Trabalho 1
Tipos de busca de caminho

Edgar Sampaio De Barros, 16/0005213

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC 116653 - Introdução A Inteligência Artificial

160005213@aluno.unb.br

Este trabalho relata a experiência de implementação assim como a realização manual passo a passo de três métodos de busca e análise dos algoritmos. Para realizar as buscas foi proposto o mapa de cidades ilustrado na figura 1. No mapa o número entre duas cidades indica a distância entre elas.

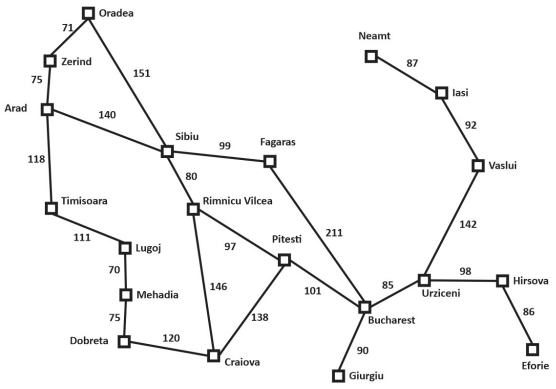

Figure 1. Mapa das cidades.

Para algumas buscas analisadas foi necessário construir a tabela a seguir (tabela 1), para realizar a sua construção foi utilizado a distância em linha reta da cidade e a cidade destino (Bucharest). [Stuart Russel 2013]

Cidade	Distância até Bucharest
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Table 1. Distâncias de cidades até Bucharest.

Todos os algoritmos foram testados no mesmo computador, tal computador tem as seguintes configurações:

Processador: Intel core i7-4770 3.4Ghz

Memoria RAM: 8gb

Sistema Operacional: Windows 10

1. Busca por custo uniforme

Segundo [Stuart Russel 2013], a busca de custo uniforme é uma variação da busca em largura, com uma simples extensão do algoritmo ele se torna ótimo para qualquer função em que as passadas tem custo. A extensão feita é, ao invés de escolher o nó mais raso do vértice para expandir, é escolhido o nó que tem o menor custo de caminho.

A seguir podemos ver a aplicação do algoritmo de busca por custo uniforme no grafo de exemplo, o vértice inicial será ”Arad” e o destino ”Bucharest”.

Passo 1: Começamos na cidade de ”Arad” calculamos a distância para cada cidade vizinha e comparamos os valores, como a cidade de ”Zerind” é a com menor custo o algoritmo expande para ela.

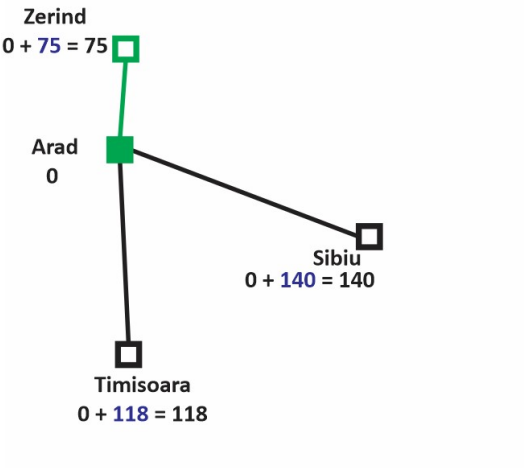

Figure 2. Primeiro passo do algoritmo de busca por custo uniforme.

Passo 2: Como a cidade de ”Zerind” só tem uma cidade adjacente que não seja ”Arad” calculamos o valor do caminho ate ela e depois comparamos os valores das bordas dos caminhos, é possivel ver que na figura 3 que o caminho com menor custo agora é o com borda em ”Timisoara” com o custo de 118, portanto o algoritmo irá expandir até ela.

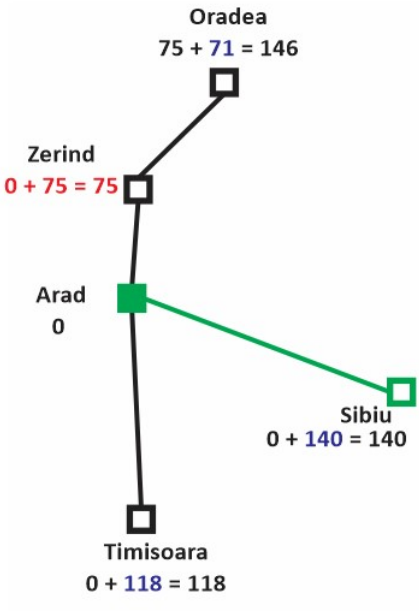

Figure 3. Segundo passo do algoritmo de busca por custo uniforme.

