INSTITUTO TECNOLÓGICO DE AERONÁUTICA - ITA

DIVISÃO DE CIÊNCIA DA COMPUTAÇÃO

CTC-17 – Inteligência Artificial

Projeto I - Buscas

- **Menor caminho**
- **Jogo da Velha**

ALUNO

Felipe Tuyama de Faria Barbosa ftuyama@gmail.com

PROFESSORA

Paulo André Lima de Castro pauloac@ita.br

São José dos Campos, 01 de Setembro de 2016

1. INTRODUÇÃO

O objetivo deste projeto é implementar algoritmos de inteligência artificial envolvendo busca. Em um primeiro momento, é explorado o problema do menor caminho entre duas cidades. Em seguida, temos a criação de um agente para o clássico jogo da velha.

O simulador foi implementado na linguagem de alto nível Python 2.7, devido à facilidade de prototipação conferida pela linguagem. Foi utilizada a biblioteca graphics para a criação da GUI, além de bibliotecas adicionais de estruturas de dados.

2. DESCRIÇÃO

As duas aplicações desenvolvidas contam com uma GUI (interface gráfica) a fim de mostrar a situação/solução do problema. Elas podem ser executadas diretamente a partir do arquivo .py principal de projeto.

Para o problema de determinar o menor caminho entre as duas cidades, são explorados os algoritmos greedy e A*, dadas as coordenadas de cada cidade e as ligações entre elas. As rotas entre cidades foram aqui consideradas linhas retas, assim como a heurística utilizada para a tomada de decisão nestes algoritmos (distância em linha reta até a cidade objetivo).

Em seguida, explora-se um cenário de busca competitiva através da criação de um agente para o Jogo da Velha, usando o algoritmo Minimax combinado a heurísticas de jogo e podas de árvore de busca para otimizações. A heurística usada foi a mesma apresentada em sala de aula, o número de possibilidades de vitória para um dado jogador, sendo que a busca ocorre sempre com uma profundidade máxima de jogadas à frente, evitando expandir demasiadamente a árvore de busca.

3. IMPLEMENTAÇÃO

A implementação de ambos os problemas de busca decorreu de forma similar, em termos de arquitetura. Houve a implementação das seguintes funções:

- Função de representação gráfica do problema (GUI) – Output.
- Função para capturar e validar os Inputs do problema.
- Função principal com os parâmetros do problema e da AI.
- Funções de inteligência artificial e tomada de decisões.

3.1 Implementação do melhor trajeto

Este problema tem como input um arquivo .csv contendo em cada linha a informação de uma cidade, em que cada linha tem um significado:

1. Id da cidade
2. Posição x e y no mapa
3. As demais colunas são Ids das cidades adjacentes

Essas entradas são transformadas para uma lista de listas em python (uma lista de elementos cidade, em que cada elemento cidade é uma linha da tabela). Suas coordenadas sofrem adaptação de escala para representação na tela.

	A	B	C	D	E	F	G	H	I	J
1	1	30,133	57,633	2	6					
2	2	30,167	57,1	1	6	7				
3	3	30,233	57,583	5	7					
4	4	30,25	56,85	6	7					
5	5	30,25	56,95	3	7	10				
6	6	30,25	57,583	1	2	4	7	9	10	
7	7	30,3	56,967	2	3	4	5	6	9	11
8	8	30,317	56,817	11	12					
9	9	30,4	56,467	6	7	11	14			
10	10	30,4	56,783	5	6	13	17	18		
11	11	30,433	57,433	7	8	9	12	20		
12	12	30,467	56,55	8	11	13	17			
13	13	30,483	56,517	10	12	15	16			
14	14	30,5	56,45	9	15	16				

Figura 1 – Input (mapa) do problema do melhor trajeto.

