і, отже, приносять більше потомства, ніж погано пристосовані особини. При цьому, завдяки передачі генетичної інформації (генетичному спадкуванню), нащадки успадковують від батьків основні їх якості.

Результати досліджень

9

Результати налаштування генетичного алгоритму

Генетичні алгоритми мають багато параметрів для налаштувань. Через це, перед початком дослідження та порівняльного аналізу роботи алгоритмів пошуку найкоротшого шляху у різних умовах, потрібно налагодити роботу генетичного алгоритму, для знаходження саме таких параметрів, що забезпечують можливість використання для вирішення задання та ефективну роботу алгоритму.

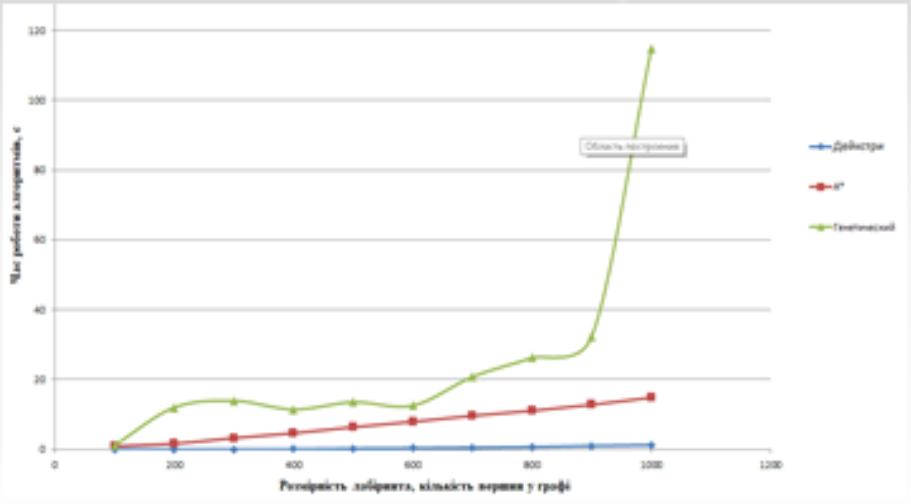
У даній роботі розроблене тестове програмне забезпечення надає змогу налагоджувати та аналізувати вплив наступних параметрів роботи генетичного алгоритму:

- розмір популяції;
- швидкість зміни популяції, яка налагоджується через встановлення
- вірогідності кросоверінгу для тих особин популяції, що приймають участь у схрещуванні;
- вірогідність мутації, при цьому ступінь впливу мутації на хромосому не корегується;
- довжину хромосоми.

Результати досліджень у статичних умовах

10

Для аналізу було обрано лабіринти, які генерувалися випадковим чином, що описуються графами розмірністю від 100 вершин до 1000. Експерименти проводилися на графах з кроком у 100 вершин.