Passo 3: Assim como a cidade de ”Zerind” a cidade atual (”Timisoara””) só tem uma cidade possível para expandir, portando é calculado o custo para a cidade de ”Lugoj” que é a vizinha.

Como o caminho com o menor custo total é que tem ”Sibiu” como borda o algoritmo irá expandir para lá.

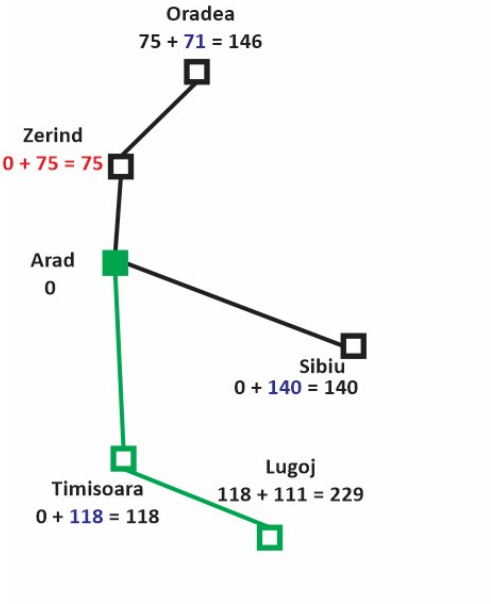

Figure 4. Terceiro passo do algoritmo de busca por custo uniforme.

Passo 4: Partindo da cidade de ”Simbiu” o algoritmo calcula o caminho para todas as cidades vizinhas e analisa o custo de todos os caminhos, e o caminho com menor custo é ”Arad” - ”Zerind” - ”Oradea” com custo total de 146, porém ao expandir ”Oradea” teremos um caminho com borda ”Sibiu” com um custo maior do que os outros que terminam no mesmo destino.

Como o caminho como menor custo total é que tem ”Rimnicu Vilcea” como borda o algoritmo irá expandir para la.

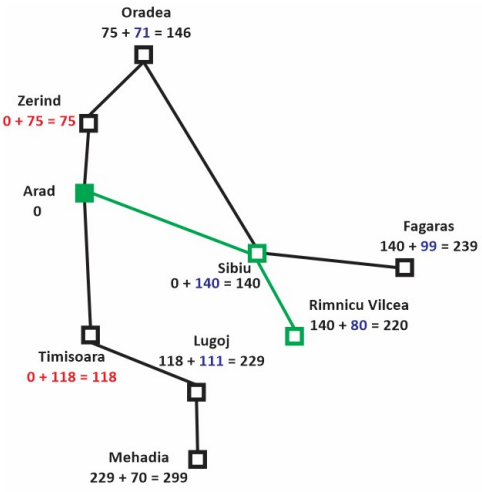

Figure 5. Quarto passo do algoritmo de busca por custo uniforme.

Passo 5: Como o caminho de menor custo é o com borda em ”Rimnicu Vilcea” o algoritmo explora e calcula a distância para cada cidade vizinha a ela.

O caminho escolhido para expansão da busca por ser o de menor custo é o com borda em ”Fagaras” que tem como custo 239.

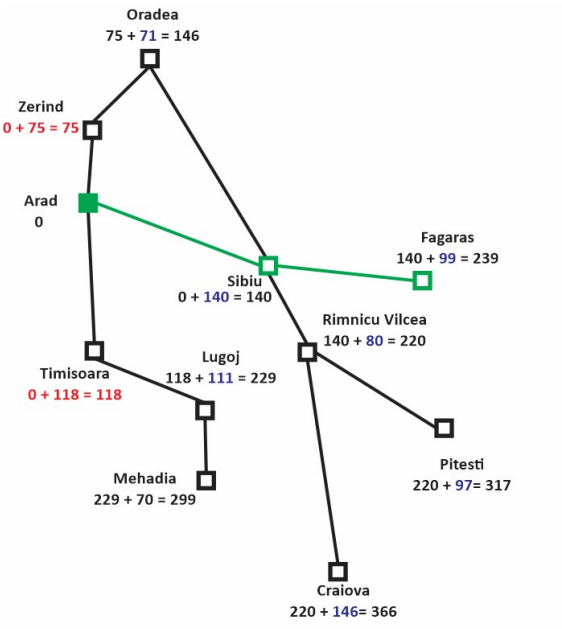

Figure 6. Quinto passo do algoritmo de busca por custo uniforme.