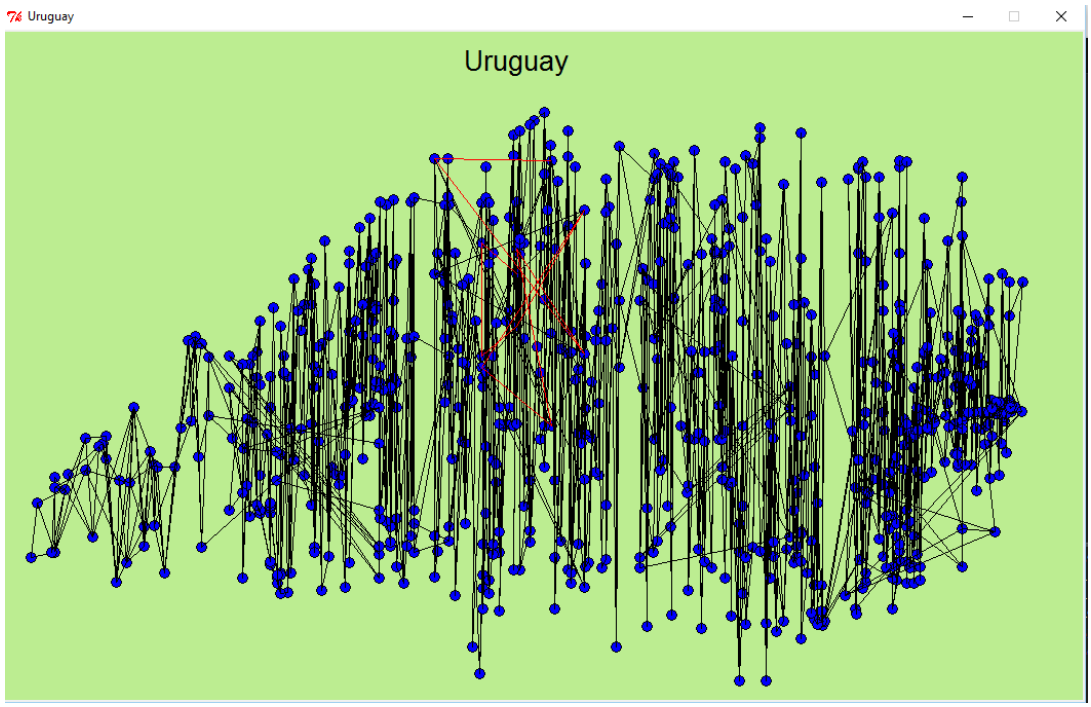


Figura 2 – Output (GUI) do problema do melhor trajeto.

A linha vermelha simboliza a solução encontrada para o problema.

Tanto o algoritmo greedy como A* foram implementados usando a fila de prioridade do Python (para começar a expandir os nós mais próximos do objetivo primeito), basicamente seguindo a ideia do seguinte pseudo-código (ambas compartilham de mesma ideia, A* é uma variante de algoritmo guloso, usando heurística específica):

```
while !queue.empty():
      city_analysed = queue.pop()
      for city in city_analysed:
            if city == destination:
                  return path
            if city.visited:
                  pass
            estimate = . . .
            queue.append(estimate, city)
```

Ideia do algoritmo guloso. A heurística adotada define as diferenças.

A função heurística adotada pelo algoritmo greedy para a busca informada é a "city_destination_distance" - distância em linha reta do nó adjacente (que está sendo expandido) até o objetivo final.

```
city_destination_distance = distance(id_city, id_destination)
queue.put((city_destination_distance, id_city, visited, path))
```

Heurística adotada pelo algoritmo greedy.

Enquanto a função heurística do algoritmo A* evita expandir nós que já ficaram caros, adicionando à função heurística a componente "current_city_distance" – distância total percorrida até a cidade atual analisada. A componente heurística "city_destination_distance" também se faz presente neste algoritmo.

```
# Distância da cidade atual até a adjacente
current_city_distance = total_distance(solution)
# Estima distância da cidade adjacente até o destino
city_destination_distance = distance(id_city, id_destination)
# Estimativa da distância total através da cidade adjacente
estimated_current_distance =\
     current_city_distance + city_destination_distance
queue.put((estimated_current_distance, id_city, visited, path))
```

Heurística adotada pelo algoritmo A*.

3.2 Implementação do jogo da velha

Este problema tem como input as jogadas do jogador humano, que são realizadas através de cliques na interface gráfica gerada, os quais são validados (detecta o quadrado desejado e se ele está disponível). O output são as jogadas respostas da AI, representadas também nesta interface gráfica.

