



Ministério da Educação
Secretaria de Educação Profissional e Tecnológica
Instituto Federal Catarinense
Campus Rio do Sul

CLEVERTON HOFFMANN

INTELIGÊNCIA ARTIFICIAL APLICADA NO DOMINÓ DE DUAS PONTAS

Bacharelado em Ciência da Computação

Inteligência Artificial

Juliano Tonizetti Brignoli

Rio do Sul

2021

SUMÁRIO

1 INTRODUÇÃO	3
2 - REVISÃO BIBLIOGRÁFICA	3
2.1 Jogo de dominó	3
2.2 MinMax, Métodos de Busca e Estratégias Heurísticas aplicadas no Jogo de Dominó	5
3 - DESENVOLVIMENTO	7
4 - CONCLUSÃO	12
5 - REFERÊNCIAS	12
6 - ANEXOS	13

1 INTRODUÇÃO

Atualmente jogos são utilizados para a distração, lazer e estimulação do raciocínio lógico. Além disso muito explorados em cursos superiores de computação e das áreas exatas, por conterem um nível de complexidade a ser resolvido que muitas vezes se torna maçante e impossível de resolver manualmente. O jogo de dominó muito conhecido no mundo todo, utilizado em rodas de família e amigos como um entretenimento. Existem diversas formas de jogar e regras utilizadas no dominó variando de região para região. Essas variações e regras determinam também a dificuldade de resolução do mesmo.

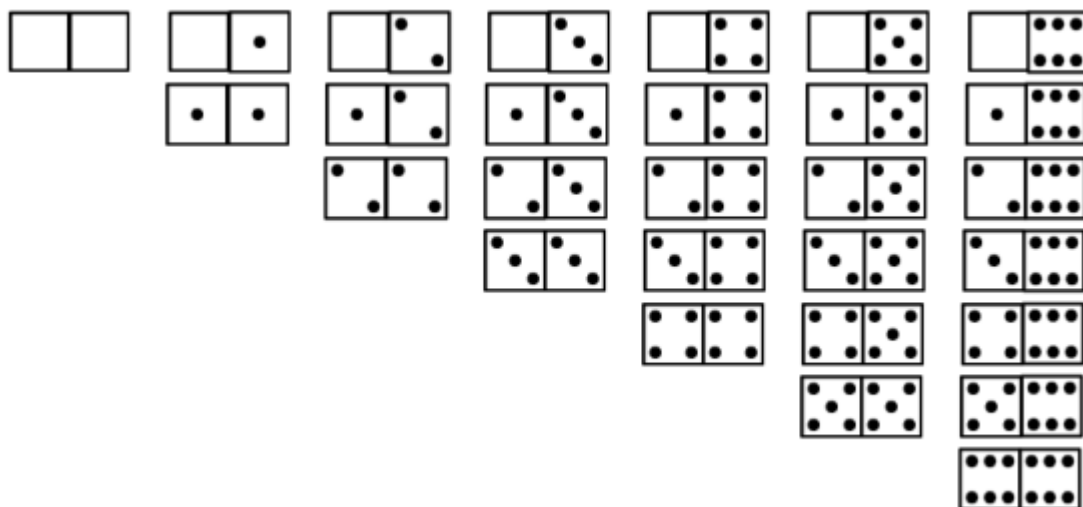
Uma das áreas específicas da computação que explora a resolução de problemas complexos em jogos é a Inteligência Artificial - IA. Existem diversos estudos sobre o jogo de dominó, suas variações e a resolução dele por meio de técnicas de IA. No presente trabalho buscou-se a implementação do jogo de dominó de duas pontas, utilizando como ideia a estratégia do MinMax, com algumas limitações. Para a implementação utilizou-se da linguagem de programação Java. O trabalho foi desenvolvido na disciplina de Ciências da Computação do Instituto Federal Catarinense *Campus* Rio do Sul. O Jogo de dominó foi escolhido devido sua complexidade e dificuldade de previsão de jogadas adversárias e por afinidade e interesse do autor com o assunto.

