ДОДАТОК А

Слайди презентації

Дослідження оптимізаційних моделей та методів з метою створення розвинутого інтелекту для комп'ютерних ігор в жанрі "Tower Defense"

ВИКОНАВ: СТ. ГР.ІПЗм-18-2 ТОПЧІЙ О.Д.

КЕРІВНИК: ДОЦЕНТ МАЗУРОВА О.О.

Аналіз проблемної області

Дослідники поділяють ігри з урахуванням штучного інтелекту (ШІ) на три категорії:

- Ігри **з базовим ШІ** є найменьш вимогливими до обчислюваних ресурсів комп'ютера, проте з точки зору ігрового досвіду гравця вони не є дуже інтерактивними та ніяк не реагують на дії гравця.
- Ігри з **високореактивним (або середнім) ШІ** потребують достатньо обчислюваних ресурсів, проте вони задовільняють варіативний ігровий досвід, та адаптуються до дій ігрока у процесі геймплею.
- Ігри із **розвиненим (високорозвиненим) ШІ** потребують найбільше обчислюваних потужностей, але й в процесі ігри вони використовують більш прораховані, оптимальні стратегії, навчаються та приспособлюються під стиль гравця

Постановка задачі

- розробити механіку та принципи гри в жанрі "Tower Defence";
- провести аналіз оптимізаційних моделей та методів з метою створення розвинутого штучного інтелекту ;
- розробити математичні моделі поведінки ШІ у грі в жанрі Tower Defense на базі оптимізаційних моделей та методів ;
- розробити алгоритми вирішення оптимізаційних задач на базі розроблених моделей;
- виконати програмну реалізацію ігрової системи;
- спланувати та провести експериментальне дослідження розроблених алгоритмів та моделей;
- розробити рекомендації стосовно використання обраного підходу для створення розвинутого ШІ в іграх-стратегіяі у жанрі Tower Defense.

3

Механіка та принципи гри в жанрі “Tower Defense”:

1. Грають 2 гравця через інтернет. Один гравець грає за вежі (захищається), другий гравець за монстрів (атакує);
2. Битва відбувається на ігровому полі розбитому на клітини. На поле є дороги, по яких ходять монстри і земля, на якій будуються вежі;
3. Мета атакуючого гравця - зруйнувати за відведений час замок противника;
4. Мета гравця, що захищається - не дати атакуючим зруйнувати замок за відведений час. Вежі атакують і знищують монстрів;
5. Бій триває 30 хвилин. Ігрове поле генерується кожен раз.
6. Настільний додаток для ПК
7. Естетика – середньовічне фентезі

4

Аналіз оптимізаційних моделей та методів

- прийняття рішень в умовах визначеності:
  - на базі теорії корисності:
    - **лінійні згортки:**
      - лінійна адитивна згортка;
      - **лінійна адитивна згортка з ваговими коефіцієнтами;**
    - нелінійні згортки:
  - на базі теорії дослідження операцій:
    - задачі математичного програмування:
      - **Задачі лінійного програмування:**
        - задачі про суміші;
        - **задачі про розподіл ресурсів;**
        - **задачі про призначення;**
        - транспортні задачі та інше;
      - задачі динамічного програмування ;
      - інші;
- прийняття рішень в умовах невизначеності та ризику.

5

Модель задачі призначення юнітів на шляхи на основі лінійної адитивної згортки

Задача вибору ігрової сутності та його призначення на ігровому полі промодельована за допомогою моделі лінійної адитивної згортки:

$$Z = \sum_{j=1}^{n} \alpha_j \beta_j K_j(P_i), \quad (1)$$

де α_j- нормуючі множники,

β_j- вагові коефіцієнти, що залежать від відносного вкладу всіх критеріїв в загальний критерій,

$K_j(P_i)$– критеріальна оцінки за критеріїм K_j для варіанту P_i.

6

Критерії до задачі призначення юнітів на шляхи

- K_1 – загальна кількість захисних веж на шляху (ваговий коефіцієнт β_1=0,1);

- K_2 – безпека шляху: кількість одиниць шкоди, які потенційно можуть завдати нападнику вежі на шляху (β_2=0,5);

- K_3 – потенційна кількість ресурсів, які може отримати нападник, пройшовши по шляху або його частині (β_3=0,3);

- K_4 – відношення захисних веж на шляху з особливими ефектами при атаці до загальної кількості веж на шляху (β_4=0,1).

7

Критерії до задачі розміщення веж

- K_1 – загальна кількість захисних веж на шляхах, що можуть бути атаковані вежею з позиції (ваговий коефіцієнт β_1=0,3);

- K_2 – кількість клітин на ігровому полі, що належить до множини шляхів та може бути атакована вежею з позиції (β_2=0,3);

- K_3 – близкість до середини ігрового поля (β_3=0,2);

- K_4 – загальна кількість нападників що в момент обчислення знаходяться на ігровому полі, та можуть бути потенційно атаковані вежею вежею з позиції (β_4=0,2).

8

Введені позначення для мат. моделей

$\{U_i\}_{i=1}^{m}$ – множина юнітів Ui, що доступні для запуску на поле в грі, всього доступно m юнітів;

$P = \{P_i\}_{i=1}^{n}$ – множина шляхів Pi на ігровому полі;

$T = \{T_i\}_{i=1}^{k}$ – список усіх веж, що доступні для запуску на поле в грі, всього доступно m веж;

P*= $\{P^*_{(x,y)}\}_{x=1,y=1}^{X,Y}$ – множина клітин на ігровому полі XxY;

X – ширина ігрового поля;

Y – довжина ігрового поля;

C_i – ціна i-го юніту або вежі у внутрішніх ресурсах;

Q_i – сила атаки юніта i (шкода замку супротивника);

H_i – кількість одиниць життя юніту i;

V_i – сила атаки вежі i (шкода здоров'ю юніта);

M_M – кількість ресурсів гравця за монстрів;

M_T – кількість ресурсів гравця за вежі.

9

Табличне надання оптимізаційної задачі про призначення юнітів на шляхи (ЗЛП)

Шляхи / Юніти	P_1	P_2	...	P_n	Обмеження за юнітами U_i
U_1 (C_1)	D_{11}, X_{11}	D_{12}, X_{12}	...	D_{1n}, X_{1n}	1
U_2 (C_2)	D_{21}, X_{21}	D_{22}, X_{22}	...	D_{2n}, X_{2n}	1
...	...	...	...	...	
U_m (C_m)	D_{m1}, X_{m1}	D_{m2}, X_{m2}	...	D_{mn}, X_{mn}	1
Обмеження за шляхами P_j	1	1		1	

10

Модель оптимізаційної задачі про призначення юнітів на шляхи (ЗЛП)

Dij – потенційний урон, що може бути завданий замку супротивника, у випадку, якщо юніт *i* буде запущений на шлях *j*.

Потенційний урон Dij розраховується за формулою:

$$D_{ij} = Q_i L_{ij} + \sum_{u=0}^{U} Q_u L_{uij}$$

де Q_i – сила атаки юніта i, L_{ij} – факт того, дійде юніт і до кінця ігрового полю по шляху j, чи ні, може приймати значення 0 чи 1:

$$L_{ij} = \begin{cases} 0, H_i > T_{ij} \\ 1, H_i \leq T_{ij} \end{cases}$$

де H_i – кількість одиниць життя юніту *i*, T_{ij} – кількість урону, що буде завдано юніту *i* на шляху *j*.

12

Модель оптимізаційної задачі про призначення юнітів на шляхи (ЗЛП)

Цільова функція повинна забезпечувати, щоб юніти призначались на такі шляхи, які максимально збільшують потенційний збиток замку супротивника (що еквівалентно мінімальній шкоді гравця-захисника).

$$F = \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{X_{ij}(Q_i L_{ij} + \sum_{u=0}^{U} Q_u L_{uij})}{C_i} \rightarrow max,$$

$$\sum_{j=1}^{n} x_{ij} \leq 1, \forall_i = 1, m;$$

$$\sum_{i=1}^{m} x_{ij} = 1, \forall_j = 1, n;$$

$$X_{ij} = 1|0, \forall i = 1, m, \forall j = 1, n.$$

11

Табличне надання оптимізаційної задачі про призначення веж на позиції (ЗЛП)

Вежі / Клітини	$P_{(1,1)}$	...	$P_{(x,y)}$	Обмеження на вежі T_i
T_1 (C_1)	$D_{1(1,1)}, A_{1(1,1)}, K_{1(1,1)}, B_{1(1,1)}$	...	$D_{1(x,y)}, A_{1(x,y)}, K_{1(x,y)}, B_{1(x,y)}$	1
IT_2 (C_2)	$D_{2(1,1)}, A_{2(1,1)}, K_{2(1,1)}, B_{2(1,1)}$	...	$D_{2(x,y)}, A_{2(x,y)}, K_{2(x,y)}, B_{2(x,y)}$	1
...	...	...	...	
T_m (C_m)	$D_{m(1,1)}, A_{m(1,1)}, K_{m(1,1)}, B_{m(1,1)}$	...	$D_{m(x,y)}, A_{m(x,y)}, K_{m(x,y)}, B_{m(x,y)}$	1
Обмеження на клітини $P^*_{(x,y)}$	1		1	

13

Модель оптимізаційної задачі про призначення веж на позиції (ЗЛП)

Цільова функція повинна забезпечувати, щоб вежі розміщувались у таких позиціях, які максимально збільшують потенційний збиток юнітам супротивника.

$$F = \sum_{i=1}^{m} \sum_{x=1}^{X} \sum_{y=1}^{Y} \frac{(A_{i(x,y)}+1)(K_{i(x,y)}+1)D_{i(x,y)}X_{i(x,y)}}{C_i} \to max,$$

$$\sum_{x=1}^{X} \sum_{y=1}^{Y} b_{ixy} \le 1, \forall_x = 1, X, \forall_y = 1, Y,$$

$$\sum_{i=1}^{m} b_{ixy} = 1, \forall_j = 1, n,$$

$$B_{ij} = 1|0, \forall i = 1, m, \forall_x = 1, X, \forall_y = 1, Y.$$

14

Модель оптимізаційної задачі про призначення веж на позиції (ЗЛП)

$A_i(x, y)$ – урон, що може бути завданий юнітам супротивника, які у даний момент знаходяться на полі, якщо вежа і буде розташована у позиції (x,y).

$$A_{i(x,y)} = \sum_{u=0}^{U}(Z_{iu}F_{ui(x,y)} + \sum_{t=0}^{T} Z_{tu}F_{tui(x,y)}),$$

де U – множина юнітів, що зараз є на полі; T - множина веж, що зараз є на полі; Ziu - кількість урону, що отримає юніт u від вежі t; $Fui(x,y)$ – кількість разів, коли юніт u атакований вежою i, якщо вежа і буде розташована у клітині (x,y); $Ftui(x,y)$ - кількість разів, коли юніт u атакований вежою t, чи ні, якщо вежа і буде розташована у клітині (x,y) (може приймати значення 0 чи 1).