Além da GUI, o jogo conta com uma variável chamada tabuleiro que facilita as análises de jogada e da própria AI. O loop de jogo basicamente decorre esperando jogadas, seja do jogador ou da AI, usando funções auxiliares para conferir o resultado de jogo a cada mudança do tabuleiro.

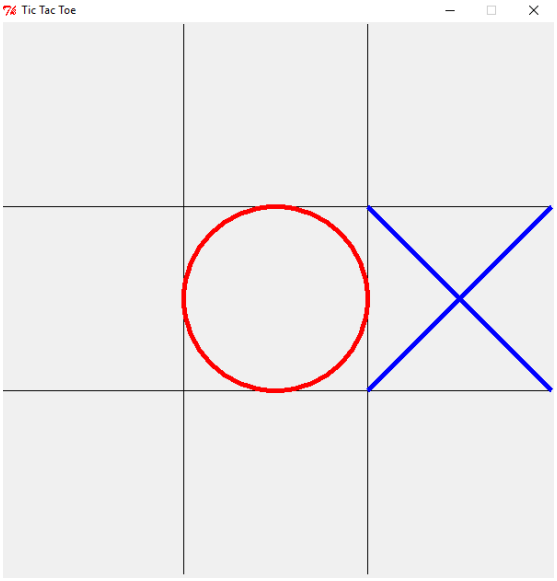


Figura 3 – GUI elaborada para o jogo da velha.

O ciclo de jogo pode ser descrito pelo seguinte pseudo-código (representado de forma didática e simplificada):

```
create_board()
for i in range(squares^2):
      if player == ‘x’:
            tic_tac_toe(input())
      if player == ‘o’:
            tic_tac_toe(AI())
      check_winner()
      draw_changes()
draw()
```

Ideia simplificada do algoritmo de loop de jogo.

O algoritmo do agente do jogo da velha inicialmente foi implementado de forma bruta, percorrendo toda a árvore de jogo até as suas folhas, contabilizando vitórias e derrotas do agente. A execução era lenta e os resultados ineficientes, mesmo usando a abordagem do algoritmo minimax (escolher o máximo dentre os mínimos – algoritmo usado em buscas competitivas, em que o agente toma a melhor decisão considerando que o outro jogador tentará tomar a melhor decisão também).

Para determinar a árvore de busca, o agente percorre as posições do tabuleiro identificando as jogadas válidas, expandindo-as usando recursão (até uma folha em que ocorra empate (velha) ou até a vitória de um dos jogadores). Poder-se-ia utilizar uma pilha para tal, mas por simplicidade adotou-se a pilha de chamadas recursivas como ferramenta de DFS (busca em profundidade na árvore):

```
def minimax_o(board, x, y):
    u"""Minimax para jogada de o."""
    results = []
    moves = []
    board = copy.deepcopy(board)
    board[x][y] = 'x'
    # Varre tabuleiro procurando jogadas
    for i in range(0, squares):
        for j in range(0, squares):
            if board[i][j] == '':
                point = points(board, i, j, 'o')
                if point == 0:
                    # Continua buscando nós
                    value = minimax_x(board, i, j)[0]
                    results.append(value)
                else:
                    # Nó. Ponto. Voltar
                    results.append(1)
                moves.append((i, j))

    # Caso não haja mais jogadas
    if len(results) == 0:
        return (0, None)
    # Calcula a jogada com maior pontuação (Max)
    max_id = results.index(max(results))
    poda[n_plays(board)] = max(poda[n_plays(board)],
results[max_id])
    return (results[max_id], moves[max_id])
```

Algoritmo recursivo de Minimax sem aplicação de heurísticas.

Assim, foi implementada a função de heurística para limitar o número de jogadas analisadas à frente da atual (limitando a profundidade da árvore de busca):

```
def minimax(board, x, y, player):
    global depth, max_depth, heuristic, poda
    # Características da AI
    depth = n_plays(board)  # Profundidade começa na jogada atual
    max_depth = 4           # Máxima profundidade de análise
    poda = []               # Ramo da árvore analisado
    heuristic = True        # Uso de heurística ou não
```

Parâmetros do agente AI implementado.