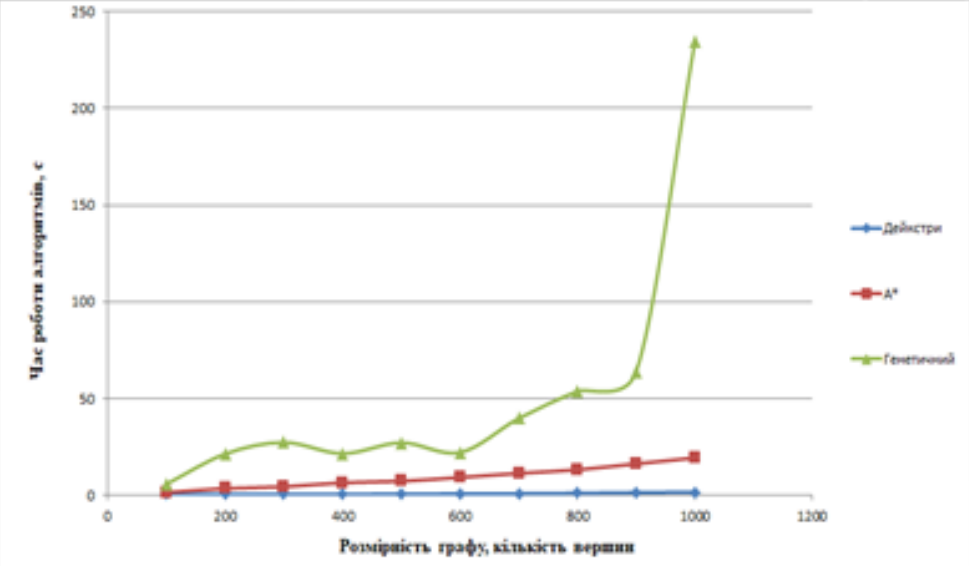
Результати досліджень у змінних умовах

11

Змінні умови реалізовані у вигляді зміни кінцевої точки маршруту, через те, що така реалізація часто зустрічається у розробці ігор. Зміна кінцевої точки реалізована у програмному коді, а не шляхом встановлення користувачем. Зміна позиції відбувається кожні 0,2 секунди. Кількість змін позиції користувач може налаштувати. Дослідження проводилися з однією зміною позиції. Позиції для зміни в усіх тестових лабіринтах однакові. Кількість секунд, при яких відбувається зміна кінцевої точки, незалежна. Це дозволяє досліджувати алгоритми, не втручаючись у їх швидкість роботи, у статичних умовах. Тобто при зміні кінцевої точки маршруту останні вершини у маршрутах обходу, що побудовані за допомогою різних алгоритмів можуть бути на різній відстані від попередньої цілі у один момент часу. Через це, агенти, що керуються алгоритми з нешвидким пошуком можуть продовжувати рухатись до іншої цілі, майже не міняючи траєкторію. В свою чергу, більш швидкодіючі алгоритми можуть опинитися у ситуації, коли потрібно повертатися назад.

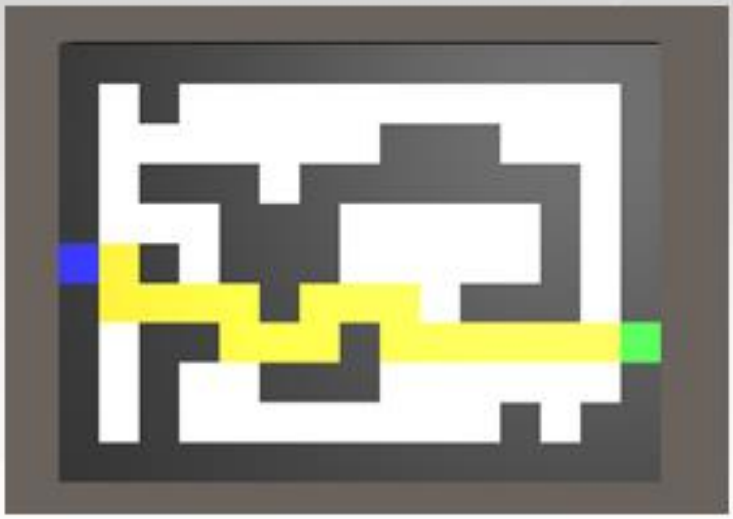
Результати досліджень у змінних умовах

12



Візуалізація лабіринту невеликої розмірності

13



Висновки

14

В даній роботі досліджено ефективність алгоритмів пошуку найкоротшого шляху у змінних умовах у лабіринтах, що описуються графами різної розмірності, для застосування при розробці ігрового штучного інтелекту неігрових персонажів. Розроблено застосунок для тестування алгоритмів пошуку найкоротшого шляху на платформі розробки ігор Unity 3D з використанням компоненту Navmesh. У дослідженні розглядалися такі алгоритми пошуку найкоротшого шляху, як алгоритм Дейкстри, алгоритм A* та генетичний алгоритм.

Практичне значення роботи полягає в розробці елементів ігрового штучного інтелекту для неігрових персонажів, що забезпечує пересування по ігровим світам з урахуванням зміни умов у реальному часі, через це поведінка персонажів виглядає більш природньою для гравця.