Passo 6: Expandido para a única cidade vizinha a ”Fagaras” encontramos a cidade destino da busca, porém o algoritmo se mantém e continua expandindo para encontrar o caminho com menor custo, então ele se expande para o caminho com o menor custo, que é o com borda em ”Mehadia”.

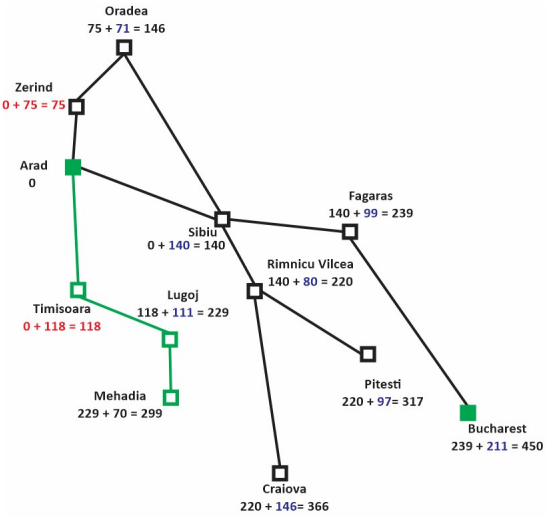

Figure 7. Sexto passo do algoritmo de busca por custo uniforme.

Passo 7: Após expandir para a única cidade vizinha a ”Mehadia” o algoritmo novamente irá se expandir para o caminho com o menor custo, que no caso é o com borda

em ”Pitesti”, porém ao expandir para as bordas dessa cidade, novamente ele se encontra com a cidade destino da busca, porém ele continua se expandindo para ter certeza que irá encontrar o caminho de menor custo.

O algoritmo segue se expandindo a partir da cidade de ”Craiova”.

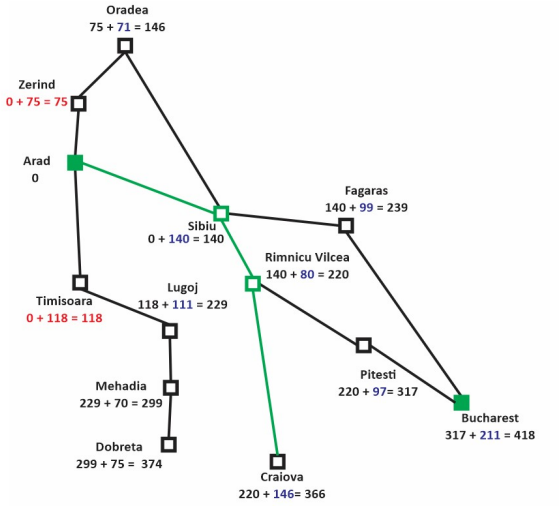

Figure 8. Sétimo passo do algoritmo de busca por custo uniforme.

Passo 8: Por fim ao calcular o custo das rotas das cidades vizinhas a ”Craiova” o algoritmo iria se expandir para o caminho em verde na figura 9 porém como o nó da borda do caminho é a cidade destino da busca, o algoritmo tem certeza que este é o caminho de menor custo e encerra sua execução.

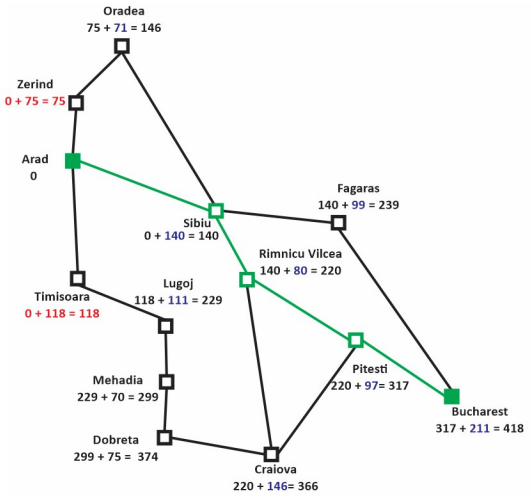

Figure 9. Oitavo passo do algoritmo de busca por custo uniforme.