A cada chamada recursiva, é verificado o número de peças no tabuleiro, limitando o número de jogadas à frente analisadas:

```
   # Verifica profundidade máxima
    if heuristic:
        if n_plays(board) - depth >= max_depth:
            return (heuristics(board, 'x'), None)
```

Heurística adotada para limitar profundidade.

Em seguida, foi implementado a poda das árvores de busca que contrariem o princípio do algoritmo Minimax: Alpha não pode diminuir e Beta não pode aumentar no decorrer de uma análise.

```
                if point == 0:
                    # Continua buscando nós
                    value = minimax_x(board, i, j)[0]
                    # Alpha não pode aumentar
                    if heuristic and value > poda[n_plays(board)]:
                        # Poda da árvore de busca
                        return (0, None)
                    else:
                        results.append(value)
                else:
                    # Nó. Ponto. Voltar
                    results.append(1)
                moves.append((i, j))
```

Poda da árvore de busca, reduzindo nós expandidos.

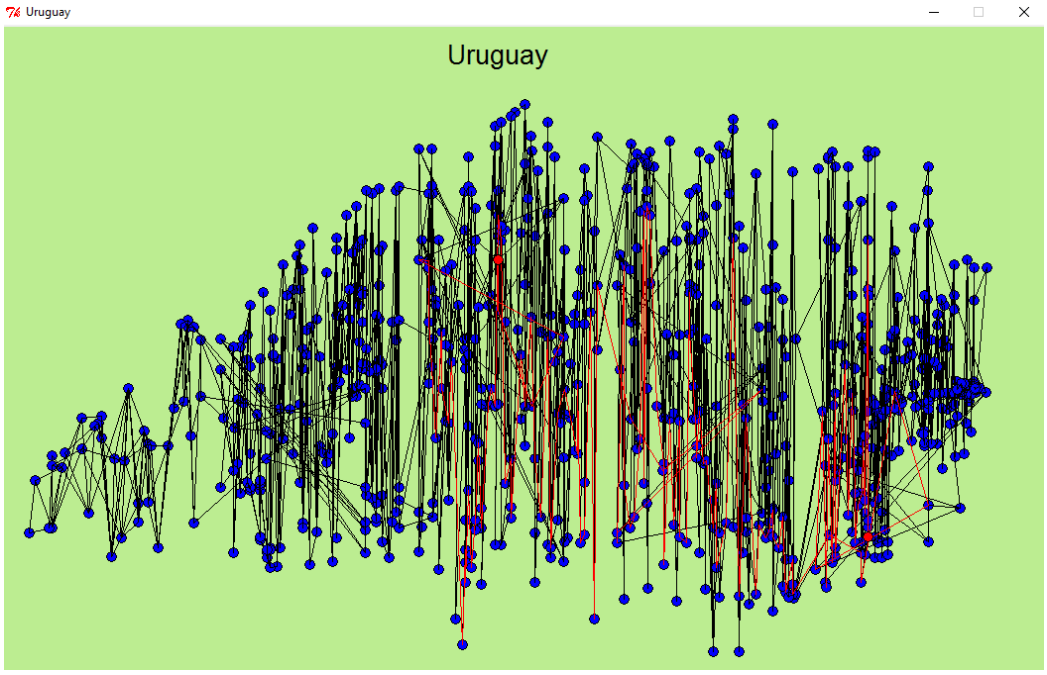
Após as otimizações realizadas, o tempo de execução melhorou significativamente (de aproximadamente 5s para 0.5s), também produzindo resultados de jogo satisfatórios. Admito aqui que não implementei um sistema de simetria para o jogo, o que poderia otimizar ainda mais os resultados em questão de performance.

4. RESULTADOS

Nesta seção apresento os resultados encontrados decorrentes da execução dos algoritmos, avaliando a consistência dos outputs gerados.