ДОДАТОК Б
ВИХІДНИЙ КОД

Б.1 Реалізація ігрового світу

Б.1.1 Ігровий світ для тестування алгоритмів Дейкстри та A*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GridGenerator : MonoBehaviour
{
    [SerializeField] private int rowInGrid;
    [SerializeField] private int columnInGrid;
    [SerializeField] private float padding;
    [SerializeField] private Transform nodePrefab;
    [SerializeField] private DijkstraPathfinder dijkstraPathfinder;
    [SerializeField] private GameObject pathNode;

    public List<Transform> grid = new List<Transform>();

    public Transform startNode;
    public Transform endNode;

    public int startNodeIndex;
    public int endNodeIndex;


    void Start()
    {
        this.GenerateGrid();
        this.GenerateNeighbours();
    }

    void Update()
    {

        if (startNode != null)
        {
            this.SetColor(startNode, Color.green);
        }
        if (endNode != null)
        {
            this.SetColor(endNode, Color.blue);
        }

        if (dijkstraPathfinder.IsDone == false)
        {
            List<Transform>                    paths                    =
dijkstraPathfinder.StartDijkstra(StartNode, EndNode);

            foreach (Transform path in paths)
            {
```

```
                    Renderer rend = path.GetComponent<Renderer>();
                    rend.material.color = Color.yellow;

                }
            }
        }

        private void GenerateGrid()
        {
            int counter = 0;
            for (int i = 0; i < rowInGrid; i++)
            {
                for (int j = 0; j < columnInGrid; j++)
                {
                    Transform node = Instantiate(nodePrefab, new Vector3((j*
padding)  +  gameObject.transform.position.x,  gameObject.transform.position.y,
(i * padding) + gameObject.transform.position.z), Quaternion.identity);
                    node.name = "node (" + counter + ")";
                    grid.Add(node);
                    counter++;
                }
            }

            startNode = grid[startNodeIndex];
            endNode = grid[endNodeIndex];

            if (startNode != null)
            {
                this.SetColor(startNode, Color.green);
            }
            if (endNode != null)
            {
                this.SetColor(endNode, Color.blue);
            }
        }

        void SetColor(Transform node, Color color)
        {
            Renderer rend = node.GetComponent<Renderer>();
            rend.material.color = color;
        }

        private void GenerateNeighbours()
        {
            for (int i = 0; i < grid.Count; i++)
            {
                Node currentNode = grid[i].GetComponent<Node>();
                int index = i + 1;

                if (index % columnInGrid == 1)
                {
                    if (i + columnInGrid < columnInGrid * rowInGrid)
                    {
                        currentNode.AddNeighbourNode(grid[i+columnInGrid]);
                    }

                    if (i - columnInGrid >= 0)
                    {
                        currentNode.AddNeighbourNode(grid[i-columnInGrid]);
                    }
                    currentNode.AddNeighbourNode(grid[i + 1]);
                }
```

```
            else if (index % columnInGrid == 0)
            {
                if (i + columnInGrid < columnInGrid * rowInGrid)
                {
                    currentNode.AddNeighbourNode(grid[i+columnInGrid]);
                }

                if (i - columnInGrid >= 0)
                {
                    currentNode.AddNeighbourNode(grid[i-columnInGrid]);
                }
                currentNode.AddNeighbourNode(grid[i - 1]);
            }

            else
            {
                if (i + columnInGrid < columnInGrid * rowInGrid)
                {
                    currentNode.AddNeighbourNode(grid[i+columnInGrid]);
                }

                if (i - columnInGrid >= 0)
                {
                    currentNode.AddNeighbourNode(grid[i-columnInGrid]);
                }
                currentNode.AddNeighbourNode(grid[i + 1]);
                currentNode.AddNeighbourNode(grid[i - 1]);
            }

        }
    }

}
```

Б.1.2 Ігровий світ для тестування генетичного алгоритму

```
using System;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MazeController : MonoBehaviour {
public int[,] map;
public Vector2 startPosition;
public Vector2 endPosition;
public GameObject wallPrefab;
public GameObject exitPrefab;
public GameObject startPrefab;
public GameObject pathPrefab;
public GameObject nodePrefab;
public GeneticAlgorithm geneticAlgorithm;
public List<int> fittestDirections;
public List<GameObject> pathTiles;

private bool isReached = false;
private int pathTilesLengthBeforeCheck;
private bool isCheckStarted = false;
public GameObject PrefabByTile(int tile) {
      if (tile == 1) return wallPrefab;
```