2 - REVISÃO BIBLIOGRÁFICA

2.1 Jogo de dominó

O jogo de dominó como conhecido atualmente tem origem bem controversa alguns autores afirmam o seu surgimento na china, por volta de 234 a 181 A.C. com o herói Hung Ming, com a intenção de distrair seus soldados. Em meados do século XVIII teve força na Europa e se espalhou pelo mundo inteiro. Se espalhou bastante entre os religiosos da época pois sempre que conseguiam realizar determinada combinação exclamavam em latim *Benedicamus Domino* “Louvemos ao Senhor” (COSTA, 2016). Ele é constituído na sua versão mais conhecida como duplo 6, constituído por 28 peças distintas, cada peça possui um valor de 0 a 6, conforme a Figura 1. O dominó apesar de o duplo 6 ser mais conhecido, existem também outras versões bem populares como o ponta de 5 entre outros utilizando regras diferenciadas de acordo com a quantidade de peças e a forma de jogar.

Figura 1 – 28 Peças de um dominó no formato duplo-6 ordenado

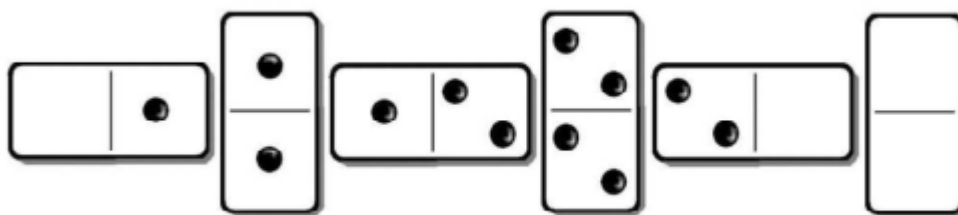


Fonte: Pinto (2018, p.15)

As regras do jogo são simples, segue inicialmente as peças todas viradas com os números para baixo constituindo o monte na mesa, em seguida são embaralhadas. Após o monte embaralhado, cada jogador retira 7 peças constituindo assim o monte na mesa. Após a distribuição sai o jogador que tiver a maior carreta (ou denominados duplos) ou peça. Sempre encaixando no lado nas peças na mesa conforme a numeração correspondente. (COSTA, 2016).

A Figura 2 mostra uma configuração de encaixe de peças.

Figura 2 – Exemplo de configuração de encaixe de peças do dominó



Fonte: Costa (2016, p.7)

Após um jogador jogar, é a vez do outro, caso o jogador não tenha a peça de encaixe ele retorna para o que sobrou do monte de peças na mesa, também conhecido como cemitério, retirando do cemitério até encontrar a peça necessária para jogar ou não ter mais peças no cemitério. Caso não tenha mais peças no cemitério o jogador passa a vez para o outro jogador jogar e a vitória é do jogador que não possuir mais peças na mão (Pinto, 2018). Um exemplo

de passe podemos ver na Figura 3. Geralmente o objetivo do dominó é somar os pontos de cada partida buscando obter 100 e duzentos pontos (SILVA, 2015).

Figura 3 – Configuração de um passe do jogador



Fonte: Silva (2015, p.31)

2.2 MinMax, Métodos de Busca e Estratégias Heurísticas aplicadas no Jogo de Dominó

Existem diversas estratégias de resoluções de problemas de jogos dentro da área de Inteligência Artificial. O MinMax como conhecido segundo Silva (2015, p.34): “é um método empregado para minimizar a perda máxima possível. Na área de jogos, para a criação de agentes inteligentes, muitos autores utilizam este algoritmo.” Para o funcionamento do mesmo é construído durante a busca ou antes uma árvore de estados, onde cada estado possui um valor heurístico. Esse valor heurístico é utilizado ou pelo Max ou pelo Min para maximizar ou minimizar as possibilidades de jogadas e vitórias do oponente. Nesse sentido o “primeiro jogador, chamado jogador MAX, tentará maximizar a sua pontuação, enquanto o segundo jogador, conhecido como jogador MIN, tenta minimizar a pontuação do oponente (SILVA, 2015, p. 34).”