4.1 Resultados do menor caminho

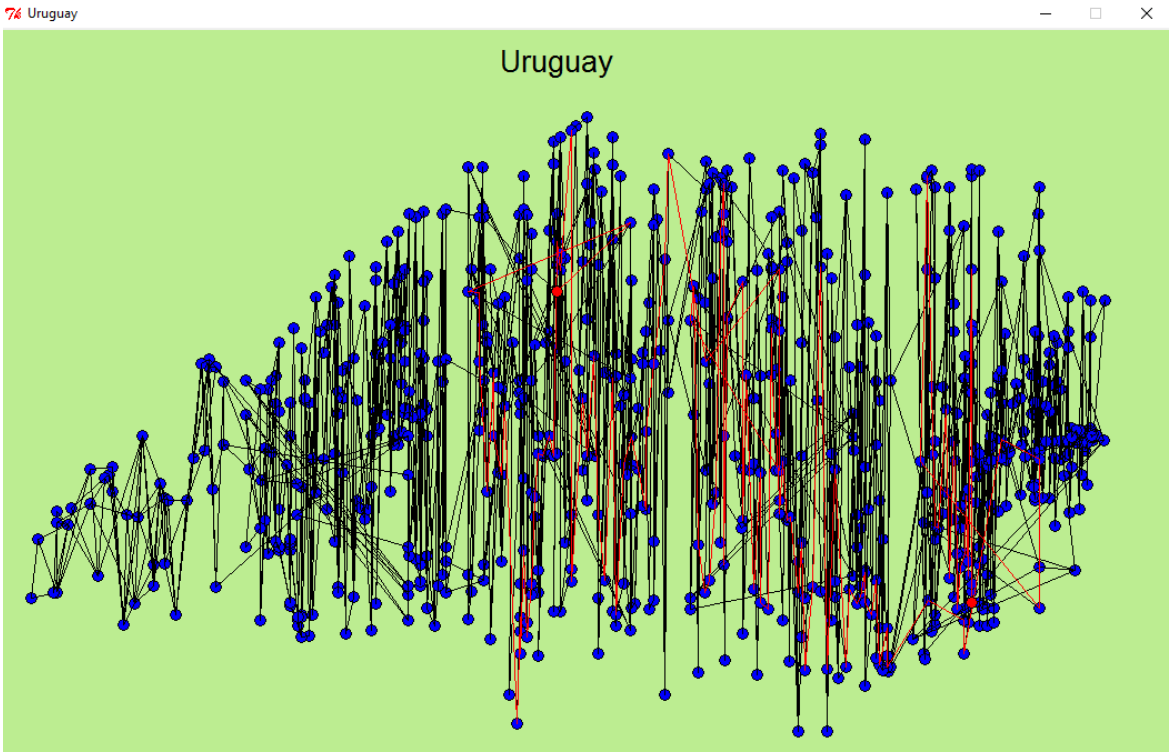
Para o algoritmo greedy, foi obtido o seguinte resultado da figura e output no console, mostrados logo abaixo:




```
 ['202', '203', '205', '207', '211', '217', '219', '224', '227',
'230', '231', '236', '239', '242', '246', '248', '253', '257',
'262', '265', '270', '274', '277', '280', '283', '288', '292',
'294', '296', '300', '305', '310', '312', '317', '321', '325',
'330', '335', '340', '345', '346', '348', '353', '355', '357',
'359', '364', '365', '371', '374', '375', '380', '381', '384',
'387', '392', '394', '395', '399', '404', '408', '411', '412',
'416', '419', '422', '424', '426', '431', '432', '435', '438',
'439', '442', '447', '451', '454', '459', '460', '465', '469',
'473', '478', '483', '485', '490', '494', '499', '500', '504',
'508', '513', '517', '519', '523', '526', '527', '530', '535',
'539', '541', '544', '548', '551', '555', '558', '559', '563',
'567', '570', '575', '576', '580', '584', '586', '590', '595',
'601']
Menor distância = 121.826004347
[Finished in 0.5s]
```

Resultado 1 – Resultado do algoritmo greedy.