$K_i(x, y)$ – кількість юнітів, з тих, що у даний момент знаходяться на ігровому полі та що будуть знищені, якщо розмістити і-ю вежою у позиції (x,y).

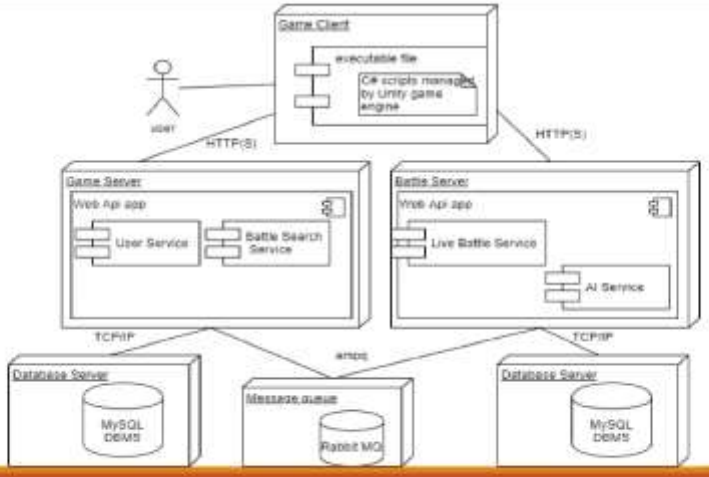
$$K_{i(x,y)} = \sum_{u=0}^{U} L_{ui(x,y)},$$

де $L_{ui(x,y)}$ - факт того, дійде юніт u до кінця ігрового полю, чи ні, якщо вежа і буде розташована у позиції (x,y). Може приймати значення 0 чи 1:

$L_{uij} = \begin{cases} 0, H_u > T_{uij} \\ 1, H_u \leq T_{uij} \end{cases}$, where H_u – кількість одиниць життя юніту u; T_{uij} – кількість урону, що буде завдано юніту u, якщо вежа і буде розташована у позиції *(x,y)*.

$D_i(x, y)$ – потенційний урон, що може бути завданий по навколишнім клітинам і-ю вежою у позиції *(x,y)*.

Архітектура системи (діаграма розгортання)



Геймплей

17

Планування експериментів

- базовий ШІ в серії експериментів не розглядався; його експериментальне дослідження було проведено окремо та показало, що він не є дуже інтерактивними та не дає необхідного рівня перемог над супротивником-людиною;
- в якості високореактивного (або середнього) ШІ розглядалася поведінка ШІ на базі розробленої багатокритеріальної моделі вибору;
- в якості розвинутого ШІ розглядалася поведінка ШІ на базі розроблених моделей оптимізаційних задач про призначення.

18

Методи, що досліджувались

Для юнітів:

- призначення юнітів на шляхи шляхом багатокритеріального вибору на базі лінійної адитивної згортки (А31);
- призначення юнітів на шляхи (ЗЛП) на основі жадібного алгоритму (ЖА1);
- призначення юнітів на шляхи (ЗЛП) на основі симплекс-методу (БА1).

Для веж:

- призначення веж на клітини шляхом багатокритеріального вибору на базі лінійної адитивної згортки (А32);
- призначення веж на клітини (ЗЛП) на основі жадібного алгоритму (ЖА2);
- призначення веж на шляхи (ЗЛП) на основі симплекс-методу (БА2).

19

Експерименти проводилися на іграх різною розмірності

- (Іграl): кількість юнітів - 5, розмір поля 10X10, кількість веж 3, кількість шляхів – 2;
- (Іграl): кількість юнітів - 8, розмір поля 20X20, кількість веж 5, кількість шляхів – 5;
- (Іграl): кількість юнітів - 10, розмір поля 50X50, кількість веж 7, кількість шляхів – 10.

Для кожної серії експериментів було проведено по 100 ігрових сесій.

20

Результати серії експериментів для призначення юнітів

Ігра 1

Алгоритм/ Показники	Середній час одної роботи алгоритму, мс	Середнє використання оперативної пам'яті, Кб	Процент перемог юнітів, %
АЗ1	9	312	44
БА1	82	1016	48
ЖА1	51	956	48

Ігра 2

Алгоритм/ Показники	Середній час одної роботи алгоритму, мс	Середнє використання оперативної пам'яті, Кб	Процент перемог юнітів, %
АЗ1	32	425	42
БА1	196	1598	47
ЖА1	117	1056	46

Ігра 3

Алгоритм/ Показники	Середній час одної роботи алгоритму, мс	Середнє використання оперативної пам'яті, Кб	Процент перемог юнітів, %
АЗ1	183	531	39
БА1	567	1900	46
ЖА1	479	1576	45

21

Результати серії експериментів для призначення веж

Ігра 1

Алгоритм/ Показники	Середній час одної роботи алгоритму, мс	Середнє використання оперативної пам'яті, Кб	Процент перемог веж, %
БА2	801	7011	52
ЖА2	591	5678	53
А32	54	988	47

Ігра 2

Алгоритм/ Показники	Середній час одної роботи алгоритму, мс	Середнє використання оперативної пам'яті, Кб	Процент перемог веж, %
БА2	2570	8329	55
ЖА2	1961	6571	55
А32	870	1492	43

Ігра 3

Алгоритм/ Показники	Середній час одної роботи алгоритму, мс	Середнє використання оперативної пам'яті, Кб	Процент перемог веж, %
БА2	4132	9221	56
ЖА2	3006	7221	55
А32	1081	1912	41

22

Рекомендації на основі експериментів

1. багатокритеріальна модель поведінки ШІ на основі методів згорток дає значну економію обчислювальних ресурсів для ігор будь-якого розміру, але поступається іншим моделям за показниками перемог і, загалом, не дозволяє перейти 50-відсотковий поріг. Такі показники дозволяють рекомендувати модель для впровадження середнього інтелекту, але не рекомендується для реалізації розвинутого ШІ, тобто для вірня «Експерт»;
2. не лінійна залежність зростання обчислювальних ресурсів від розмірності ігри, між тим, дозволяє рекомендувати модель поведінки ШІ на базі багатокритеріальної згортки (алгоритми А31 та А32) для ігрових полів надвеликих розмірів (експерименти для цих типів полів не проводились); така модель надає задовільний рівень ефективності рішень та має значно кращу обчислювальну швидкість;
3. модель поведінки ШІ з використанням базового алгоритму (БА1 та БА2), взагалі, не рекомендовано для реалізації, оскільки майже на такому ж рівні перемог (+1%), як і при використання жадібного алгоритму (ЖА1 та ЖА2), вона отримує значно гірші обчислівальні показники;
4. найкращим варіантом для реалізації розвинутого інтелекту (для ігрових полів середніх та великих розмірів, як «Ігра2» та «Ігра3») є модель поведінки ШІ з використанням жадібного алгоритму (ЖА1 та ЖА2), адже вони надають гарний рівень перемог, та мають пристойну обчислювальну швидкість.

23

Висновки

- розроблено механіку та принципи гри в жанрі "Tower Defence";
- проведено аналіз опитізаційних моделей та методів з метою створення штучного інтелекту;
- розроблено математичні моделі поведінки ШІ у грі в жанрі Tower Defense на базі оптимізаційних задач теорії прийняття рішень:
  - багатокритеріальна згорткова модель для поведінки ШІ під час призначення нападників (монстрів) на ігрові шляхи та захисних веж на клітини ігрового поля;
  - оптимізаційна модель задачі про призначення для поведінки ШІ з боку нападника та захисника : під час запуску монстра на ігрове поле по певному шляху, під час розташування захисних веж на шляхах карти;
- виконано програмну реалізацію ігрової системи;
- розроблено алгоритми вирішення оптимізаційних задач на базі розроблених моделей;
- були спланувані експериментальні дослідження розроблених методів та моделей;
- за результатами проведених експериментів були розроблені рекомендації стосовно використання обраного підходу для створення розвинутого ШІ в іграх у жанрі Tower Defense;
- була подана стаття до науково-технічного журналу «Радіоелектроніка, інформатика, управління» що індексується наукометричною базою Web of Science (Core Collection ESCI);
- були опубліковані тези доповіді на молодіжному форумі «Радіоелекторніка та молодь у XXI ст».