Silva (2015) em seu trabalho aborda o dominó de 4 pontas, e também argumenta que nem sempre é possível realizar toda a expansão da árvore de nodos de um jogo, como no

exemplo o dominó. Nesses casos o mesmo abordou uma tentativa de estratégia de Expectiminimax, onde realiza a busca em profundidade até um certo nível da árvore, com mais elementos no algoritmo, contendo os nós Min, Max e chance, também conhecido como nó para valores aleatórios. O autor também ressalta em uma abordagem completa de testes a eficiência do Expectiminimax. Pinto (2018) em seu trabalho também comenta em seu trabalho sobre um método de poda conhecido como podagem alpha-beta, que visa por meio dos nós (conhecimento a priori) podar a expansão de determinados galhos da árvore, método muito útil quando utilizado juntamente com o algoritmo MinMax. E por fim o último autor Costa (2016) em seu trabalho propõe para o jogo de dominó utilizando o conceito para os oponentes de agentes, uma estratégia de defensiva e ofensiva, no fim utiliza as duas combinadas. As estratégias são parecidas com o MinMax, contudo utilizam de regras para realizar as mesmas. A ideia inicial de implementação do presente trabalho se baseou no trabalho de Costa (2016). As estratégias ofensivas e defensivas utilizadas pelo autor podem ser vistas na Figura 4.

Figura 4 – Estratégias de Heurísticas propostas por Costa (2016) para jogo de dominó

Por defensivo, entende-se que o jogador deve:

- **Posicionar as carroças primeiro. Elas têm o mesmo número em ambos os lados, limitando opções de jogadas.**
- **Manter uma variedade de pedras pelo máximo de tempo possível**
- **Posicionar as pedras mais altas primeiro. No caso de a cadeia ser bloqueada, o jogador terá uma mão leve.**

Por ofensivo, entende-se que o jogador deve:

- **Perceber os pontos fracos do oponente lembrando dos naipes que ele não possui.**
- **Encontrar a extremidade mais vulnerável através da contagem dos naipes das próprias pedras mais as da cadeia.**
- **Tornar as extremidades da cadeia iguais, especialmente se foram um ponto fraco do oponente. O adversário terá menos oportunidades de posicionar suas pedras.**

Fonte: Costa (2016, p.15-16)

Existem diversas outras estratégias no trabalho de Costa (2016) a combinação das duas estratégias resultaram melhores resultados.

3 - DESENVOLVIMENTO

No trabalho inicialmente foi implementado o jogo de dominó de duas pontas duplo-6 com dois personagens computacionais sem interação com o usuário, como se o computador estivesse jogando contra ele mesmo, com a escolha das peças sequencialmente conforme disponível na mão do jogador. Em seguida foi implementado a possibilidade de interação com um jogador humano e finalmente acrescentado uma heurística de IA para o computador saber qual a melhor peça para se jogar. Para isso implementou-se uma classe que representasse uma **Peca** que representa uma peça de dominó conforme Anexo 1, uma classe **MontedePecas** (Anexo 2) que representa o monte de peças do dominó para a distribuição das peças dos dois jogadores. A classe **MontedePecas** possui três métodos a se destacar, um que inicializa o monte com todas as peças do jogo de um dominó normal de 28 peças conforme a Tabela 1, utilizando da estrutura de ArrayList de peças para armazenar o monte de peças.