Para o algoritmo A*, foi obtido o seguinte resultado da figura e output no console, mostrados logo abaixo. Note que a solução encontra é melhor que a do algoritmo greedy, sendo que a primeira divergência de caminhos ocorre apenas na 13ª cidade escolhida. Interessante reparar que os tempos de execução são muito similares (considerando apenas tempo de leitura e execução do algoritmo, sem GUI).




```
 ['202', '206', '211', '217', '219', '224', '227', '230', '231',
'236', '239', '242', '246', '248', '253', '257', '262', '265',
'268', '272', '277', '280', '283', '288', '293', '296', '300',
'305', '310', '312', '317', '321', '325', '330', '335', '340',
'345', '346', '351', '356', '358', '363', '366', '368', '373',
'377', '381', '384', '387', '392', '394', '396', '397', '401',
'405', '409', '411', '413', '419', '422', '424', '426', '431',
'432', '435', '438', '439', '442', '447', '451', '454', '459',
'463', '465', '469', '473', '478', '483', '485', '490', '494',
'499', '500', '504', '508', '513', '517', '519', '523', '526',
'527', '530', '534', '536', '541', '545', '550', '556', '560',
'563', '567', '570', '575', '576', '580', '584', '586', '590',
'595', '601']
Menor distância = 108.77451572
[Finished in 0.5s]
```

Resultado 2 – Resultado do algoritmo A*.

A fim de conferir os resultados do A* (verificar que ele realmente encontra o caminho ótimo entre duas cidades, adicionei o seguinte trecho de código no retorno da função, de modo que o algoritmo continua vasculhando soluções mesmo após o primeiro resultado:

```
            # Verifica se o destino foi alcançado
             if id_city == id_destination:
                 var.append(total_distance(path))
                 queue.get()
                 visited.remove(id_destination)
                 if len(var) == 5:
                     pprint(var)
                     return path
```

Trecho de código que exibe as 5 primeiras soluções encontradas.

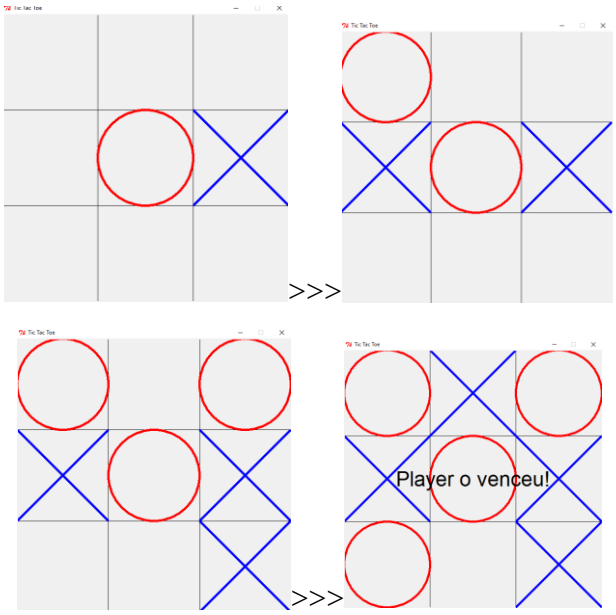
```
************************************
*                                  *
*       Algoritmos Greedy & A*     *
*                                  *
************************************
[108.77451572030004,
 108.77451572030006,
 111.28408980680555,
 111.14272327572996,
 112.74051572030004]
['202', '206', '211', '217', '219', '224', '227', '230', '231',
'236', '239', '242', '246', '248', '253', '257', '262', '265',
'268', '272', '277', '280', '283', '288', '293', '296', '300',
'305', '310', '312', '317', '321', '325', '330', '335', '340',
'345', '346', '351', '356', '358', '363', '366', '368', '373',
'377', '381', '384', '387', '392', '394', '396', '397', '401',
'405', '409', '411', '413', '419', '422', '424', '426', '431',
'432', '435', '438', '439', '442', '447', '451', '454', '459',
'463', '465', '469', '473', '478', '483', '485', '490', '494',
'499', '500', '504', '508', '513', '517', '519', '523', '526',
'527', '530', '534', '536', '541', '545', '550', '556', '560',
'563', '567', '570', '575', '576', '580', '584', '586', '590',
'595', '600', '597', '601']
Menor distância = 108.77451572030004
[Finished in 0.7s]
```