24

ДОДАТОК Б

Лістинг коду

```
public class PathOptimisation : IPathOptimiser
{
  private readonly IStatsLibrary _statsLib;
  private readonly AdditiveConvolutionCalculator _additiveConvolutionCalculator;
  private readonly GameCalculator _gameCalculator;

  private static readonly double[] OptimalPathCriteriaWeights =
  {
    0.1, // first criteria - number of towers on the path
    0.5, // second criteria - total damage of towers on the path for the monster
    0.3, // third criteria - remoteness of towers from beginning (length of the possible path)
    0.1, // fourth criteria - ratio of towers with special coef
  };

  public PathOptimisation(IStatsLibrary statsLibrary)
  {
    _gameCalculator = new GameCalculator(statsLibrary);
    _additiveConvolutionCalculator = new
AdditiveConvolutionCalculator(OptimalPathCriteriaWeights);
    _statsLib = statsLibrary;
  }

  public int GetFastestPath(Path[] path, Unit unit)
  {
    var pos = unit.Position;
    var possible = GetPossiblePath(path, pos).ToArray();

    int[] coefs = new int[possible.Length];

    for (int i = 0; i < possible.Length; i++)
    {
      coefs[i] = possible[i].Length - possible[i].PointOnThePathPosition(pos);
    }

    return Array.IndexOf(coefs, coefs.Min());
  }

  public int GetOptimalPath(Field field, Unit unit){
    var paths = field.StaticData.Path;
    var tableToAnalize = new double[paths.Length,
_additiveConvolutionCalculator.NumberOfCriterias];
    for (int i = 0; i < paths.Length; i++){
      var towersOnPath = FindTowersThatCanAttackPath(field, i);
      var towers = towersOnPath.Select(t => field[t]).ToArray();
      var towersTypes = towers.Select(t => t.Type).ToArray();
      tableToAnalize[i, 0] = 1 - (double)towersOnPath.Count / field.State.Towers.Count;
      tableToAnalize[i, 1] = GetTotalAttackDamage(towersTypes, unit);
      tableToAnalize[i, 2] = GetAvgTowersRemoteness(paths[i], towers);
      tableToAnalize[i, 3] = GetTowersWithSpecialEffectRate(towersTypes);
    }
    return _additiveConvolutionCalculator.FindOptimalVariantIndex(tableToAnalize);
  }

  private double GetTotalAttackDamage(IEnumerable<GameObjectType> towerTypes, Unit unit){
    return 1 - (double)towerTypes.Sum(t => _gameCalculator.CalculateDamage(unit.Type, t)) /
unit.Health;
  }
  private HashSet<int> FindTowersThatCanAttackPath(Field field, int pathId){
    var path = field.StaticData.Path[pathId];
```

```
    var towers = new HashSet<int>();
    foreach (var point in path)
      towers.UnionWith(field.FindTowersThatCanAttack(point, _statsLib));
    return towers;
  }

  private double GetTowersWithSpecialEffectRate(ICollection<GameObjectType> towerTypes)
  {
    return 1 - (double)towerTypes.Count(t => _statsLib.GetTowerStats(t).SpecialEffects != null) /
towerTypes.Count;
  }

  private double GetAvgTowersRemoteness(Path path, ICollection<GameObject> towers)
  {
    double remotenessSum = 0;
    foreach (var tower in towers)
    {
      var pos = path.First(point => _gameCalculator.IsTowerCanAttack(tower, point));
      remotenessSum += (double)path.PointOnThePathPosition(pos) / path.Length;
    }

    return remotenessSum / towers.Count;
  }

  private static IEnumerable<Path> GetPossiblePath(IEnumerable<Path> path, Point position)
  {
    return path.Where(p => p.PointOnThePathPosition(position) != -1);
  }
}

public class StateCalculator
{
  private readonly IStatsLibrary _statsLib;
  private readonly SpecialEffectLogicFactory _effectLogicFactory;
  private readonly Field _field;
  private readonly MoneyProvider _moneyProvider;
  private readonly GameCalculator _gameCalculator;

  private const int NotFound = -1;
  private static readonly Point NotFoundPoint = new Point(-1, -1);

  public Field Field => _field;

  public StateCalculator(IStatsLibrary statsLibrary, Field fieldState)
  {
    _statsLib = statsLibrary;
    _field = (Field)fieldState.Clone();

    _effectLogicFactory = new SpecialEffectLogicFactory();
    _moneyProvider = new MoneyProvider(statsLibrary);
    _gameCalculator = new GameCalculator(statsLibrary);
  }

  public void SetState(FieldState fieldState)
  {
    _field.SetState(fieldState);
  }

  public GameTick[] CalculateActionsByTicks()
  {
    var ticks = new List<List<GameAction>>(40);
    while (_field.State.Castle.Health > 0
      && _field.State.Units.Any())
    {
      var actions = new List<GameAction>();

      actions.AddRange(GetUnitActions());
      actions.AddRange(GetTowerActions());
```

```
      ticks.Add(actions);
    }
    if (_field.State.Castle.Health <= 0)
    {
      ticks.Add(new List<GameAction>
      {
        new GameAction{ActionId = ActionId.MonsterPlayerWins}
      });
    }

    var result = new GameTick[ticks.Count];
    for (int i = 0; i < ticks.Count; i++)
    {
      result[i] = new GameTick
      {
        RelativeTime = i,
        Actions = ticks[i]
      };
    }
    return result;
  }

  private List<GameAction> GetUnitActions()
  {
    var actions = new List<GameAction>();
    var unitsToRemove = new List<int>();

    foreach (var unit in _field.State.Units)
    {
      if (unit.Health <= 0)
      {
        unitsToRemove.Add(unit.GameId);
        continue;
      }
      if (unit.WaitTicks != 0)
      {
        unit.WaitTicks -= 1;
        continue;
      }
      if (unit.Effect != null && unit.Effect.Effect != EffectId.None)
      {
        unit.Effect.Duration -= 1;
        if (unit.Effect.Duration == 0)
        {
          unit.Effect = SpecialEffect.Empty;
          actions.Add(new GameAction {ActionId = ActionId.UnitEffectCanseled, UnitId =
unit.GameId});
        }
      }

      var path = _field.StaticData.Path[unit.PathId.Value];
      var stats = _statsLib.GetUnitStats(unit.Type);
      if (path.End == unit.Position)
      {
        var attackAction = new GameAction
        {
          ActionId = ActionId.UnitAttacksCastle,
          UnitId = unit.GameId,
          Damage = stats.Damage
        };
        actions.Add(attackAction);
        _field.State.Castle.Health -= stats.Damage;
        unitsToRemove.Add(unit.GameId);

        var unitReward = _moneyProvider.GetUnitReward(_field, attackAction);
        _field.State.MonsterMoney += unitReward;
        actions.Add(new GameAction{ActionId = ActionId.MonsterPlayerRecievesMoney, Money =
unitReward});
      }
```

```
      else
      {
        ApplyUnitEffects(unit, actions);
        var contextSpeedCoeff = GetContextSpeedCoeff(unit);
        var nextPoint = path.GetNext(unit.Position);
        unit.Position = nextPoint;
        unit.WaitTicks = stats.Speed * contextSpeedCoeff;
        actions.Add(new GameAction
        {
          ActionId = ActionId.UnitMoves,
          Position = nextPoint,
          UnitId = unit.GameId,
          WaitTicks = stats.Speed * contextSpeedCoeff
        });
      }
    }

    _field.RemoveMany(unitsToRemove);

    return actions;
  }

  private List<GameAction> GetTowerActions()
  {
    var actions = new List<GameAction>();

    foreach (var tower in _field.State.Towers)
    {
      if (tower.WaitTicks != 0)
      {
        tower.WaitTicks -= 1;
        continue;
      }

      var stats = _statsLib.GetTowerStats(tower.Type);

      switch (stats.Attack)
      {
        case TowerStats.AttackType.Usual:
        case TowerStats.AttackType.Magic:
          var targetId = FindTarget(_field, tower, stats);
          if (targetId != NotFound)
          {
            var unit = (Unit) _field[targetId];
            var damage = _gameCalculator.CalculateDamage(unit.Type, stats);
            tower.WaitTicks = stats.AttackSpeed;

            actions.Add(new GameAction
            {
              ActionId = ActionId.TowerAttacks,
              TowerId = tower.GameId,
              UnitId = targetId,
              WaitTicks = stats.AttackSpeed
            });
            actions.Add(new GameAction
            {
              ActionId = ActionId.UnitRecievesDamage,
              UnitId = targetId,
              Damage = damage
            });

            ApplyTowerEffects(stats, unit, actions);
            unit.Health -= damage;
            if (unit.Health <= 0)
            {
              var unitTrue = _field.State.Units.First(u => u.GameId == targetId);
              var dieAction = new GameAction {ActionId = ActionId.UnitDies, UnitId = targetId,
TowerId = tower.GameId, Position = unitTrue.Position};
              var killAction = new GameAction {ActionId = ActionId.TowerKills, TowerId =
```

```
tower.GameId, UnitId = targetId, Position = unitTrue.Position};

                actions.Add(dieAction);
                actions.Add(killAction);

                var towerReward = _moneyProvider.GetTowerReward(_field, dieAction);
                var unitReward = _moneyProvider.GetUnitReward(_field, killAction);

                _field.State.MonsterMoney += unitReward;
                _field.State.TowerMoney += towerReward;

                actions.Add(new GameAction{ActionId = ActionId.TowerPlayerRecievesMoney, Money =
towerReward});
                actions.Add(new GameAction{ActionId = ActionId.MonsterPlayerRecievesMoney, Money =
unitReward});

                _field.RemoveGameObject(targetId);
              }
            }
            break;

          case TowerStats.AttackType.Burst:
            var targetPoint = FindBurstTarget(_field, tower, stats);
            if (targetPoint != NotFoundPoint)
            {
              tower.WaitTicks = stats.AttackSpeed;

              actions.Add(new GameAction
              {
                ActionId = ActionId.TowerAttacksPosition,
                TowerId = tower.GameId,
                Position = targetPoint,
                Damage = stats.Damage,
                WaitTicks = stats.AttackSpeed
              });
              var units = _field.FindUnitsAt(targetPoint);
              var deadUnits = new List<int>();
              foreach (var unit in units)
              {
                ApplyTowerEffects(stats, unit, actions);
                var damage =  _gameCalculator.CalculateDamage(unit.Type, stats);

                if(damage == 0)
                  continue;

                actions.Add(new GameAction
                {
                  ActionId = ActionId.UnitRecievesDamage,
                  UnitId = unit.GameId,
                  Damage = damage
                });
                unit.Health -= damage;
                if (unit.Health <= 0)
                {
                  var dieAction = new GameAction {ActionId = ActionId.UnitDies, UnitId =
unit.GameId, TowerId = tower.GameId, Position = unit.Position};
                  var killAction = new GameAction {ActionId = ActionId.TowerKills, TowerId =
tower.GameId, UnitId = unit.GameId, Position = unit.Position};

                  actions.Add(dieAction);
                  actions.Add(killAction);

                  var towerReward = _moneyProvider.GetTowerReward(_field, dieAction);
                  var unitReward = _moneyProvider.GetUnitReward(_field, killAction);

                  _field.State.MonsterMoney += unitReward;
                  _field.State.TowerMoney += towerReward;

                  actions.Add(new GameAction{ActionId = ActionId.TowerPlayerRecievesMoney, Money =
```

```
towerReward});
                actions.Add(new GameAction{ActionId = ActionId.MonsterPlayerRecievesMoney, Money
= unitReward});

                deadUnits.Add(unit.GameId);
              }
            }
            _field.RemoveMany(deadUnits);
          }
          break;
      }
    }

    return actions;
  }

public class LiveBattleService : ILiveBattleService, IBattleInitializationService
{
  private readonly ConcurrentDictionary<Guid, int> _battles;

  private readonly IBattleRepository _battleRepository;
  private readonly IUserRepository _userRepository;