Tabela 1 – Método **iniciaMonte** da Classe **MontedePecas**

```
1. - private ArrayList<Peca> monte;
2. - private Peca p;
3. -
4. - public MontedePecas() {
5. -     this.inicializaMonte();
6. - }
7. -
8. - public void inicializaMonte() {
9. -     monte = new ArrayList();
10. -     for (int i = 0; i <= 6; i++) {
11. -         for (int j = 0 + i; j <= 6; j++) {
12. -             p = new Peca(i, j);
13. -             monte.add(p);
14. -         }
15. -     }
16. - }
```

Fonte: Elaboração autor.

Outra parte importante da classe **MontedePecas** a se destacar é a função **compraPeca**, responsável por retornar uma peça do monte caso jogador não tenha nenhuma peça para jogar, conforme linha 5 até 17 da Tabela 2. E por último o método **distribuiPecas** que é responsável por retornar um arrayList de 7 peças para cada jogador, conforme código das linhas 19 a 32 da Tabela 2. Importante ainda destacar que no momento da compra de uma peça, a peça é escolhida e retornada de forma aleatória.

Tabela 2 – Métodos **compraPeca** e **distribuiPecas** da Classe **MontedePecas**

```

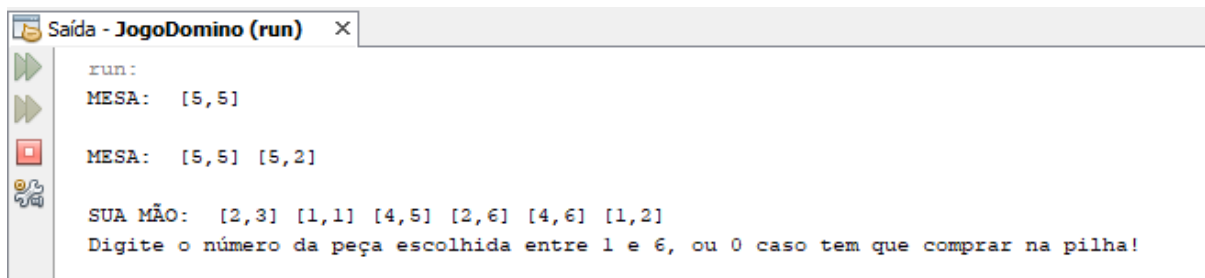
1. -  /**
2. -   * Retorna peça aleatória do monte removendo ela
3. -   * @return Peca aleatória ou null quando o monte estiver vazio
4. -   */
5. -  public Peca compraPeca() {
6. -      Peca p = new Peca(0,0);
7. -      if(this.getMonte().size()!=0){
8. -          int j = (int) ((int) (monte.size()-1)*Math.random());
9. -          if (this.getMonte().get(j) != null) {
10. -              p = (Peca) this.getMonte().get(j);
11. -              this.monte.remove(j);
12. -          }
13. -          return p;
14. -      }else{
15. -          return null;
16. -      }
17. -  }
18. -
19. -  public ArrayList<Peca> distribuiPecas() {
20. -      Peca p;
21. -      int j;
22. -      ArrayList<Peca> ap = new ArrayList();
23. -      for (int i = 0; ap.size() < 7; i++) {
24. -          j = (int) ((int) (monte.size())*Math.random());
25. -          if (this.getMonte().get(j) != null) {
26. -              p = (Peca) this.getMonte().get(j);
27. -              ap.add(p);
28. -              this.monte.remove(j);
29. -          }
30. -      }
31. -      return ap;
32. -  }

```

Fonte: Elaboração autor.

Outra Classe desenvolvida foi a classe **Jogo** responsável por gerenciar todo o jogo de dominó, chamando métodos responsáveis para o funcionamento bem como a apresentação dos dados em console. Desta forma o jogo inicializa chamando a função **playGame** da classe **Jogo**, escolhendo a peça de maior dobre na mão de um dos jogadores, essa peça é jogada automaticamente, passando a vez para o outro jogador jogar. Inicialmente na classe **Jogo** após a primeira peça ser jogada a mesa já é imprimida em console, conforme Figura 5, onde a mesa é imprimida e a primeira peça é removida da mão do jogador humano pois possui o maior dobre.