Resultado 3 – Resultado exaustivo do algoritmo A*,

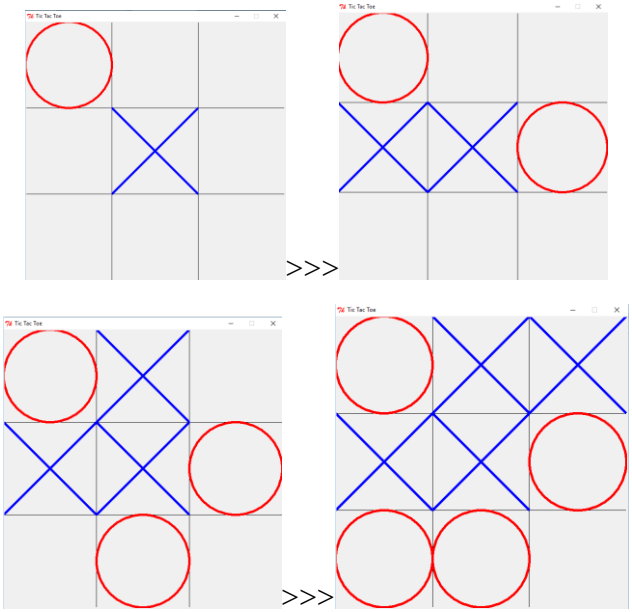
mostrando que a solução encontrada é ótima.

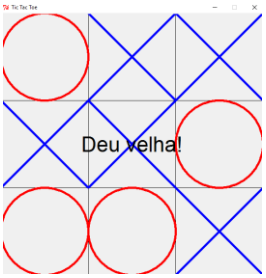
4.2 Resultados do jogo da velha

Essa foi a parte mais lúdica deste relatório. Primeiro, fez a “jogada ruim da borda” vista em sala de aula, a fim de testar a decisão da máquina. Ela responde jogando no centro, maximizando suas chances de vitória, conforme o esperado. Jogo despretensiosamente na outra borda, sendo conduzido a uma amarga derrota:

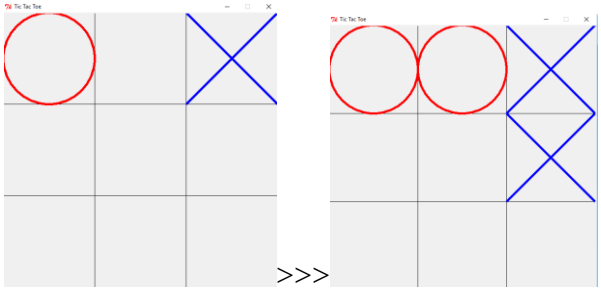


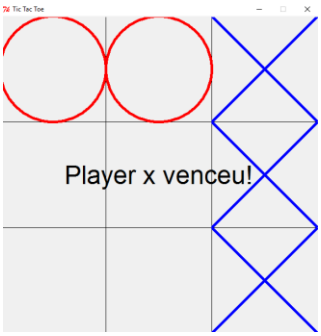
Em seguida, resolvi apelar começando a jogar no centro. Pressionei o agente até o empate inevitável:



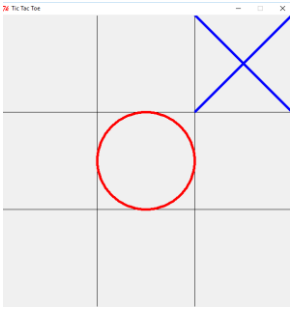


Brinquei por algum tempo tentando conseguir vitória, até que consegui a seguinte combinação, usando máxima profundidade de análise como 4:





Aumentando esse número para 5, a resposta da AI me surpreendeu:



E nunca mais venci.

A diferença é que usando profundidade 4 temos: [X, O, X, O], enquanto com profundidade 5 temos [X, O, X, O, X], ou seja, uma jogada a mais do agente a ser considerada na análise. E é justamente na terceira jogada de ‘x’ em que o jogador humano

‘x’ obtém a vitória sobre a AI, de modo que deixar de contabilizá-la criou uma espécie de ponto cego na estratégia do agente, levando-o à derrota.

Assim, a segunda jogada da AI aparentemente irracional que a levou a uma derrota feia é na verdade uma resignação de jogo devido à vitória certa do jogador humano (suponha que ela tentasse impedir a vitória jogando na posição inferior esquerda. O humano então jogaria no centro, garantindo uma vitória certa sobre o agente).