  private readonly IProvider<LiveBattleModel> _provider;
  private readonly IStateChangeRecalculator _recalculator;
  private readonly IFieldFactory _fieldFactory;

public async Task RecalculateAsync(StateChangeCommand command, int curTick)
{
  using (new BattleLocker(command.BattleId))
  {
    var fieldSerialized = _provider.Find(command.BattleId);
    var fieldState = fieldSerialized.State;
    var ticks = fieldSerialized.Ticks.Take(curTick);

    await Task.Run(() => ResolveActions(fieldState, ticks, _statsLibrary));

    if (command.UnitCreationOptions != null)
    {
      foreach (var opt in command.UnitCreationOptions)
      {
        _recalculator.AddNewUnit(fieldState, opt.Type, _mapper.Map<CreationOptions>(opt));
      }
    }
    if (command.TowerCreationOptions != null)
    {
      foreach (var opt in command.TowerCreationOptions)
      {
        _recalculator.AddNewTower(fieldState, opt.Type, _mapper.Map<CreationOptions>(opt));
      }
    }

    var calc = new StateCalculator(_statsLibrary, fieldState);
    fieldSerialized.Ticks = await Task.Run(() => calc.CalculateActionsByTicks());
    _provider.Update(fieldSerialized);

    IncrementBattleVersion(command.BattleId);
  }
  GC.Collect();
}

public async Task TryEndBattleAsync(Guid battleId, Guid userId)
{
  Guid? left = null;
  using (new BattleLocker(battleId))
  {
    var battle = _provider.Find(battleId);
    var entity = await _battleRepository.FindAsync(battleId);
    var expCalc = new UserExperienceCalculator();
```

```
    PlayerSide winSide;
    if (battle.State.StaticData.EndTimeUtc > DateTime.UtcNow)
    {
      winSide = entity.Monsters_UserId == userId ? PlayerSide.Towers : PlayerSide.Monsters;
      left = userId;
    }
    else
    {
      if (entity.WinnerId != Guid.Empty)
      {
        return;
      }
      ResolveActions(battle.State, battle.Ticks, _statsLibrary);
      winSide = battle.State.State.Castle.Health > 0 ? PlayerSide.Towers : PlayerSide.Monsters;
    }

    entity.WinnerId = winSide == PlayerSide.Monsters ? entity.Monsters_UserId :
entity.Towers_UserId;
    entity.EndTime = DateTime.UtcNow;

    await Task.WhenAll(_battleRepository.UpdateAsync(entity),
      _userRepository.IncrementExperienceAsync(entity.Monsters_UserId,
expCalc.CalcUserExp(entity, entity.Monsters_UserId, left)),
      _userRepository.IncrementExperienceAsync(entity.Towers_UserId, expCalc.CalcUserExp(entity,
entity.Towers_UserId, left)));

    battle.Ticks = CreateBattleEndTick(winSide);

    IncrementBattleVersion(battleId);
    _provider.Update(battle);
    DisposeBattleAsync(battleId);
  }
}


public class FieldManager : MonoBehaviour
{
   public const float TickSecond = 0.4f;
   private readonly Quaternion Quaternion =Q uaternion.Euler(90, 90, 270);

   private Coroutine _resolver;
   private int _revision;
   private Guid _battleId;
   private string _session;
   private ObjectPool _pool;
   private int _tickCount;
   private GameProcessNetworkWorker _gameProcessNetworkWorker;

   public GameObjectType Selected { get; set; }
   public PlayerSide Winner { get; set; }
   public PlayerSide Side { get; set; }
   public Field Field { get; private set; }
   public int Width { get; private set; }
   public int Height { get; private set; }

   private Dictionary<int, GameObjectScript> _gameObjects;
   private IActionResolver _stateResolver;
   private IActionResolver _viewResolver;

   // Use this for initialization
   void Start ()
   {
      _gameProcessNetworkWorker = new GameProcessNetworkWorker();
      _battleId = LocalStorage.CurrentBattleId;
      _session = LocalStorage.Session;
      Side = LocalStorage.CurrentSide;
      IFieldFactory fact = new FieldFactoryStub();
```

```
        _pool = GetComponent<ObjectPool>();
        _gameObjects = new Dictionary<int, GameObjectScript>();

        StartCoroutine(Init());
        StartCoroutine(NetworkWorker());
    }

    private void InstantiateField()
    {
        GameObject tmp;
        //look through all the cells in array, wich reprsents the maze, and draw it
        //the coords of the graphical cells match the index of array elements
        for (int i = 0; i < Height; i++)
        {
            for (int j = 0; j < Width; j++)
            {

                //draw wall or floor
                try
                {
                    var point = new Point(i, j);
                    if (Field.StaticData.Cells[i, j].Object == FieldObject.Entrance)
                    {
                        tmp = Instantiate(Entrance, CoordinationHelper.GetViewPoint(point), Quaternion)
as GameObject;
                        tmp.transform.parent = transform;
                    }
                    if (Field.StaticData.Cells[i, j].Object == FieldObject.Road)
                    {
                        tmp = Instantiate(Road, CoordinationHelper.GetViewPoint(point), Quaternion) as
GameObject;
                        tmp.transform.parent = transform;
                    }
                    if (Field.StaticData.Cells[i, j].Object == FieldObject.Ground)
                    {
                        tmp = Instantiate(Ground, CoordinationHelper.GetViewPoint(point), Quaternion)
as GameObject;
                        tmp.GetComponent<CellController>().Point = new Point(i, j);
                        tmp.transform.parent = transform;
                    }
                    if (Field.StaticData.Cells[i, j].Object == FieldObject.Castle)
                    {
                        tmp = Instantiate(Castle, CoordinationHelper.GetViewPoint(point), Quaternion)
as GameObject;
                        tmp.transform.parent = transform;
                    }
                }
                catch (IndexOutOfRangeException e)
                {
                    Debug.Log(e.Message);
                }

            }
        }
    }

    public void RenderFieldState()
    {
        foreach (var tower in Field.State.Towers)
        {
            if(_gameObjects.ContainsKey(tower.GameId))
                continue;

            var obj = _pool.GetFromPool(tower.Type);
            obj.GameId = tower.GameId;
            _gameObjects.Add(tower.GameId, obj);
            obj.transform.position = CoordinationHelper.GetViewPoint(tower.Position);
        }
        foreach (var unit in Field.State.Units.ToArray())
```

```
        {
            if (_gameObjects.ContainsKey(unit.GameId))
            {
                var obj = _gameObjects[unit.GameId];

                obj.transform.position = FlyingObjects.Contains(obj.Type)
                    ? CoordinationHelper.GetViewPoint3(unit.Position)
                    : (Vector3)CoordinationHelper.GetViewPoint(unit.Position);
                continue;
            }

            var obj1 = _pool.GetFromPool(unit.Type);
            obj1.GameId = unit.GameId;
            _gameObjects.Add(unit.GameId, obj1);
            obj1.transform.position = FlyingObjects.Contains(obj1.Type)
                ? CoordinationHelper.GetViewPoint3(unit.Position)
                : (Vector3)CoordinationHelper.GetViewPoint(unit.Position);
        }

        //delete unexisting
        foreach (var gId in _gameObjects.Keys.ToArray())
        {
            if (!Field.State.Objects.ContainsKey(gId))
            {
                Debug.Log("Garbage collected " + gId);
                RemoveGameObject(gId);
            }
        }
    }

    public GameObjectScript GetGameObjectById(int id)
    {
        return _gameObjects[id];
    }

    public bool TryGetGameObjectById(int id, out GameObjectScript obj)
    {
        obj = null;
        if (!_gameObjects.ContainsKey(id))
        {
            return false;
        }
        obj = _gameObjects[id];
        return true;
    }

    public void RemoveGameObject(int id)
    {
        var obj = _gameObjects[id];
        _gameObjects.Remove(id);
        _pool.PutToPool(obj);
    }

    public void Command(Point? position = null)
    {
        StartCoroutine(PostCommand(Selected, position));
    }

    private IEnumerator PostCommand(GameObjectType type, Point? position, string cheat = null)
    {
        var command = new StateChangeCommandRequestModel
        {
            BattleId = _battleId,
            CurrentTick    = _tickCount,
            TowerCreationOptions = GameObjectLogical.ResolveType(type) == GameObjectType.Tower
                ? new []{new TowerCreationOption{Type = type, Position = position.Value}}
                : null,
            UnitCreationOptions = GameObjectLogical.ResolveType(type) == GameObjectType.Unit
                ? new []{new UnitCreationOption{Type = type}}
```

```
            : null,
         CheatCommand = cheat
      };
      var www = _gameProcessNetworkWorker.PostCommand(command);
      yield return www;
   }

   private IEnumerator NetworkWorker()
   {
      while (true)
      {
         if (Field == null)
         {
            yield return null;
            continue;
         }
         if (Field.StaticData.EndTimeUtc < DateTime.UtcNow)
         {
            yield return TryPostEnd();
         }
         var www = _gameProcessNetworkWorker.GetCheckBattleStateChange(_battleId, _revision);
         yield return www;
         if (!string.IsNullOrEmpty(www.text))
         {
            var ticks = JsonConvert.DeserializeObject<ActionsResponseModel>(www.text);
            _revision = ticks.Revision;
            Field.SetState(ticks.State);
            if (_resolver != null)
            {
               StopCoroutine(_resolver);
            }
            RenderFieldState();
            _resolver = StartCoroutine(ResolveActions(ticks.ActionsByTicks));
         }
         yield return new WaitForSeconds(0.4f);
      }
   }

   private IEnumerator Tick()
   {
      yield return new WaitForSeconds(TickSecond);
      _tickCount++;
   }

   private IEnumerator ResolveActions(IEnumerable<GameTick> actionList)
   {
      _tickCount = 0;
      foreach (var tick in actionList)
      {
         //var currentTick = _tickCount;
         //StartCoroutine(Tick());
         foreach (var action in tick.Actions)
         {
            _stateResolver.Resolve(action);
            yield return null;
            _viewResolver.Resolve(action);
         }
         _tickCount++;
         yield return new WaitForSeconds(TickSecond);
         //yield return new WaitUntil(() => currentTick < _tickCount);
      }
   }

   public void LeaveBattle()
   {
      StartCoroutine(PostEnd(Side.Invert()));
   }

   public void Cheat()
```

```
    {
        StartCoroutine(PostCommand(GameObjectType.Undefined, null, "addm"));
    }

    private IEnumerator TryPostEnd()
    {
        var www = new WwwWrapper(string.Format(ConfigurationManager.TryEndUrl, _battleId),
_session);
        yield return www.WWW;
    }

    private IEnumerator PostEnd(PlayerSide winner)
    {
        var www = new WwwWrapper(string.Format(ConfigurationManager.TryEndUrl, _battleId),
_session);
        yield return www.WWW;
        StartCoroutine(Leave(winner));
    }

    private IEnumerator Leave(PlayerSide playerSide)
    {
        Winner = playerSide;
        yield return new WaitForSeconds(3);
        SceneManager.LoadScene("StartPages");
    }

    public void End(PlayerSide winner)
    {
        StartCoroutine(PostEnd(winner));
    }

    private IEnumerator Init()
    {
        var www = new WwwWrapper(string.Format(ConfigurationManager.InitFieldUrl, _battleId),
_session);
        yield return www.WWW;
        Field = JsonConvert.DeserializeObject<Field>(www.WWW.text);

        _stateResolver = new FieldStateActionResolver(Field);
        _viewResolver = new ViewActionResolver(this);

        Width = Field.StaticData.Width;
        Height = Field.StaticData.Height;
        CoordinationHelper.Init(Width, Height);
        InstantiateField();
    }
```