Figura 5 – Jogo de dominó iniciando pelo jogador seguido do computador



```
run:
MESA:  [5,5]

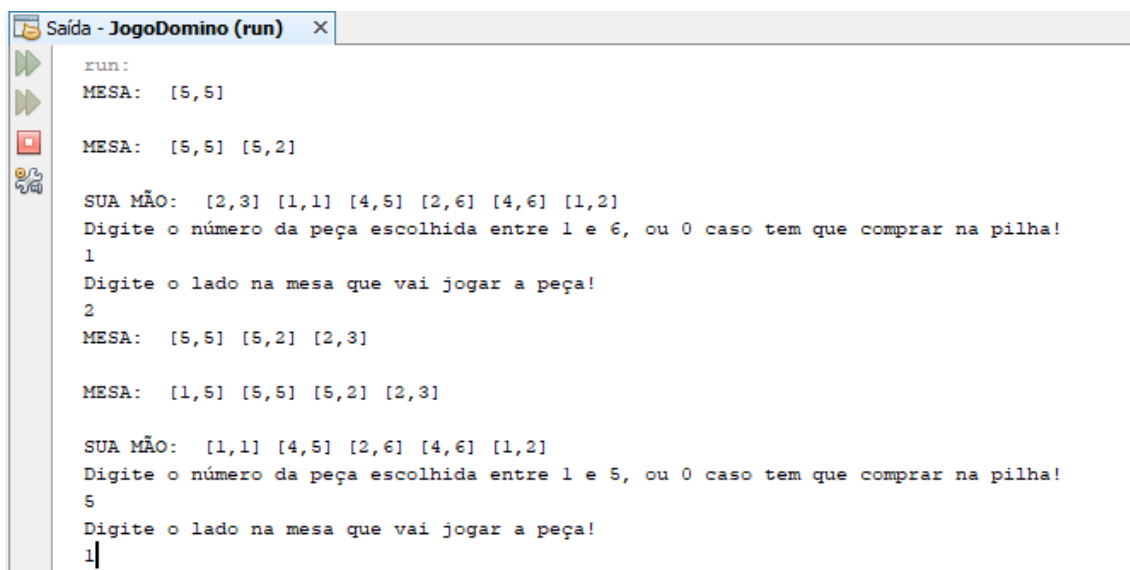
MESA:  [5,5] [5,2]

SUA MÃO:  [2,3] [1,1] [4,5] [2,6] [4,6] [1,2]
Digite o número da peça escolhida entre 1 e 6, ou 0 caso tem que comprar na pilha!
```

Fonte: Elaboração autor.

Após assim a primeira jogada, o programa entra em um laço while que só encerra caso o jogo tenha sido fechado ou a mesa e a mão de um jogador esteja vazia. Retornando no final da função **playGame** a soma dos pontos da partida e o vencedor. Inicialmente foi implementado o jogo com apenas uma partida, contudo sendo expansível para a implementação de mais partidas. Além disso na função **playGame** também realiza a distribuição das peças nos lados da mesa conforme são escolhidas pelos jogadores. Conforme o jogo anda ao jogador humano é solicitado que digite uma peça entre 0 e 7, ou de acordo com a quantidade de peças que possui na mão e solicita também o lado na mesa que deseja jogar, caso a peça seja inválida ou os valores incorretos retorna mensagem de erro e solicita jogar novamente. Podemos ver o exemplo do jogo na Figura 6 antes de o jogador teclar enter para confirmar a jogada.