A seguir é mostrado uma implementação do algoritmo de busca por custo uniforme feita usando python3.

Na implementação da função que realiza a busca primeiramente é feito uma

checagem para conferir se os nós de começo e termino fazem parte do grafo, caso algum deles não faça a função retorna pois é impossível realizar a busca.

Em sequência é criado com auxílio de uma biblioteca padrão do python3 uma fila de prioridade, tal estrutura armazena valores de modo que o primeiro elemento da fila sempre será com menor valor. Na fila criada será armazenado uma tuple cujo primeiro elemento é o custo do caminho e o segundo elemento é um vetor contendo todas as cidades que fazem parte do caminho. A fila é inicializada com um elemento (0, cidadeDePartida).

A função termina com um looping que irá iterar até que a fila de prioridade esteja vazia ou caso ache o menor caminho.

Dentro do looping é desenfileirado o primeiro elemento da fila, com ele nos temos as informações do custo e do menor caminho até agora, primeiro é realizado uma checagem para averiguar se o último elemento do caminho é o destino, caso seja, o algoritmo retorna o caminho e o custo e logo depois se encerra, pois este é o menor caminho que buscamos. Caso o algoritmo continue será realizado outro looping que irá a iterar para todos os vizinhos da última cidade do caminho e adicionar na fila a tuple (novo_custo, [caminho] + vizinho), onde o novo custo é calculado na função calc_cost que realiza a soma $custo\_atual + distância\_do\_nó\_ate\_o\_vizinho$.

Caso o looping principal termine e não encontre o menor caminho significa que o algoritmo não foi possível encontrar um caminho da cidade de origem até a cidade destino, portanto ele retorna None, -1.

```
from graph import *
import queue as Q

def calc_cost(cost, grafo, current, neighbor):
    return cost + grafo[current][neighbor]

def uniform_cost(graph, start, end):
    if not start in graph or not end in graph:
        return None, -1

    q = Q.PriorityQueue()
    q.put((0, [start]))

    while not q.empty():
        teste = q.queue
        node = q.get()
        cost = node[0]
        path = node[1]
        current = path[-1]

        if current == end:
            return path, cost

        for neighbor in graph[current]:
            if neighbor in path:
                continue
            temp = path.copy()
            temp.append(neighbor)
            temp_cost = calc_cost(cost, graph, current, neighbor)

```

```
31              q.put((temp_cost, temp))
32
33      return None, -1
```

Listing 1. Implementação do algoritmo Busca gulosa

2. Busca gulosa

A busca gulosa tenta expandir o nó de acordo com a função heurística $f(n) = h(n)$, onde h é a distância entre o nó n e o objetivo, ou seja, ela sempre vai expandir sua busca de acordo com o nó mais próximo do objetivo final.

A seguir é demonstrado como o algoritmo funciona realizando passo a passo uma busca de caminho de origem na cidade ”Arad” com destino a cidade ”Bucharest”.

Passo 1: Começamos na cidade de Arad, ao consultar na tabela 1 as distâncias das cidades adjacentes a ela temos:

Zerind com 347, Sibiu com 253 e Timisoara com 329

Como a cidade de Sibiu é a com menor distância o algoritmo irá expandir para la.

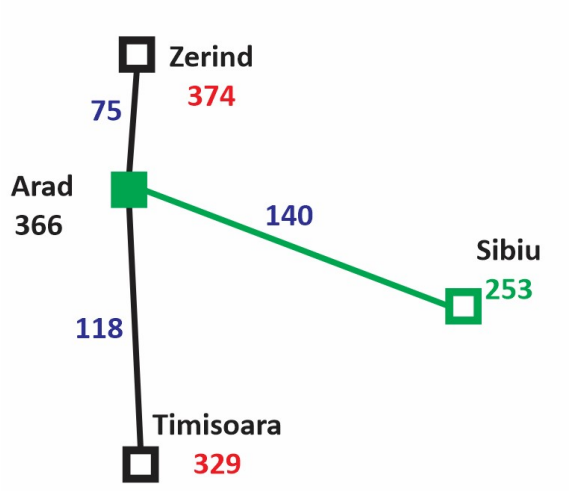

Figure 10. Primeiro passo do algoritmo de busca gulosa.