```
            if (tile == 5) return startPrefab;
            if (tile == 8) return exitPrefab;
            if (tile == 0) return nodePrefab;
            return null;
      }

      public Vector2 Move(Vector2 position, int direction) {
            switch (direction) {
            case 0: // North
                  if (position.y - 1 < 0 || map [(int)(position.y - 1),
(int)position.x] == 1) {
                        break;
                  } else {
                        position.y -= 1;
                  }
                  break;
            case 1: // South
                  if (position.y + 1 >= map.GetLength (0) || map
[(int)(position.y + 1), (int)position.x] == 1) {
                        break;
                  } else {
                        position.y += 1;
                  }
                  break;
            case 2: // East
                  if (position.x + 1 >= map.GetLength (1) || map
[(int)position.y, (int)(position.x + 1)] == 1) {
                        break;
                  } else {
                        position.x += 1;
                  }
                  break;
            case 3: // West
                  if (position.x - 1 < 0 || map [(int)position.y,
(int)(position.x - 1)] == 1) {
                        break;
                  } else {
                        position.x -= 1;
                  }
                  break;
            }
            return position;
      }

      public double TestRoute(List<int> directions) {
            Vector2 position = startPosition;

            for (int directionIndex = 0; directionIndex < directions.Count;
directionIndex++) {
                  int nextDirection = directions [directionIndex];
                  position = Move (position, nextDirection);
            }

            Vector2 deltaPosition = new Vector2(
                  Math.Abs(position.x - endPosition.x),
                  Math.Abs(position.y - endPosition.y));
            double result = 1 / (double)(deltaPosition.x+ deltaPosition.y+1);
            if (result == 1)
            return result;
      }

      public void Populate() {
            Debug.Log ("length(0)=" + map.GetLength(0));
            Debug.Log ("length(1)=" + map.GetLength(1));
```

```
            for (int y = 0; y < map.GetLength(0); y++) {
                  for (int x = 0; x < map.GetLength(1); x++) {
                        GameObject prefab = PrefabByTile (map [y, x]);
                        if (prefab != null) {
                              GameObject wall = Instantiate (prefab);
                              wall.transform.position = new Vector3 (x,0,-y);
                        }
                  }
            }
      }

      void Start () {
            map = new int[,] {
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
        {1,0,1,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,1,1,1,0,0,0,1},
        {1,0,1,1,1,0,1,1,1,1,1,1,1,0,1},
        {1,0,0,0,1,1,1,0,0,0,0,0,1,0,1},
        {8,0,1,0,1,1,1,0,0,0,0,0,1,0,1},
        {1,0,0,0,0,1,0,0,0,0,1,1,1,0,1},
        {1,0,1,1,0,0,0,1,0,0,0,0,0,0,5},
        {1,0,1,0,0,1,1,1,0,0,0,0,0,0,1},
        {1,0,1,0,0,0,0,0,0,0,0,1,0,1,1},
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
        };

            Populate();
            fittestDirections = new List<int> ();
            pathTiles = new List<GameObject> ();

            geneticAlgorithm = new GeneticAlgorithm ();
            geneticAlgorithm.mazeController = this;
            geneticAlgorithm.Run ();
      }

      public void ClearPathTiles() {
            foreach (GameObject pathTile in pathTiles) {
                  Destroy(pathTile);
            }
            pathTiles.Clear();
      }

      public void RenderFittestChromosomePath() {
            ClearPathTiles ();
            Genome                        fittestGenome                        =
geneticAlgorithm.genomes[geneticAlgorithm.fittestGenome];
            List<int>     fittestDirections     =     geneticAlgorithm.Decode
(fittestGenome.bits);
            Vector2 position = startPosition;

            foreach (int direction in fittestDirections) {
                  position = Move (position, direction);
                  GameObject pathTile = Instantiate (pathPrefab);
                  pathTile.transform.position = new Vector3(position.x, 0, -
position.y);
                  pathTiles.Add (pathTile);
            }
      }

      void Update () {
            if(!isCheckStarted)
            {
                  StartCoroutine(CheckOnStopGeneticWork());
```