ДОДАТОК В

Тези доповіді на молодіжному форумі «Радіоелекторніка та молодь у XXI ст»

ИГРОВОЕ ВЕБ-ПРИЛОЖЕНИЕ В ЖАНРЕ "TOWER DEFENCE"

Топчий А.Д.
Научный руководитель – к.т.н., доцент Мазурова О.А.
Харьковский национальный университет радиоэлектроники
(61166, Харьков, пр. Науки,14, каф. Программной инженерии, тел.
(057) 702-14-46)
e-mail: oleksandr.topchii@nure.ua; телефон (095) 603-11-03

The given work is devoted to developing gaming web-application for personal computers. The game is developed in Tower Defense genre which relates to Real-time strategies. There are a lot of games in this genre. However, existing analogues supports only in single player mode. In addition, the behavior of game objects is not optimized enough. This gaming web-application allows users to fight with other players. The behavior of game objects should be based on an intellectual component that takes into account different criteria with different priorities for choosing the optimal paths, or the optimal targets for attack.

Компьютерные игры в последние годы заняли значительное место на рынке развлечений и досуга. Одним из наиболее распространённых и популярных жанров видеоигр является жанр стратегий в реальном времени (Real-time strategy). Именно к данному жанру относятся игры в стиле Tower Defense.

На сегодняшний день существует множество игр данного жанра. Например, такие игры как Defense Zone 2 HD, Kingdom Rush Origins, Tower Storm, Anomaly 2 являются достаточно популярными среди пользователей и предлагают различные особенности геймплея. Однако, несмотря на масштабность данных игр и наличие качественной графики, эти программные продукты имеют ряд недостатков. В частности, многие из подобных игр не обладают достаточно проработанными алгоритмами, которые позволяли бы игровым объектам принимать наиболее эффективные и оптимальные решения во время передвижения, поиска целей, сражения и т.д.

Таким образом, актуальной является задача разработки игровой веб-системы, в которой поведение игровых объектов было бы основано на оптимизационных моделях. Использование таких моделей при выборе путей либо целей для атаки, которые будут учитывать различные критерии оптимизации, сделает поведение игровых сущностей более разнообразным и интересным.

Для моделирования игровой ситуации были разработаны:

– база данных, которая содержит информацию о пользователях системы, боях между игроками, информацию о всех игровых объектах с их характеристиками (характеристики юнитов и башен) и т.п.;

– графовая модель для программного представления путей расположенных на игровом поле системы, которая позволяет оптимизировать движение игровых объектов по карте и поиск путей;

– оптимизационная модель многокритериальной задачи принятия решений для нахождения оптимальных путей для движения юнитов и оптимальных целей для атаки башен.

Для поиска кратчайших путей на карте была спроектирован и реализован алгоритм, основанный на алгоритме Дейкстры нахождения кратчайшего пути между вершинами на графе [1]. Данный алгоритм используется при движении некоторых видов юнитов.

Для поиска оптимального пути нападающего юнита адаптирован алгоритм взвешенной суммы оценок на основании аддитивной свертки с взвешенными коэффициентами [2], который учитывает следующие критерии:

– общее количество защитных башен на пути;

– безопасность пути: количество единиц урона, которые потенциально могут нанести юниту башни на пути;

– потенциальное количество ресурсов, которые может получить юнит, пройдя по пути или его части;

– отношение защитных башен на пути с особыми эффектами при атаке к общему количеству башен на пути.

В результате была разработана игровая веб-система, которая выполняет следующие основные функции:

– возможность участия пользователя в битве с другим пользователем, установки на карте новых башен, запуска новых монстров;

– автоматический расчет на сервере порядка действий всех игровых объектов на карте для каждого состояния битвы;

– поиск оптимальных путей юнитов и оптимальных целей для атаки башен на основании многокритериальной оптимизационной модели и т.д.

Новизна разработки заключается в наличии математической составляющей, которая позволяет игровым объектам находить оптимальные решения, за счёт чего поддерживается более разнообразный геймплей. Кроме того, новизна разработанной системы заключается в возможности сражаться и соревноваться в режиме реального времени с другими игроками, что не распространено среди игр представленного жанра.

Список источников:

1. Волченская Т.В., Князьков В.С. Компьютерная математика: Часть 2. Теория графов: Учебное пособие. - Пенза: Изд-во Пенз. гос. ун-та, 2002. – 101 с.

2. Лотов А.В., Поспелова И.И. Многокритериальные задачи принятия решений: Учебное пособие. – М.:Издательский отдел факультета ВМиК МГУ им. М.В. Ломоносова; МАКС Пресс, 2008.

ДОДАТОК Г

Стаття для науково-технічного семінару «CMIS-2020»

A Study of Optimization Models for Creation of Advanced Artificial Intelligence for the Computer Game in the Tower Defense Genre

O. Topchii [1][0000-0003-0322-7772], O. Mazurova [2][0000-0003-3715-3476], O. Samantsov[3][0000-0002-4708-4144], M. Shirokopetleva[4][0000-0002-7472-6045]

[1] Kharkiv National University of Radioelectronics, Nauka av. 14, Kharkiv, 61166, Ukraine, oleksandr.topchii@nure.ua

[2] Department of Software Engineering, Kharkiv National University of Radioelectronics, Nauka av. 14, Kharkiv, 61166, Ukraine, oksana.mazurova@nure.ua

[3] Department of Software Engineering, Kharkiv National University of Radioelectronics, Nauka av. 14, Kharkiv, 61166, Ukraine, oleksandr.samantsov@nure.ua

[4] Department of Software Engineering, Kharkiv National University of Radioelectronics, Nauka av. 14, Kharkiv, 61166, Ukraine, marija.shirokopetleva@nure.ua

Abstract: The work is devoted to the study of optimization models of decision theory for the development of artificial intelligence to control the computer strategy in the game of the Tower Defense genre.