Passo 2: Partindo da cidade de Sibiu, repetimos o processo do passo anterior, consultando a tabela 1 obtemos:

Oradea com 390, Rimnicu Vilcea com 193 e Fagaras com 178

Como a cidade de Fagaras é a com menor distância o algoritmo irá expandir para la.

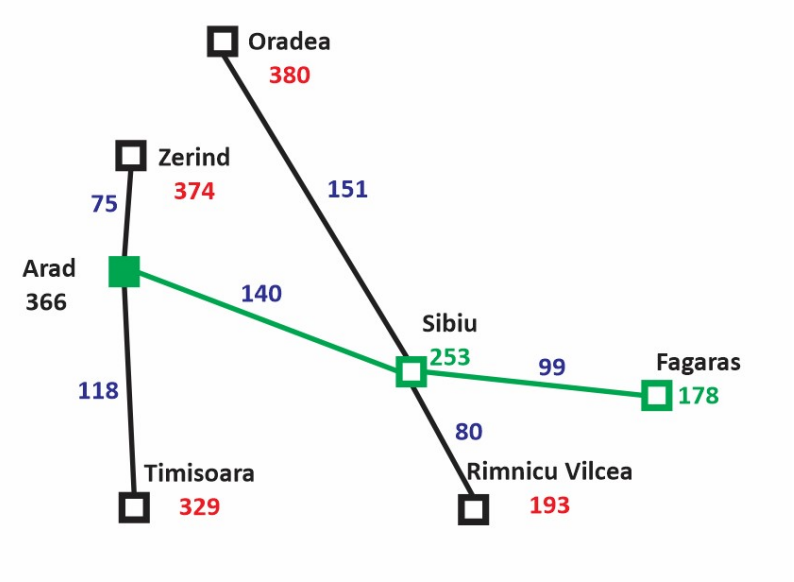

Figure 11. Segundo passo do algoritmo de busca gulosa.

Passo 3: Partindo da cidade de Fagaras, repetimos o processo, consultando a tabela 1 obtemos:

Como Fagaras só tem a cidade de Bucharest adjacente temos que expandir para ela.

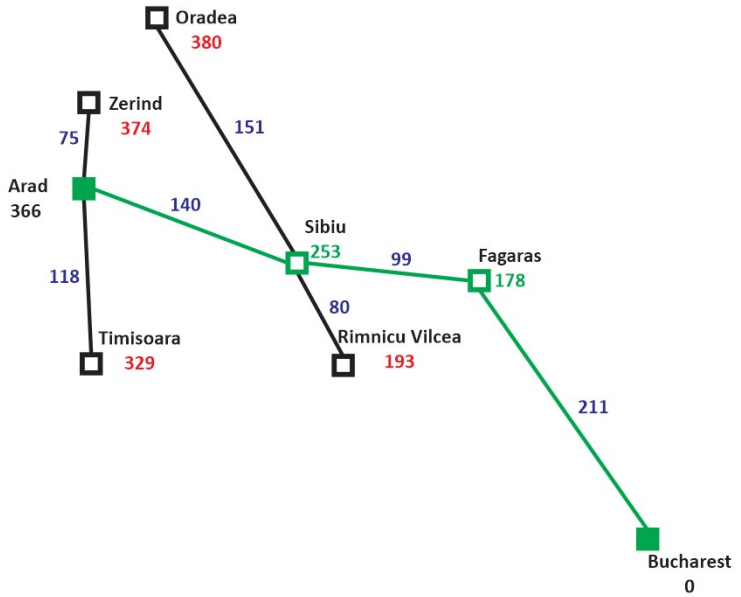

Figure 12. Terceiro passo do algoritmo de busca gulosa.

Como chegamos na cidade destino o algoritmo tem fim, o caminho encontrado pode ser visto na imagem 13.