```
        }

        if(!isReached)
        {
            RenderFittestChromosomePath();
        }
        if (geneticAlgorithm.busy) geneticAlgorithm.Epoch();
}


private IEnumerator CheckOnStopGeneticWork()
{
        isCheckStarted = true;

        Debug.Log("work");
        yield return new WaitForSeconds(1f);

        if(pathTiles.Count() != this.pathTilesLengthBeforeCheck)
        {
            pathTilesLengthBeforeCheck = pathTiles.Count();
        } else
        {
            isReached = true;
        }

        isCheckStarted = false;
}

public void StartAlgorithm()
{
        ClearPathTiles();
        StopAllCoroutines();
        isReached = false;
        pathTiles = new List<GameObject>();
        this.pathTilesLengthBeforeCheck = 0;

        isCheckStarted = false;

        Populate();
        fittestDirections = new List<int>();

        geneticAlgorithm = new GeneticAlgorithm();
        geneticAlgorithm.mazeController = this;
        geneticAlgorithm.Run();
}
}
```

Б.2 Реалізація алгоритмів пошуку найкоротшого шляху

Б.2.1 Реалізація алгоритму Дейкстри

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DijkstraPathfinder : MonoBehaviour
{
    private bool isDone = false;
```

```
    public bool IsDone
    {
        get { return isDone; }
        set { isDone = value; }
    }

    private GameObject[] nodes;

    Node currentNode;

    public List<Transform> StartDijkstra(Transform start, Transform end)
    {
        nodes = GameObject.FindGameObjectsWithTag("Node");

        List<Transform> result = new List<Transform>();
        Transform node = DijkstrasAlgo(start, end);

        while (node != null)
        {
            result.Add(node);
            Node currentNode = node.GetComponent<Node>();
            node = currentNode.GetParentNode();
        }

        result.Reverse();
        return result;
    }

    private Transform DijkstrasAlgo(Transform start, Transform end)
    {
        double startTime = Time.realtimeSinceStartup;

        List<Transform> unexplored = new List<Transform>();

        foreach (GameObject obj in nodes)
        {
            Node n = obj.GetComponent<Node>();
            if (n.IsWalkable())
            {
                n.ResetNode();
                unexplored.Add(obj.transform);
            }
        }

        Node startNode = start.GetComponent<Node>();
        startNode.SetWeight(0);

        while (unexplored.Count > 0)
        {
            unexplored.Sort((x,y)=>x.GetComponent<Node>().GetWeight()
.CompareTo(y.GetComponent<Node>().GetWeight()));

            Transform current = unexplored[0];
            unexplored.Remove(current);

            currentNode = current.GetComponent<Node>();
            List<Transform> neighbours = currentNode.GetNeighbourNode();

            foreach (Transform neighNode in neighbours)
            {
                Node node = neighNode.GetComponent<Node>();

                if (unexplored.Contains(neighNode) && node.IsWalkable())
```

```
                    {
                        float                   distance                    =
Vector3.Distance(neighNode.position, current.position);
                        distance = currentNode.GetWeight() + distance;

                        if (distance < node.GetWeight())
                        {
                            node.SetWeight(distance);
                            node.SetParentNode(current);
                        }
                    }
                }
            }
            double endTime = (Time.realtimeSinceStartup - startTime);
            Debug.Log("Compute time: " + endTime);
            isDone = true;
            Debug.Log("Path completed!");
            return end;

        }
    }
```

Б.2.2 Реалізація алгоритму A*

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System;

public class Pathfinding : MonoBehaviour {

PathRequestManager requestManager;
Grid grid;
public GameObject node;
void Awake() {
      requestManager = GetComponent<PathRequestManager>();
      grid = GetComponent<Grid>();
}
public void StartFindPath(Vector3 startPos, Vector3 targetPos) {
      StartCoroutine(FindPath(startPos,targetPos));
}
IEnumerator FindPath(Vector3 startPos, Vector3 targetPos) {