Figura 6 – Jogo de dominó depois de o jogador escolher a peça e o lado da mesa



```
run:
MESA:  [5,5]

MESA:  [5,5] [5,2]

SUA MÃO:  [2,3] [1,1] [4,5] [2,6] [4,6] [1,2]
Digite o número da peça escolhida entre 1 e 6, ou 0 caso tem que comprar na pilha!
1
Digite o lado na mesa que vai jogar a peça!
2
MESA:  [5,5] [5,2] [2,3]

MESA:  [1,5] [5,5] [5,2] [2,3]

SUA MÃO:  [1,1] [4,5] [2,6] [4,6] [1,2]
Digite o número da peça escolhida entre 1 e 5, ou 0 caso tem que comprar na pilha!
5
Digite o lado na mesa que vai jogar a peça!
1|
```

Fonte: Elaboração autor.

A função **retornaPecaEncaixe** como o próprio nome sugere, retorna a peça de encaixe de acordo com as peças escolhidas pelo jogador ou computador. No caso do computador para

a escolha de peças foi inicialmente implementado apenas uma escolha normal das peças disponíveis na mão, o método comentado **retornaPecaAleatóriaEncaixe**, porém modificado e implementado uma árvore de nodos contendo as peças e os seus respectivos valores heurísticos, estes que auxiliam na tomada de decisão de qual peça ser escolhida. A heurística utilizada para escolha da melhor peça foi a quantidade de possibilidades de jogadas que o jogador humano pode realizar caso o computador jogue uma determinada peça. Para isso buscou-se a montagem de uma árvore de possibilidades, a qual ainda está em fase de desenvolvimento. A árvore de possibilidades apenas foi montada até em 2 níveis, sendo que para o segundo nível é atribuído um valor heurístico ao nó, o mesmo representa uma peça do dominó. Para a realização da montagem da árvore de possibilidades e o método heurístico, foi criada duas classes, **BuscaIA** a qual possui a função **IA** que retorna a peça escolhida, percorrendo a árvore e verificando o menor valor heurístico, conforme heurística aplicada, e chama a classe **Arvore** para a construção da árvore e atribuição dos valores heurísticos conforme linha 3 da Tabela 3. Podemos ver o método **IA** da classe **BuscaIA** que retorna a peça conforme a heurística aplicada na Tabela 3, percorrendo os nodos e verificando a peça com menor valor heurístico, o que implica que o jogador humano tem um número menor de peças possíveis a ser jogadas.

Tabela 3 – Método **IA** da classe **BuscaIA**

```

1. - Public Peca IA(){
2. -     Arvore a = new Arvore();
3. -     a.constroiArvore(p1, p2, m, j, 0, 3);
4. -     ArrayList<Arvore.Nodo> Ln = a.arvore.get(0).Ln;
5. -     int menHeur = 100;
6. -     Peca p = new Peca(8,8);
7. -     if(Ln.size()>0){
8. -         for (int i = 0; i < Ln.size(); i++) {
9. -             if(Ln.get(i).heuristicaMinMax<menHeur){
10. -                 p = new Peca(Ln.get(i).ponta1, Ln.get(i).ponta2);
11. -                 this.ladoEncaixe = Ln.get(i).lado;
12. -                 menHeur = Ln.get(i).heuristicaMinMax;
13. -             }
14. -         }
15. -         return p;
16. -     }
17. -     return null;
18. - }

```

Fonte: Elaboração autor.

E a classe **Arvore** por sua vez possui dois métodos um **constroiArvore** que recebe as pontas da mesa, bem como a lista de peças visíveis e encaixadas já na mesa a mão de peças do

computador, o nível inicial e o nível de profundidade da árvore. Em seguida entra em um loop expandido os nodos de possibilidade de uma peça a ser jogada pelo computador partindo das pontas na mesa disponíveis. Para cada peça possível de ser encaixada na mesa pelo computador, é calculada a quantidade de peças que o próximo jogador supostamente pode ter, descontando as peças na mesa e na mão do computador, retornando um valor heurístico, sendo que quanto menor, menor a possibilidades de jogadas que o jogador humano pode realizar, sendo a peça de menor heurística utilizada pelo computador na jogada, ideia básica do MinMax.