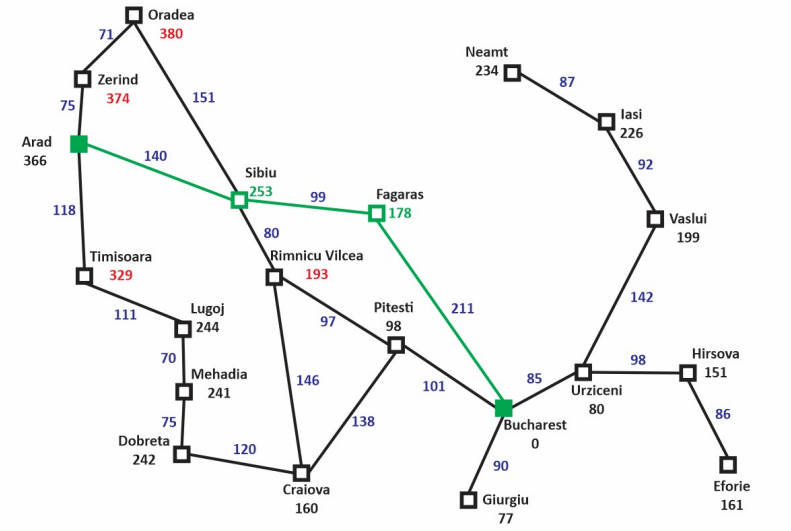

Figure 13. Caminho encontrado no algoritmo de busca gulosa.

O algoritmo desenvolvido para realizar a busca gulosa pode ser visto logo abaixo, nele assim como na busca por custo uniforme a primeira coisa a se fazer é conferir se o nó de começo e termino estão presentes no grafo, após a checagem o algoritmo da busca começa.

O primeiro passo é inicializar algumas variáveis para auxiliar na busca, começando com a variável que irá armazenar o custo total do caminho ate certo ponto, em sequencia uma variável para armazenar o caminho realizado ate então, uma que armazena o nó que irá realizar a checagem e uma para o conjunto que irá guardar os nós que já foram visitados.

Logo após a inicialização das variáveis vem um looping que irá iterar até que o algoritmo chegue ao destino ou não seja mais capaz de prosseguir. Em cada iteração do looping o algoritmo irá calcular por meio da função 'calc_next_node' o vizinho da última cidade visitada com menor custo de distância de acordo com a tabela 1 e irá adicioná-lo no vetor que armazena o caminho e somar o custo da distância na variável $cost$.

```
from grafo import *

def calc_next_node(node):
    return min(graph[node].keys(), key= lambda x:d_bucharest[x])

def gulosa(graph, start, end):
    if not start in graph or not end in graph:
        return -1

    cost = 0
    path  = [start]
    node = start
    visited = set(node)

    while(node != end):
        next_node = calc_next_node(node)
        cost += graph[node][next_node]
        node = next_node
```

```
        path.append(node)

        if node in visited:
            return "Caminho n o encontrado"
        else:
            visited.add(node)

    return (path, cost)
```

Listing 2. Implementação do algoritmo Busca gulosa

3. Busca A*

O último algoritmo analisado é o de busca A*, para decidir qual nó ela irá expandir sua busca o algoritmo faz uso da função heurística $f(n) = g(n) + h(n)$, onde $g(n)$ é o custo do nó inicial até o nó n e $h(n)$ é o custo para ir do nó n ate o nó final (valores da tabela 1). Contanto que a função heurística $h(n)$ satisfaça certas condições a busca A* será completa e ótima [Stuart Russel 2013].

A seguir podemos ver o passo a passo de como o algoritmo é aplicado em uma busca com início em ”Arad” para ”Bucharest”. Nas figuras utilizadas para ilustrar o precedimento de busca abaixo do nome das cidades se encontra uma soma que representa a função $f(n)$, parcela da soma em azul representa a distância do nó até o nó inicial ($g(n)$) e a parcela em preto representa a distância em linha reta até a nó final ($n(n)$).

Passo 1: Calculado o $f(n)$ para o nó inicial.

Figure 14. Primeiro passo passo do algoritmo de busca A*.

Passo 2:

Agora o algoritmo calcula $f(n)$ de cada um dos nós vizinhos de ”Arad”, como o valor de $f(n)$ de ”Sibiu” foi o menor encontrado até agora, o algoritmo expande para lá.