      Vector3[] waypoints = new Vector3[0];
      bool pathSuccess = false;

      Node startNode = grid.NodeFromWorldPoint(startPos);
      Node targetNode = grid.NodeFromWorldPoint(targetPos);


      if (startNode.walkable && targetNode.walkable) {
            Heap<Node> openSet = new Heap<Node>(grid.MaxSize);
            HashSet<Node> closedSet = new HashSet<Node>();
            openSet.Add(startNode);

            while (openSet.Count > 0) {
                  Node currentNode = openSet.RemoveFirst();
                  closedSet.Add(currentNode);
                  if (currentNode == targetNode) {
                        pathSuccess = true;
```

```
                              break;
                        }

                        foreach (Node neighbour in
grid.GetNeighbours(currentNode)) {
                              if(!neighbour.walkable ||
closedSet.Contains(neighbour)) {
                                    continue;
                              }
                              int newMovementCostToNeighbour =
currentNode.gCost + GetDistance(currentNode, neighbour);
                              if (newMovementCostToNeighbour <
neighbour.gCost || !openSet.Contains(neighbour)) {
                                    neighbour.gCost =
newMovementCostToNeighbour;
                                    neighbour.hCost = GetDistance(neighbour,
targetNode);
                                    neighbour.parent = currentNode;

                                    if (!openSet.Contains(neighbour))
                                          openSet.Add(neighbour);
                              }
                        }
                  }
            }
            yield return null;
            if (pathSuccess) {
                  waypoints = RetracePath(startNode,targetNode);
            }
            requestManager.FinishedProcessingPath(waypoints,pathSuccess);

      }

      Vector3[] RetracePath(Node startNode, Node endNode) {
            List<Node> path = new List<Node>();
            Node currentNode = endNode;
            while (currentNode != startNode) {
                  path.Add(currentNode);
                  currentNode = currentNode.parent;
            }
            Vector3[] waypoints = SimplifyPath(path);
            Array.Reverse(waypoints);
            return waypoints;

      }

      Vector3[] SimplifyPath(List<Node> path) {
            List<Vector3> waypoints = new List<Vector3>();
            Vector2 directionOld = Vector2.zero;
            for (int i = 1; i < path.Count; i ++) {
                  Vector2 directionNew = new Vector2(path[i-1].gridX -
path[i].gridX,path[i-1].gridY - path[i].gridY);
                  if (directionNew != directionOld) {
                        waypoints.Add(path[i].worldPosition);
                  node=MonoBehaviour.Instantiate(Resources.Load("Current",
typeof(GameObject)))as GameObject;
                        node.transform.position = path[i].worldPosition;
                  }
                  directionOld = directionNew;
            }
            return waypoints.ToArray();
      }
      int GetDistance(Node nodeA, Node nodeB) {
            int dstX = Mathf.Abs(nodeA.gridX - nodeB.gridX);
```

```
            int dstY = Mathf.Abs(nodeA.gridY - nodeB.gridY);
            if (dstX > dstY)
                  return 14*dstY + 10* (dstX-dstY);
            return 14*dstX + 10 * (dstY-dstX);
      }}
```

Б.2.3 Реалізація генетичного алгоритму

```
      using System;
      using System.Linq;
      using System.Collections;
      using System.Collections.Generic;
      using UnityEngine;

      public class GeneticAlgorithm {
      public List<Genome> genomes;
      public List<Genome> lastGenerationGenomes;