Apesar de não possuir todos os casos, a árvore de jogo da figura abaixo ajudou a entender melhor as decisões do agente no decorrer do jogo:

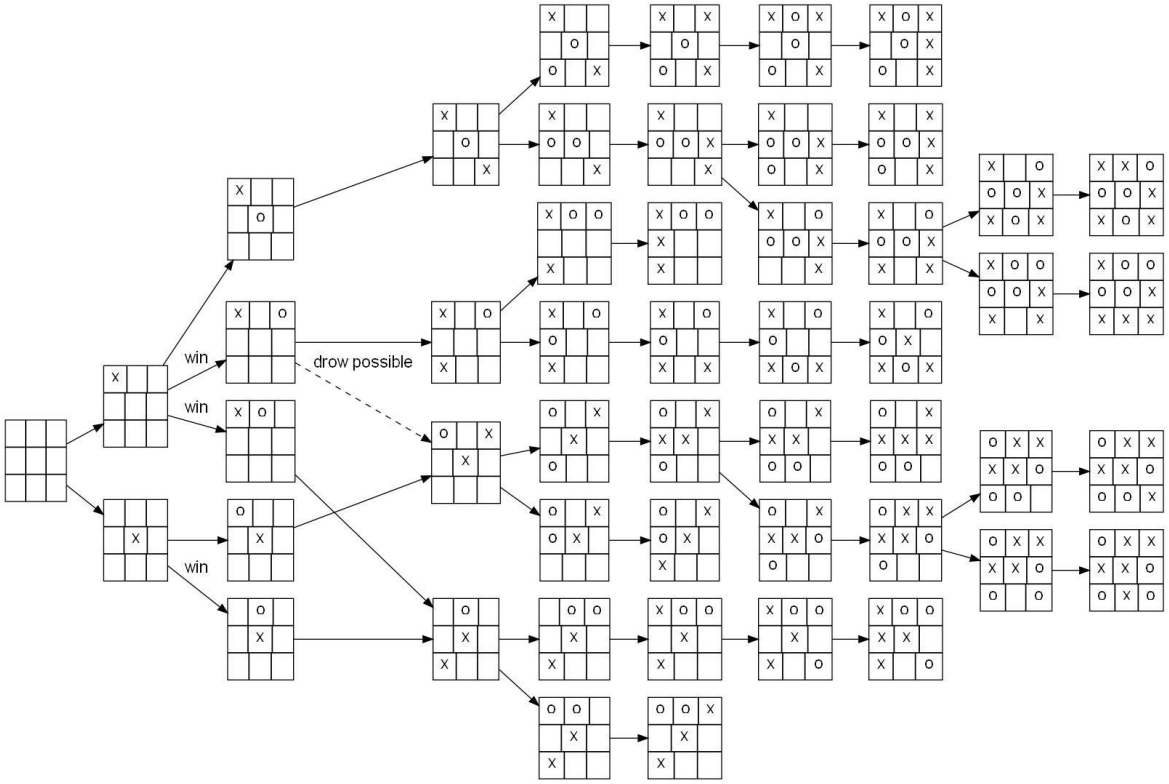


Figura 4 - Árvore completa do Jogo da Velha[1].

5. CONCLUSÃO

Esses dois projetos são muito interessantes por envolverem aplicações práticas dos conhecimentos adquiridos sobre busca, possuindo estratégias de resolução completamente diferentes, no caso da busca competitiva.

Como sugestão, acho que seria positivo considerar um mapa com informações sobre as distâncias reais das rotas, talvez até mesmo incorporar dados reais para a aplicação (ainda melhor se fossem provenientes de uma API como Google Maps), pois se tornaria uma verdadeira ferramenta e muito útil também.

Os algoritmos se provaram especialmente complexos, senti falta de casos de teste mais complexos para o programa, além de rotinas de teste propriamente ditas (ao estilo TDD).

6. REFERÊNCIAS BIBLIOGRÁFICAS

[1] Slides de aula: http://www.comp.ita.br/~pauloac/

[2] Stackoverflow (sempre): http://stackoverflow.com/

[3] Wikipedia: https://www.**wikipedia**.org