In games of this genre, the strategy of a computer depending on the side of the player may be either to send monsters to certain paths of the game card or to place defensive towers along those paths. To simulate the behavior of artificial intelligence, mathematical models of the optimization problem of assigning monsters to paths and the task of assigning towers to map cells were designed. Artificial intelligence, implemented on the basis of such models, can be classified as advanced intelligence. Multi-criteria convolutional models based on utility theory are proposed for modeling of medium complexity intelligence.

An experimental study of the developed models and algorithms for solving the corresponding problems is carried out, taking into account the indicators of the calculated complexity and efficiency of the behavior of artificial intelligence. Based on the conducted experiments, recommendations for the use of medium and advanced artificial intelligence in the game in the genre of Tower Defense were made.

Keywords: Game, Gaming System, Artificial Intelligence, Tower Defense, Linear Programming, Convolution method, Optimization Problem, Advanced Artificial Intelligence, Mathematical Model, Experiment.

1 Introduction

Among the variety of computer games, real-time strategy games are of great interest. In strategy games, the player is faced with many difficult tasks: the player must react in real-time to certain events and to think over his strategy in order to beat his opponent. Human's opponent in such games is artificial intelligence, which in order to ensure a certain gameplay interest must respond to the actions of the player in real-time, make various and unpredictable decisions.

One of the most popular types of strategy games is strategy games in the Tower Defense genre. For games in this genre there are the following general features:

- there is a map with one or more entry points and one or more exits;
- there are winding paths between these points;
- players operate with different game objects that can be divided into two classes: attackers (such as monsters) and defenders;
- the main purpose of the attackers is to go from entry to exit (or specific target) through the paths on the map;
- the main purpose of the defenders is to create obstacles in the way of attackers by building attacking towers along the roads;
- game objects (towers and attackers) have different characteristics.

At the moment, the main games in the genre which are presented in the market offer players quite straightforward actions by an artificial opponent, who are usually pre-programmed to make certain decisions in advance, and do not respond to the actions of the player in real time. Players need an opponent who can make various decisions of varying levels of complexity and unpredictability. Thus, such games require the use of advanced artificial intelligence.

2 Analysis of basic researches

At the moment, game developers have not fully decided which approach to artificial intelligence is more effective: pre-programmed strategies, i.e. lack of intelligence, or advanced artificial intelligence (AI), where AI actions depend on the specific actions of the player [7, 15].

General researchers divide AI in games into three categories: basic AI, highly reactive AI (or medium), advanced AI (AAI) [8]. Games with basic AI are the least demanding for computer resources, but in terms of gaming experience, they are not very interactive and do not respond to the player's actions. High-reactive AI games require a lot of computational resources, but they provide the player with variable gameplay experience and adapt to player action during the gameplay. Advanced AI games require the most computing power, but throughout the game, they use more calculated, optimal strategies, learn and adapt to player style [11].

It needs to be noted that there is still debate over which type of AI is the most efficient to use in different kinds. Moreover, the popularity of games among players often depends on what kind of AI is used in game. Some researchers believe that the use of AAI is not always appropriate, and in some cases, it is more appropriate to use highly

reactive AI or even basic intelligence. Some statistics show that games with advanced AI are not very popular among users [13] due to the use of sophisticated mathematical calculations, which lead to considerable computational cost. It is against this background that games with basic AI are more popular.

Adherents of the developed artificial intelligence propose to use various mathematical models [12], neural networks [14] and genetic algorithms [2]. Today there are various studies of the development of mathematical models of artificial intelligence for games in the Tower Defense genre [3]. However, a pre-programmed strategy dominates among games. That is, not-advanced artificial intelligence is used. This is justified primarily by the economy of computer resources. This is justified primarily by the savings in computer resources and the fact is that, as a rule, such games do not require much gameplay variability from the computer.

Some researchers consider the AAI as a decision-making intellect [9]. Therefore, it is possible to approach the AAI modeling by using models of decision theory (DT) [10].

Among the diversity of DT approaches, we need to mention such methods as resource-efficient multi-criteria analysis and various convolutions [4], which allow evaluating the usefulness of a solution, taking into account the various quality criteria of such decisions. In a computer game, these criteria usually take the maximum health points, speed, location, and form of movement of the game units, range, firing rate, etc. [1]. Convolutional multi-criteria analysis methods can be used in the development of strategy game algorithms, not only to find optimal paths but also for other needs, in particular, to find the optimal target among attackers. On the basis of this approach, an average AI can be constructed.

The theory of operations [5] is a quite popular trend in DT, as well as optimization problems of mathematical programming, which allow modeling of real processes in various subject areas. Mathematical programming methods of assignment problems [6] allow to adequately model problems where there are so-called tasks and resources that can be allocated to perform such tasks. In computer games, you can use similar assignment tasks to assign game units to game paths. This approach, based on mathematical models with defined optimization functions and sets of constraints, allows us to obtain optimal solutions to assignment problems and, thus, to build more sophisticated intelligence [15].

3 Problem Statement

Therefore, the task is to conduct mathematical modeling of artificial intelligence for the strategy game in the Tower Defense genre on the basis of optimization models of decision theory and to conduct their research with the aim of creating effective AAI. To achieve this goal it is necessary to:

- to develop mathematical models of AI behavior in the game in the Tower Defense genre based on multicriteria DT and assignment tasks problems of mathematical programming;

- to develop algorithms for solving optimization problems based on the developed models;
- to plan and conduct experimental research of the developed algorithms and models;
- create recommendations on when to use different approaches to create advanced AI in strategy games in the Tower Defense genre.

4 Mathematical modeling of game artificial intelligence

4.1 Modeling of game units

Based on the game description, the following mathematical notations for game entities were introduced:

- $\{U_i\}_{i=1}^{m}$ – the set of units *Ui*, that are available to be added on the game filed;
- $P = \{P_i\}_{i=1}^{n}$ – the set of paths *Pi* on the game field;
- $T = \{T_i\}_{i=1}^{K}$ – the set of all of the towers, that are available to be added on the game filed;
- $P^* = \{P_{(x,y)}^*\}_{x=1,y=1}^{X,Y}$ – the set of cells on the game field $X \times Y$.
- X – game field width;
- Y – game field length;
- C_i – the cost of unit i (or tower) in in-game resources;
- Q_i – attack points of unit i (damage to the castle);
- H_i – maximum health points of unit i;
- V_i – attack points of tower i (damage to unit's health);
- M_M – amount of in-game resources of attacker player;
- M_T – amount of in-game resources of defender player.

To optimally find paths between different points on the playing field, a graphical model of storing paths on the field in program system memory as an oriented graph was developed. This approach simplifies and accelerates the calculation of the route of movement of game entities [3].

4.2 Artificial intelligence modeling based on utility theory

To design AI let's consider the behavior of a computer on the side of attack monsters. His target is to destroy the castle of an enemy player. For this purpose, artificial intelligence must inflict the maximum possible damage on the enemy. Each game unit has the points of damage it inflicts on the castle if it reaches it. However, it is not always important for one unit to reach the castle, as it can potentially help other units to get to the castle, for example by diverting the towers' attack on itself. That is, units depend on other units.

Thus, for all of the units that can be purchased at the moment (that is, enough resources), it is necessary to calculate the damage for each of the potential ways, taking into account their price (we will divide the potential damage by the price). From the obtained values we choose the largest. AI needs to calculate the potential damage that can be caused to the castle when the unit launches on the field.

The task of selecting a unit and its path on the playing field can be modeled by utility theory, namely the linear additive convolution model:

$$Z = \sum_{j=1}^{n} \alpha_j \beta_j K_j(P_i), \quad (1)$$

here α_j- normalizing multiplier, β_j- weight factors, that depend on the relative contribution of all criteria to the overall criterion, $K_j(P_i)$– the criterion score by the criterion K_j for the path variant P_i. At the same time, usually, the weighted coefficients are set with already normalized values [4].

This method allows us to effectively choose options based on many criteria. That is why this method was considered for use in the game system.

In the developed model (1) the following criteria with weighting factors were proposed:

- K_1 - total number of protective towers on the way (weight factor $\beta_1 = 0{,}1$);
- K_2 - path safety: the number of damaged units that can potentially inflict a tower attack on the path ($\beta_2 = 0{,}5$);
- K_3 - the potential amount of resources that the attacker can gain by following the path or part of it ($\beta_3 = 0{,}3$);
- K_4 - the ratio of defensive towers on the path with some special effects to the total number of towers on the path ($\beta_4 = 0{,}1$).

When selecting the unit to be added to the game field, these calculations must be performed for each unit. And then the option with the greatest value must be selected.

4.3 Modeling of artificial intelligence based on mathematical programming problems

For the creation of AI-based on linear programming problems, the behavior of the computer for both opponents was considered:

- for the attacker in time of adding the unit (monster) on the game field to a certain path;
- for the defender in time of placing defensive towers to certain cells on the game field.

The task of adding a unit to a certain path can be classified as an optimization task of an assignment problem. It can be formulated as follows: to add (assign) existing units on the game field, the path needs to be chosen in such a way as to minimize the damage that will be taken by the units as they pass these paths. A table view of this optimization task is provided in Table 1.

To model the task of assigning units to paths, we introduce the following additional notation:

- $\{U_i\}_{i=1}^{m}$ – the set of units *Ui*, that are available to be added on the game filed;
- $P = \{P_i\}_{i=1}^{n}$ – the list of paths *Pi* on the game field;
- C_i – the cost of unit *i* in in-game resources;
- Dij – potential damage that can be inflicted on the enemy's castle in case the unit *i* is added to path *j*.

Potential damage *Dij* is calculated by the formula (2):

$$D_{ij} = Q_i L_{ij} + \sum_{u=0}^{ij} Q_u L_{uij}, \quad (2)$$

where Q_i – attack points of unit i, L_{ij} – the fact that the unit i reaches the end of the playing field using the path j or not, can take the value 0 or 1:

$$L_{ij} = \begin{cases} 0, H_i > T_{ij} \\ 1, H_i \leq T_{ij}, \end{cases} \quad (3)$$

where H_i – health points of unit i, T_{ij} – potential damage that will be inflicted on unit i on the path j.