      public int populationSize = 600;
      public double crossoverRate = 0.4f;
      public double mutationRate = 1f;
      public int chromosomeLength = 40;
      public int geneLength = 2;
      public int fittestGenome;
      public double bestFitnessScore;
      public double totalFitnessScore;
      public int generation;
      public MazeController mazeController;
      public bool busy;
      double startTime;
      System.Random random = new System.Random(0);
      public GeneticAlgorithm() {
            busy = false;
            genomes = new List<Genome> ();
            lastGenerationGenomes = new List<Genome> ();
      }
      public void Mutate(List<int> bits) {
            for (int i = 0; i < bits.Count; i++) {
                  if (random.NextDouble() < mutationRate) {
                        bits [i] = bits [i] == 0 ? 1 : 0;
                  }
            }
      }
      public void Crossover(List<int> mom, List<int> dad, List<int> baby1,
List<int> baby2) {
            if (random.NextDouble() > crossoverRate || mom == dad) {
                  baby1.AddRange(mom);
                  baby2.AddRange(dad);
                  return;
            }
            int crossoverPoint = random.Next (0, chromosomeLength - 1);
            for (int i = 0; i < crossoverPoint; i++) {
                  baby1.Add (mom [i]);
                  baby2.Add (dad [i]);
            }
            for (int i = crossoverPoint; i < mom.Count; i++) {
                  baby1.Add (dad [i]);
                  baby2.Add (mom [i]);
            }
      }
```

```
	public Genome RouletteWheelSelection() {
		double slice = random.NextDouble() * totalFitnessScore;
		double total = 0;
		int selectedGenome = 0;

		for (int i = 0; i < populationSize; i++) {
			total += genomes [i].fitness;

			if (total > slice) {
				selectedGenome = i;
				break;
			}
		}
		return genomes[selectedGenome];
	}

	public void UpdateFitnessScores() {
		fittestGenome = 0;
		bestFitnessScore = 0;
		totalFitnessScore = 0;

		for (int i = 0; i < populationSize; i++) {
			List<int> directions = Decode (genomes [i].bits);
			genomes [i].fitness = mazeController.TestRoute
(directions);

			totalFitnessScore += genomes [i].fitness;

			if (genomes [i].fitness > bestFitnessScore) {
				bestFitnessScore = genomes [i].fitness;
				fittestGenome = i;

				if (genomes [i].fitness == 1) {
					busy = false;
					Debug.Log(lastGenerationGenomes.Count);
					double endTime = (Time.realtimeSinceStartup -
startTime);
					Debug.Log(endTime);
					return;
				}
			}
		}
	}

	public List<int> Decode(List<int> bits) {
		List<int> directions = new List<int> ();

		for (int geneIndex = 0; geneIndex < bits.Count; geneIndex +=
geneLength) {
			List<int> gene = new List<int> ();

			for (int bitIndex = 0; bitIndex < geneLength; bitIndex++) {
				gene.Add (bits [geneIndex + bitIndex]);
			}

			directions.Add (GeneToInt (gene));
		}
		return directions;
	}

	public int GeneToInt(List<int> gene) {
		int value = 0;
		int multiplier = 1;
```

```
        for (int i = gene.Count; i > 0; i--) {
                value += gene [i - 1] * multiplier;
                multiplier *= 2;
        }
        return value;
}

public void CreateStartPopulation() {
        genomes.Clear ();

        for (int i = 0; i < populationSize; i++) {
                Genome baby = new Genome (chromosomeLength);
                genomes.Add (baby);
        }
}

public void Run() {
        startTime = Time.realtimeSinceStartup;
        CreateStartPopulation ();
        busy = true;
}

public void Epoch() {
        if (!busy) return;
        UpdateFitnessScores ();

        if (!busy) {
                lastGenerationGenomes.Clear();
                lastGenerationGenomes.AddRange (genomes);
                return;
        }

        int numberOfNewBabies = 0;

        List<Genome> babies = new List<Genome> ();
        while (numberOfNewBabies < populationSize) {

                Genome mom = RouletteWheelSelection ();
                Genome dad = RouletteWheelSelection ();
                Genome baby1 = new Genome();
                Genome baby2 = new Genome();
                Crossover (mom.bits, dad.bits, baby1.bits, baby2.bits);
                Mutate (baby1.bits);
                Mutate (baby2.bits);
                babies.Add (baby1);
                babies.Add (baby2);

                numberOfNewBabies += 2;
        }

        lastGenerationGenomes.Clear();
        lastGenerationGenomes.AddRange (genomes);
        genomes = babies;
        generation++;
}
}
```