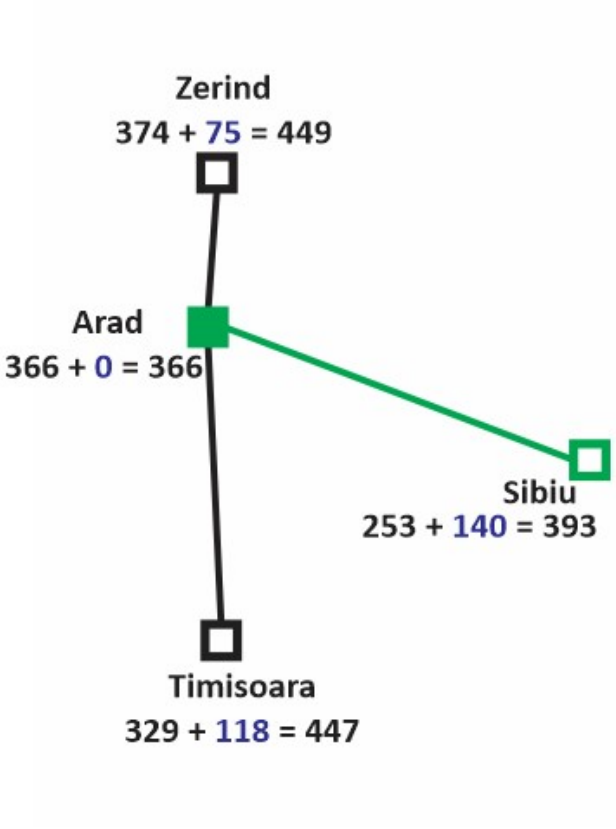

Figure 15. Segundo passo do algoritmo de busca A*

Passo 3:

Repetimos o processo do passo anterior e calculamos o $f(n)$ de cada nós vizinhos a ”Sibiu”, como o menor valor de $f(n)$ encontrado foi de ”Rimnicu Vilcea” o algoritmo expande para este nó.

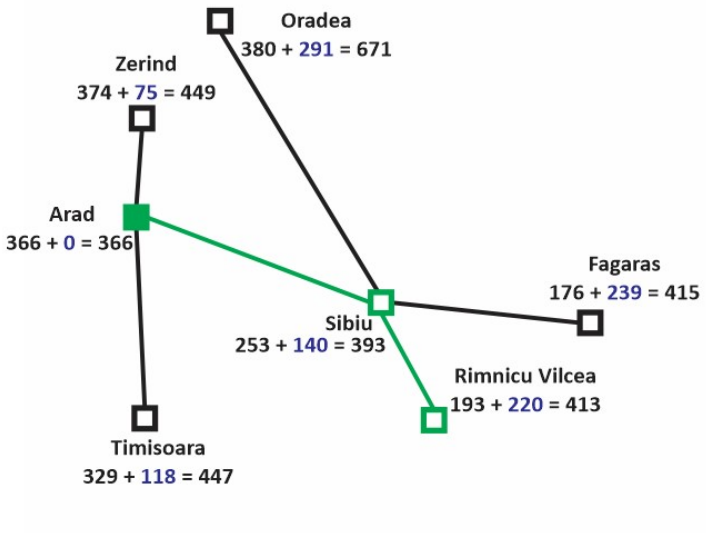

Figure 16. Terceiro passo do algoritmo de busca A*

Passo 4:

Novamente repetimos o processo de calcular o $f(n)$ de cada nós vizinhos ao nó atual (”Rimnicu Vilcea”) e encontramos como o menor valor ”Pitesti” portanto o algoritmo expande para este nó.