Table 1. Optimization problem for assigning a unit to the path

Paths / Units	P_1	P_2	…	P_n	Units limitations U_i
U_1 (C_1)	D_{11}, X_{11}	D_{12}, X_{12}	…	D_{1n}, X_{1n}	1
U_2 (C_2)	D_{21}, X_{21}	D_{22}, X_{22}	…	D_{2n}, X_{2n}	1
…	…	…	…	…	
U_m (C_m)	D_{m1}, X_{m1}	D_{m2}, X_{m2}	…	D_{mn}, X_{mn}	1
Paths limitations P_j	1	1		1	

Let's introduce a variable *Xij*, showing whether the i-th unit was added on the j-th path, the variable can be set to 1 (if the unit was added) or 0 (if it wasn't).

For this formulation, the following conditions must be satisfied:

- the price of the unit *i* must be less than or equal to the number of resources the attacker player has $Ci < MM$, where MM – the amount of in-game resources of attacker player;
- each unit can be assigned to no more than one path, therefore $\sum_{j=1}^{n} x_{ij} \leq 1, \forall_i = 1, m$;
- only one unit can be assigned to each path, therefore $\sum_{i=1}^{m} x_{ij} = 1, \forall_j = 1, n.$;
- the unit *i* may be or may not be assigned to the path *j*, therefore $X_{ij} = 1|0, \forall i = 1, m, \forall j = 1, n.$

The objective function must ensure that the units are assigned to a path that maximizes the potential damage to the castle of defender opponent (equivalent to the minimum damage to the attacker player), therefore

$$F = \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{D_{ij} X_{ij}}{C_i} \rightarrow max, \quad (4)$$

or

$$F = \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{X_{ij}(Q_i L_{ij} + \sum_{u=0}^{U} Q_u L_{uij})}{C_i} \rightarrow max. \quad (5)$$

To sum up, the mathematical model of the optimization problem for artificial intelligence of assigning units to paths takes the following form:

$$F = \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{X_{ij}(Q_i L_{ij} + \sum_{u=0}^{U} Q_u L_{uij})}{C_i} \rightarrow max, \quad (6)$$

$$\sum_{j=1}^{n} x_{ij} \leq 1, \forall_i = 1, m; \quad (7)$$

$$\sum_{i=1}^{m} x_{ij} = 1, \forall_j = 1, n; \quad (8)$$

$$X_{ij} = 1|0, \forall i = 1, m, \forall j = 1, n. \quad (9)$$

When playing on the side of the towers, the purpose of AI is to protect its castle, that is, to destroy the units of the attacker. To do this, it must deal damage and destroy units. When the tower is placed on the field, it does not disappear from it, and is able to influence the subsequent gameplay. In this case, when placing a tower in a particular cell on the game field, the values such as how much damage will be done to the units of the enemy, who are currently on the field; how many of units will be destroyed; the potential damage that can be inflicted by the tower from a selected position in the future.

It should also be mentioned that in order to simplify the calculations, we may calculate not all of the cells that are on the game field, but only those the sum of which indices are divided by a certain random number. In this way, several cells from different parts of the playing field will be uniformly selected, allowing the most efficient calculation. That is, we select cells *C[x, y]* for which *(x + y) mod r = 0*, where *r* is a random number from 1 to 10.

Thus, the task of placing towers on the playing field in a certain position can be modeled as an optimization task for linear programming, namely as a task for the purpose.

A table view of this optimization task is provided in the Table 2.

To model the task of assigning towers to cells, we introduce the following additional notation:

- $T = \{T_i\}_{i=1}^{k}$ – the set of all of the towers, that are available to be added on the game filed;
- $P^* = \{P^*_{(x,y)}\}_{x=1,y=1}^{X,Y}$ – the set of cells on the game field $X \times Y$.
- X – game field width;
- Y – game field length;
- C_i – the cost of tower i in in-game resources.

The set of cells P^* is being selected as follows: P*=U–W–B, where U – set of all of the cells on the game field, W – set of cells that are included in any of paths, B – set of cells that are already occupied by towers.

In this case, only cells that meet the following condition are selected: $(x + y) \bmod k = 0$, where k is any number from 0 to 5, which is randomly selected.

Table 2. Optimization problem of assigning a tower to a cell

Towers / Cells	$P_{(1,1)}$	…	$P_{(x,y)}$	Towers limitations T_i
T_1 (C_1)	$D_{1(1,1)}$,$A_{1(1,1)}$,$K_{1(1,1)}$, $B_{1(1,1)}$	…	$D_{1(x,y)}$,$A_{1(x,y)}$,$K_{1(x,y)}$, $B_{1(x,y)}$	1
IT_2 (C_2)	$D_{2(1,1)}$, $A_{2(1,1)}$,$K_{2(1,1)}$,$B_{2(1,1)}$	…	$D_{2(x,y)}$,$A_{2(x,y)}$,$K_{2(x,y)}$, $B_{2(x,y)}$	1
…	…	…	…	
T_m (C_m)	$D_{m(1,1)}$, $A_{m(1,1)}$,$K_{m(1,1)}$,$B_{m(1,1)}$	…	$D_{m(x,y)}$,$A_{m(x,y)}$,$K_{m(x,y)}$,$B_{m(x,y)}$	1
Cells limitations $P^*_{(x,y)}$	1		1	

$A_i(x,y)$– damage that may be inflicted on enemy units that are currently on the field if tower i is placed in position (x,y).

$A_i(x,y)$ is calculated as follows:

$$A_{i(x,y)} = \sum_{u=0}^{U}(Z_{iu}F_{ui(x,y)} + \sum_{t=0}^{T} Z_{tu}F_{tui(x,y)}), \qquad (10)$$

where U – set of units that is on the field at the moment; T - set of towers that is on the field at the moment; Z_{tu}- damage that can be inflicted on unit u from tower t; $F_{ui(x,y)}$ – the number of times when unit u is attacked by tower i, if the tower i is placed in the cell (x,y); $F_{uti(x,y)}$ – fact whether unit u is attacked by tower t, or not if the tower i is placed in the cell (x,y) (can take the value 0 or 1).

$K_i(x,y)$ – the number of units that are currently on the playing field and will be destroyed by placing the tower i in position (x,y). $K_i(x,y)$ is calculated as follows: $K_{i(x,y)} = \sum_{u=0}^{U} L_{ui(x,y)}$, where L_{uij} - the fact whether unit u gets the finish, or not, if the tower i is placed in the cell (x,y). Can take the value 0 or 1: $L_{uij} = \begin{cases} 0, H_u > T_{uij} \\ 1, H_u \le T_{uij} \end{cases}$,

where H_u – health points of the unit u; T_{uij} – the number of damage inflicted on unit u, if the tower i is placed in the cell (x,y).

$D_i(x,y)$ – the potential damage that can be inflicted on the units at cells at the range of tower attack the tower i is placed in the cell (x,y). $D_i(x,y)$ is calculated as follows: $D_{i(x,y)} = V_i N_{i(x,y)}$, where Vi – attack points of the tower i; $N_{i(x,y)}$ – the number of cells, that can be attacked by the tower i at position (x,y).

Let's introduce a variable $B_i(x,y)$, showing whether the i-th tower was placed on the cell (x,y). $B_i(x,y)$the variable can be set to 1 (if the tower was added) or 0 (if it wasn't).

For this formulation, the following conditions must be satisfied:

- the price of the tower i must be less than or equal to the number of resources the defender player has $Ci < MT$, where MT – the amount of in-game resources of defender player;
- each tower can be assigned to no more than one cells, therefore $\sum_{j=1}^{n} b_{ixy} \leq 1, \forall_i = 1, m$;
- only one tower can be assigned to each cell, therefore $P^*(x,y)$. therefore $X_{ij} = 1|0, \forall i = 1, m, \forall j = 1, n$.

The objective function is formulated as follows:

$$F = \sum_{i=1}^{m} \sum_{x=1}^{X} \sum_{y=1}^{Y} \frac{(A_{i(x,y)} + 1)(K_{i(x,y)} + 1) D_{i(x,y)} X_{i(x,y)}}{C_i} \rightarrow max.$$

To sum up, the mathematical model of the optimization problem for artificial intelligence of assigning towers to cells takes the following form:

$$F = \sum_{i=1}^{m} \sum_{x=1}^{X} \sum_{y=1}^{Y} \frac{(A_{i(x,y)+1})(K_{i(x,y)+1}) D_{i(x,y)} X_{i(x,y)}}{C_i} \rightarrow max, \quad (12)$$

$$\sum_{x=1}^{X} \sum_{y=1}^{Y} b_{ixy} \leq 1, \forall_x = 1, X, \forall_y = 1, Y, \quad (13)$$

$$\sum_{i=1}^{m} b_{ixy} = 1, \forall_j = 1, n, \quad (14)$$

$$B_{ij} = 1|0, \forall i = 1, m, \forall_x = 1, X, \forall_y = 1, Y. \quad (15)$$

In the practice of operational management, simplex and transport methods, greedy algorithm, Hungarian algorithm, and others are typical for solving linear programming problems. The Simplex method can be considered as basis for solving this class of problems. It is an algebraic procedure, in which the researcher consistently approaches to finding the optimal solution. In theory, this method can be used to solve problems that include any number of constraints and variables, but if there are more than four variables or constraints in them, then the computation becomes quite complicated.

A greedy algorithm is a simple and straightforward heuristic algorithm that makes the best decision based on the data available at the current stage, without worrying

about the possible consequences, hoping eventually to find the best solution. It is easy to implement and often very time-efficient.