Durante o jogo são também retornadas mensagens, peça errada escolhida escolha outra peça, após a finalização do jogo, mensagens de empate, jogo fechado, e vitória e a quantidade de pontos realizados pelo jogador. Conforme código na Tabela 4.

Tabela 4 – Trecho que realiza o cálculo e mostra as mensagens de resultado classe **Jogo**

1. -	int soma1 = 0;
2. -	int soma2 = 0;
3. -	for (int i = 0; i < j1.size(); i++) {
4. -	soma1 = soma1 + j1.get(i).getPonta1() + j1.get(i).getPonta2();
5. -	}
6. -	for (int i = 0; i < j2.size(); i++) {
7. -	soma2 = soma2 + j2.get(i).getPonta1() + j2.get(i).getPonta2();
8. -	}
9. -	if (!j1.isEmpty() && !j2.isEmpty()) {
10. -	if (soma1 == soma2) {
11. -	System.out.println("Jogo fechado!");
12. -	System.out.println("Empate!");
13. -	} else {
14. -	if (soma1 < soma2) {
15. -	System.out.println("Jogo fechado!");
16. -	System.out.println("Você ganhou! Fez " + soma2 + " pontos!");
17. -	} else {
18. -	System.out.println("Jogo fechado!");
19. -	System.out.println("Jogador 2 ganhou! Fez " + soma1 + " pontos!");
20. -	}
21. -	}
22. -	} else {
23. -	if (j1.isEmpty()) {
24. -	System.out.println("Você ganhou! Fez " + soma2 + " pontos!");
25. -	} else {
26. -	System.out.println("Jogador 2 ganhou! Fez " + soma1 + " pontos!");
27. -	}
28. -	}
29. -	System.out.println(j1);
30. -	System.out.println(j2);
31. -	System.out.println("Fim de Jogo!");

Fonte: Elaboração autor.

As classes de **Jogo**, **BuscaIA** e **Arvore** não foram anexadas devido ao tamanho dos arquivos, mas estão disponibilizados no link no Github do autor: <<https://github.com/CleverttonHoffmann/Domino----MinMax---IA---JAVA.git>>.

4 - CONCLUSÃO

Um trabalho não termina por si só, um conhecimento sobre algum assunto é necessário ser explorado e reelaborado, repensado. De forma similar faz-se necessário a revisão de qualquer conhecimento em qualquer área até mesmo para não cair em esquecimento. O presente trabalho apesar de não mostrar comparativos em relação a técnicas de IA, possibilitou aprendizado de algumas técnicas diferentes como Expectiminimax, o funcionamento de técnicas como MinMax e buscas heurísticas e as diferentes visões e resoluções de problemas envolvendo o jogo de dominó. Assim o código do trabalho estará sendo melhorado, com o acréscimo de algumas heurísticas abordadas na fundamentação teórica e continuará disponível no Github do autor, para eventuais consultas. Uma possível alteração não no presente momento mas em momentos futuros a inclusão de interface gráfica com a representação do dominó em figuras.

5 - REFERÊNCIAS

COSTA, J. A. S. **Inteligência Artificial em jogos de tabuleiro: proposição de uma Heurística para o Jogo de Dominós**. 2016. 23 f. Trabalho de Conclusão de Curso (Graduação em Computação) – Centro de Ciências Exatas e Sociais Aplicadas, Universidade Estadual da Paraíba, Patos, 2016. Disponível em: <<http://dspace.bc.uepb.edu.br/jspui/bitstream/123456789/10078/4/PDF%20-%20Jos%c3%a9%20Aldo%20Silva%20da%20Costa.pdf>> Acesso em 22 de dez. 2020.

PINTO, Ivan de Jesus Pereira. **Inteligência artificial aplicada ao jogo de dominó**. Monografia (Graduação em Computação), Universidade Federal do Maranhão. São Luis, Maranhão, 2018. Disponível em: <<http://monografias.ufma.br/jspui/handle/123456789/3510>> Acesso em 22 de dez. 2020.