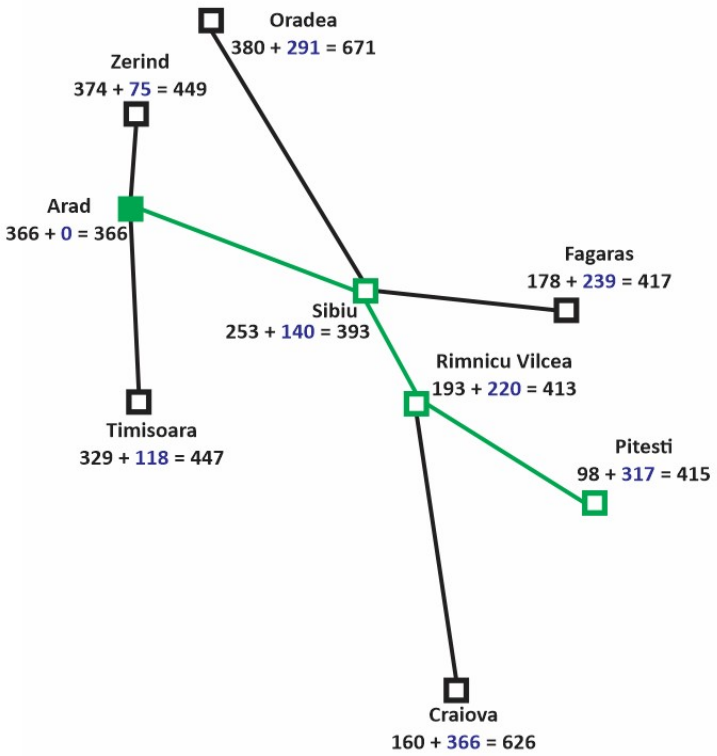

Figure 17. Quarto passo do algoritmo de busca A*

Passo 5: Por fim, como o nó que representa a cidade de ”Bucharest” é vizinha ao nó atual (”Pitesti”) o algoritmo chega na cidade destino e ao calcular o valor de $f(n)$ é obtido a distância entre a cidade inicial e o destino, pois $h(n) = 0$. Em verde é possível ver o caminho traçado pelo algoritmo para chegar do nó inicial ao nó final.

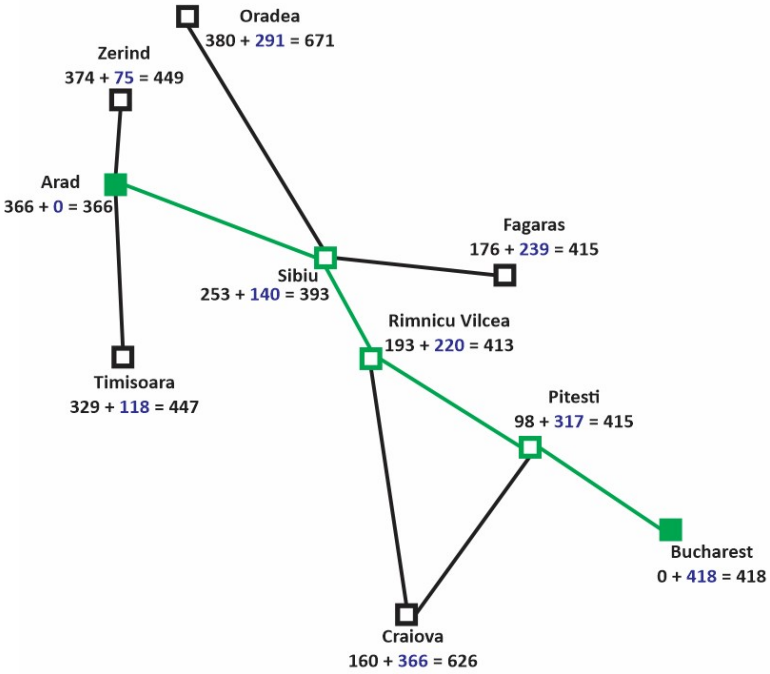

Figure 18. Quinto passo do algoritmo de busca A*

A seguir é possível ver uma implementação do algoritmo de busca A* em python3. Como se pode ver, com exceção da função ”calc_cost” o resto do algoritmo é exatamente igual ao algoritmo de busca uniforme, a diferença nas funções é que no algoritmo de A* a função ”calc_cost” utiliza $f(n) = g(n) + h(n)$ para decidir para qual nó irá expandir enquanto a função de busca uniforme usa apenas $g(n)$.

```
from graph import *
import queue as Q

def calc_path_cost(path):
    path_cost = 0
    for i in range(len(path)-1):
        path_cost += graph[path[i]][path[i+1]]

    return path_cost

def calc_cost(neighbor, path):
    global d_bucharest
    path_cost = calc_path_cost(path)

    return path_cost + d_bucharest[neighbor]

def a_start(graph, start, end):
    if not start in graph or not end in graph:
        return None, -1

    global d_bucharest
    q = Q.PriorityQueue()
    q.put((d_bucharest[start], [start]))

```

```
    while not q.empty():
        node = q.get()
        cost = node[0]
        path = node[1]
        current = path[-1]

        if current == end:
            return(path, cost)

        for neighbor in graph[current]:
            path_temp = path.copy()
            path_temp.append(neighbor)
            temp_cost = calc_cost(neighbor, path_temp)

            q.put((temp_cost, path_temp))

    return None, -1

print(a_start(graph, 'Arad', 'Bucharest'))
```

Listing 3. Implementação do algoritmo Busca gulosa

Todos os algoritmos utilizados se encontraram em um repositório no Github e podem ser acessados pelo link: github.com/edgarsamp/IIA.

References

[Stuart Russel 2013] Stuart Russel, P. N. (2013). *Inteligência Artificial*. Elsevier, terceira edição edition.