To solve the problem of assigning units to paths and the problem of assigning towers to positions, algorithms were developed and implemented based on the simplex method and on the basis of the greedy algorithm.

5. Conducting experiments and its conclusions

For the creation of the effective developed AI, it was decided to conduct an experimental study of AI behavior during assigning units to paths or assigning towers to specific positions on the game field based on developed mathematical models. The following prerequisites were used in planning the series of experiments:

- the basic AI was not considered in the series of experiments; his experimental study was conducted separately; the study found that while it is the least demanding for computing resources, it is not very interactive and does not provide the necessary level of victory over the human opponent;
- as a highly reactive (or medium) AI, the behavior based on the developed multicriteria selection model was considered (1);
- the advanced AI was considered as the behavior of AI based on of the developed models of optimization problems of assignments (6-9) and (12-15).

Two series of experiments were planned and conducted.

In the first series of experiments the following algorithms were investigated:

- the algorithm for assigning units to paths through multi-criteria selection based on a linear additive convolution (AC1);
- the greedy algorithm of solving the problem of assigning a unit to the paths (GA1);
- the basic algorithm of assigning units to paths based on the simplex method of calculating the values of the optimization model (SM1).

In the second series of experiments the following algorithms were investigated:

- algorithm for assigning towers to cells by multi-criteria selection based on linear additive convolution (AZ2);
- greedy algorithm of solving the problem of assigning towers to cells (GA2);
- the basic algorithm for assigning towers to paths based on the simplex method(SM2).

The experiments were performed on games of different sizes:

- (Game1): number of available units - 5, field size 10X10, number of available towers 3, number of the path – 2;
- (Game2): number of available units - 8, field size 20X20, number of available towers 5, number of the path – 5;
- (Game3): number of available units - 10, field size 50X50, number of available towers 7, number of the path – 10.

For each series of experiments, 100 game sessions were conducted. In sessions the effectiveness of the selected methods was tested using the following metrics:

- the usage of computing resources, namely milliseconds of time, to make one decision;
- the percentage of AI wins against the other side
- RAM used.

All studies were conducted on a single physical server that has the following characteristics:

- CPU Intel Xeon E-2224;
- processor frequency 3,4 Ghz;
- cache 8 Mb;
- 4 cores (8 logical cores);
- RAM 8 Gb 2666 MHz.

In order to ensure a sufficiently high number of study series, all the battles took place automatically and without human involvement, that is, the AI competed with itself based on the developed models. In the 100-game series, all 3 developed models were used in equal proportions as an AI opponent. This made possible to model a fairly knowledgeable player. Of course, the absence of a real person led to a certain decrease in the percentage of AI victories against such a stronger opponent (also AI). But in the aggregation, it allowed obtaining quite adequate results.

The results of the study of the first series of experiments for "Game1", "Game2" and "Game3" are shown in tables 3, 4 and 5, respectively.

Table 3. Results of the first series of experiments for Game1

Metrics / Algorithm	Average time of one algorithm operation, ms	Average RAM, KB	Attacker player victory percentage,%
AC1	9	312	44
SM1	82	1016	48
GA1	51	956	48

Table 4. Results of the first series of experiments for Game2

Metrics / Algorithm	Average time of one algorithm operation, ms	Average RAM, KB	Attacker player victory percentage,%
AC1	32	425	42
SM1	196	1598	47
GA1	117	1056	46

We can conclude that the results are quite regular and that for games of different sizes, the modified greedy algorithm is the most optimal since it gives a significant performance gain and almost no efficiency loss compared to the base one (simplex

method). However, although the linear additive convolution method uses much fewer resources, it does not provide a sufficient level of efficiency.

Table 5. Results of the first series of experiments for Game3

Metrics / Algorithm	Average time of one algorithm operation, ms	Average RAM, KB	Attacker player victory percentage,%
AC1	183	531	39
SM1	567	1900	46
GA1	479	1576	45

The results of the study of the second series of experiments for "Game1", "Game2" and "Game3" are shown in tables 6, 7 and 8, respectively.

Table 6. Results of the second series of experiments for Game1

Metrics / Algorithm	Average time of one algorithm operation, ms	Average RAM, KB	Defender player victory percentage,%
AC2	54	988	47
SM2	801	7011	52
GA2	591	5678	53

Table 7. Results of the second series of experiments for Game2

Metrics / Algorithm	Average time of one algorithm operation, ms	Average RAM, KB	Defender player victory percentage,%
AC2	870	1492	43
SM2	2570	8329	55
GA2	1961	6571	55

Based on the results of the experiments, we can conclude that the results show a pattern. For games of varying sizes, the modified greedy algorithm is the most efficient because it gives a significant performance gain over convolutional performance and almost no performance loss over the base (simplex method). In this case, the linear additive convolution method, although using much less resources, does not provide a sufficient level of decision efficiency.

Table 8. Results of the second series of experiments for Game3

Metrics / Algorithm	Average time of one algorithm operation, ms	Average RAM, KB	Defender player victory percentage,%
AC2	1081	1912	41
SM2	4132	9221	56
GA2	3006	7221	55

The model, built for the AI for the side of the defenders (towers), showed a higher computational cost due to its complexity, which is due to the considerable game load of units such as towers. Meanwhile, decisions to build defensive towers are inherently more strategic, so the timing is quite receptive to games in the genre.

Based on the series of experiments, we formulate recommendations for the use of advanced and medium AI in games of the Tower Defense genre:

1) a multi-criteria convolution-based AI behavior model gives significant savings on computing resources for games of any size, but is inferior to other models in terms of victories and, in general, does not allow it to cross the 50% threshold. Such indicators allow us to recommend a model for the implementation of medium intelligence, but it is not recommended for the implementation of AI for the Expert gameplay level;
2) non-linear growth of computing resources on the dimension of the game, meanwhile, allows to recommend a multi-criteria convolution AI model (AC1 and AC2 algorithms) for oversized playing fields (experiments for these types of fields were not carried out); such model provides a satisfactory level of decision efficiency and has a much better computational speed;
3) the AI model that uses the basic algorithm for its solution (SM1 and SM2) is not generally recommended for implementation, because it is almost at the same level of victories (+ 1%) as when using the greedy algorithm (GA1 and GA2), shows much worse computational performance;
4) the best option for implementing advanced artificial intelligence (for medium and large game fields like Game2 and Game3) is the AI model that uses the greedy algorithm (GA1 and GA2), as they provide a good level of victory and have a decent computing speed.

Conclusions

The article proposes mathematical models for the creation of artificial intelligence for a strategy game in the Tower Defense genre based on optimizations models of decision theory:

- multi-criteria convolutional model for AI for the assignment attackers (monsters) to the game paths and defensive towers on game field cells;

- an optimization model of the assignment problem for AI for attacker behavior, when adding a monster on the game field to a certain path;
- an optimization model of the assignment problem for AI for attacker behavior, when adding a monster on the game field to a certain path.

Algorithms for solving problems based on the designed models are developed. These algorithms demonstrate different levels of computational complexity. Algorithms based on additive linear convolution, greedy algorithm and, the simplex method were investigated.

For the purpose of creating efficient AAI, it is planned and conducted an experimental study of the developed algorithms and models. Recommendations for using models in order to create advanced AI in strategies in the Tower Defense genre were proposed.

References

1. Jason Gregory, Game Engine Architecture 2nd edition, 1304 c. – 2011.
2. Chakroborty, P., and Dwivedi, T. Optimal route network design for transit system using genetic algorithms. Optimization and Engineering. 34(1), 200 c. — 2003.
3. Jensen, B., Togelius, J., Nielsen, J., AI for General Strategy Game Playing. — ResearchGate, 2014. URL: https://www.researchgate.net/publication/275963230.
4. Lotov A.V., Pospelova I.I. Multicriteria decision-making problems: Textbook, 197 p. - 2008.
5. Bertsimas, D. and Tsitsiklis, J. N. Introduction to Linear Optimization, 608 c. — 1997.
6. William J. Cook, William H. Cunningham, William R. Pulleyblank, Alexander Schrijver, Combinatorial Optimization 1st Edition, 287 c. — 1998.
7. Nareyek A. AI in computer games//Queue. 2004. No 10 (кн. 1). С. 58–65. URL: http://doi.acm.org/10.1145/971564.971593
8. Yildirim S., Stene S. A survey on the need and use of AI in game agents//Proceedings of the 2008 Spring Simulation Multiconference, ser. SpringSim'08/ Сан-Диего, CA, США: Society for Computer Simulation International, 2008. С. 124–131. URL: http://dl.acm.org/citation.cfm?id=1400549.1400565
9. Millington I., Funge J. Artificial Intelligence for Games, 2nd ed. — San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. — 896 c.
10. L.Vlasenko, A. Chikrii On a differential game in a system with distributed parameters // Proceedings of the Steklov Institute of Mathematics, 2016, vol. 292 (1), P. 276-285 URL: https://link.springer.com/article/10.1134/S0081543816020243
11. Spronck P., Ponsen M., Sprinkhuizen-Kuyper I., Postma E. Adaptive game AI with dynamic scripting//Machine Learning. 2006. No 3 (кн. 63). С. 217–248. URL: http://dx.doi.org/10.1007/s10994-006-6205-6
12. Bakkes S., Spronck P., van den Herik J. Opponent modelling for case-based adaptive game AI//Entertainment Computing. 2009. No 1 (кн. 1). С. 27–37.
13. Csikszentmihalyi M. Flow: The Psychology of Optimal Experience. — New York, NY, USA, Harper and Row, 1990. — 456 c.
14. Pfeifer R., Scheier C. Understanding Intelligence. — Cambridge, MA, USA, MIT Press, 1999. — 700 c.
15. Yannakakis G., Togelius J. Artificial Intelligence and Games. — Springer, 2018. URL: http://gameaibook.org.