SILVA, Endrews Sznyder Souza da. **Proposta de um agente para o jogo de dominó de 4 pontas utilizando o algoritmo EXPECTIMINIMAX**. 2015. 91 f. Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal do Amazonas, Manaus, 2015. Disponível em: <<https://tede.ufam.edu.br/bitstream/tede/5010/2/Disserta%C3%A7%C3%A3o%20-%20Endrews%20Sznyder%20Souza%20da%20Silva.pdf>> Acesso em 22 de dez. 2020.

6 - ANEXOS

Anexo 1 – Código classe **Peca** – Representando uma peça de Dominó

```
32. - /**
33. -  * Classe responsável por implementar a estrutura de uma peça de dominó
34. -  *
35. -  * @author Cleverton
36. -  */
37. - public class Peca {
38. -
39. -     private int ponta1;
40. -     private int ponta2;
41. -
42. -     public Peca(int p1, int p2) {
43. -         this.ponta1 = p1;
44. -         this.ponta2 = p2;
45. -     }
46. -
47. -     public int getPonta1() {
48. -         return ponta1;
49. -     }
50. -
51. -     public void setPonta1(int ponta1) {
52. -         this.ponta1 = ponta1;
53. -     }
54. -
55. -     public int getPonta2() {
56. -         return ponta2;
57. -     }
58. -
59. -     public void setPonta2(int ponta2) {
60. -         this.ponta2 = ponta2;
61. -     }
62. -
63. -     @Override
64. -     public String toString() {
65. -         return "Peca{ " + "ponta1=" + ponta1 + ", ponta2=" + ponta2 + '}';
66. -     }
67. -
68. - }
```

Fonte: Elaboração Autor

Anexo 2 – Código classe **MontedePecas**

1. -	import java.util.ArrayList;
2. -	
3. -	public class MontedePecas {
4. -	
5. -	private ArrayList<Peca> monte;
6. -	private Peca p;
7. -	
8. -	public MontedePecas() {
9. -	this.inicializaMonte();
10. -	}
11. -	
12. -	/**
13. -	* Método responsável por iniciar monte de peças do dominó
14. -	*/
15. -	public void inicializaMonte() {
16. -	monte = new ArrayList();
17. -	for (int i = 0; i <= 6; i++) {
18. -	for (int j = 0 + i; j <= 6; j++) {
19. -	p = new Peca(i, j);
20. -	monte.add(p);
21. -	}
22. -	}
23. -	}
24. -	
25. -	/**
26. -	* Retorna peça aleatória do monte removendo ela
27. -	* @return Peca aleatória ou null quando o monte estiver vazio
28. -	*/
29. -	public Peca compraPeca() {
30. -	Peca p = new Peca(0,0);
31. -	if(this.getMonte().size()!=0){
32. -	int j = (int) ((int) (monte.size()-1)*Math.random());
33. -	if (this.getMonte().get(j) != null) {
34. -	p = (Peca) this.getMonte().get(j);
35. -	this.monte.remove(j);
36. -	}
37. -	return p;
38. -	}else{
39. -	return null;
40. -	}
41. -	}
42. -	
43. -	public ArrayList<Peca> distribuiPecas() {
44. -	Peca p;
45. -	int j;
46. -	ArrayList<Peca> ap = new ArrayList();
47. -	for (int i = 0; ap.size() < 7; i++) {
48. -	j = (int) ((int) (monte.size()-1)*Math.random());

49. -	if (this.getMonte().get(j) != null) {
50. -	p = (Peca) this.getMonte().get(j);
51. -	ap.add(p);
52. -	this.monte.remove(j);
53. -	}
54. -	}
55. -	return ap;
56. -	}
57. -	
58. -	public ArrayList getMonte() {
59. -	return monte;
60. -	}
61. -	
62. -	public void setMonte(ArrayList monte) {
63. -	this.monte = monte;
64. -	}
65. -	}

Fonte: Elaboração Autor