Fichas

14 de dezembro de 2023 23:35

Ficha TP1

i) a) A → B → D → H → I → E → J → K → C → F → L → M → G → N → O

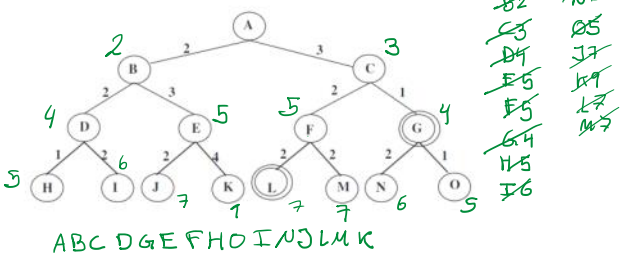
b) A C F L

c) A C G

ii) a) A B C D E F G H I J K L M N O

b) A C F L

iii)



~~B2~~ ~~C3~~ ~~D4~~ ~~E5~~ ~~F5~~ ~~G4~~ ~~H5~~ ~~I6~~

~~N6~~ ~~O5~~ ~~J7~~ ~~K7~~ ~~L7~~ ~~M7~~

A B C D G E F H O I N J L M K

iv. a) Falso

b) Verdadeiro → Não guarda nodos

c) Não tem função de desempate

d) Falso, DEPENDE

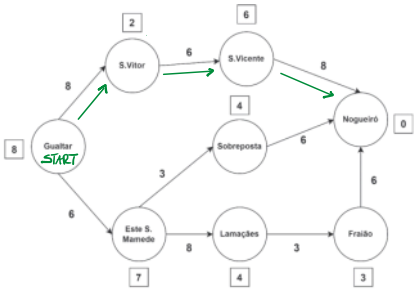
```python
def bfs(self, start, end):

    visited = set()
    fila = Queue()
    cost = 0

    fila.put(start)
    visited.add(start)

    parent = dict()
    parent[start] = None

    path_found = False
    while not fila.empty() and not path_found:
        current_node = fila.get()
        if current_node == end:
            path_found = True
        else:
            for (adjacent, peso) in self.m_graph[current_node]:
                if adjacent not in visited:
                    fila.put(adjacent)
                    parent[adjacent] = current_node
                    visited.add(adjacent)
```

```python
def dfs(self, start, end, path=None, visited=None):
    if visited is None:
        visited = set()

    if path is None:
        path = []

    path.append(start)
    visited.add(start)

    if start == end:
        cost_t = self.calculate_cost(path)
        return path, cost_t

    for (adjacent, peso) in self.m_graph[start]:
        if adjacent not in visited:
            result = self.dfs(adjacent, end, path, visited)
            if result is not None:
                return result

    path.pop()  # backtrack
    return None
```

Custo uniforme ≈ DIJKSTRA →

```python
def greedy_dijkstra(self, start, end):

    if start not in self.m_graph.keys() or end not in self.m_graph.keys():
        return None

    open_set = [(0, start)]
    visited = set()
    came_from = {}
    score = {node.get_name(): float("inf") for node in self.m_nodes}

    while open_set:
        (current_score, current_node) = heapq.heappop(open_set)

        if current_node == end:
            path = reconstruct_path(came_from, current_node)
            return path, score[current_node]

        if current_node not in visited:
            visited.add(current_node)
            for (neighbour, weight) in self.m_graph[current_node]:
                if neighbour not in visited:
                    tentative_score = current_score + weight
                    if tentative_score < score[neighbour]:
                        score[neighbour] = tentative_score
                        came_from[neighbour] = current_node
                        heapq.heappush(open_set, (score[neighbour], neighbour))
```

Ficha TP2



OPEN-SET

~~Gualtar 0~~

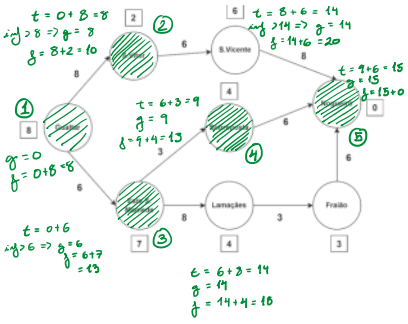
~~S.Vitor 2~~

Ests. 7

~~S.Vicente 6~~

~~Nogueiró 0~~

```python
def greedy(self, start, end):

    open_set = [(0, start)]
    visited = set()
    came_from = {}

    while open_set:
        (current_score, current_node) = heapq.heappop(open_set)

        if current_node == end:
            path = reconstruct_path(came_from, current_node)
            return path, current_score

        if current_node not in visited:
            visited.add(current_node)
            for (neighbour, weight) in self.m_graph[current_node]:
                if neighbour not in visited:
                    score = self.get_h(neighbour)
                    came_from[neighbour] = current_node
                    heapq.heappush(open_set, (score, neighbour))
```




```python
def astar(self, start, end):

    if start not in self.m_graph.keys() or end not in self.m_graph.keys():
        return None

    # The set of discovered nodes that may need to be (re-)expanded.
    # Initially, only the start node is known.
    # This is usually implemented as a min-heap or priority queue rather than a hash-set.
    open_set = [(0, start)]
    closed_set = set()

    # For node n, came_from[n] is the node immediately preceding it on the cheapest path fr
    # currently known.
    came_from = {}

    # For node n, g_score[n] is the cost of the "cheapest path from start to n" currently k
    g_score = {node.get_name(): float("inf") for node in self.m_nodes}
    g_score[start] = 0

    # For node n, f_score[n] := g_score[n] + h(n). f_score[n] represents our current best g
    # how cheap a path could be from start to finish if it goes through n.
    f_score = {node.get_name(): float("inf") for node in self.m_nodes}
    f_score[start] = g_score[start] + self.get_h(start)  # <=> 0 + self.get_h(start)
```

```python
while open_set:
    current = heapq.heappop(open_set)[1]
    if current == end:
        path = reconstruct_path(came_from, current)
        return path, g_score[current]

    for (neighbor, weight) in self.m_graph[current]:
        # d(current,neighbor) is the weight of the edge from current to neighbor
        # tentative_g_score is the distance from start to the neighbor through current

        tentative_g_score = g_score[current] + weight

        if tentative_g_score < g_score[neighbor] and neighbor not in closed_set:
            # This path to neighbor is better than any previous one
            came_from[neighbor] = current
            g_score[neighbor] = tentative_g_score
            f_score[neighbor] = g_score[neighbor] + self.get_h(neighbor)
            if neighbor not in open_set:
                heapq.heappush(open_set, (f_score[neighbor], neighbor))

    closed_set.add(current)
```

Formulação do problema

- Tipo do problema: Procura clássica
- Estado inicial: Gualtar
- Estado objetivo: Nogueiró

Notas:

Iterative Deepening DFS

- Tipo de problema : Procura clássica
- Estado inicial : Gualtar
- Estado objetivo : Nogueiró
- Operações : viajar entre as cidades
- Estados : { Gualtar, ... }
- Custo solução : determinado pelo custo de cada caminho tomado

Notas :

Iterative Deepening DFS
↳ incorpora um limite de profundidade

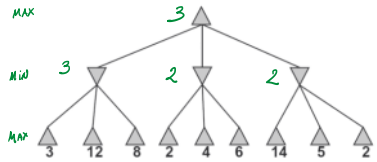
! recomeça o algoritmo desde a raíz sempre que o limite incrementa

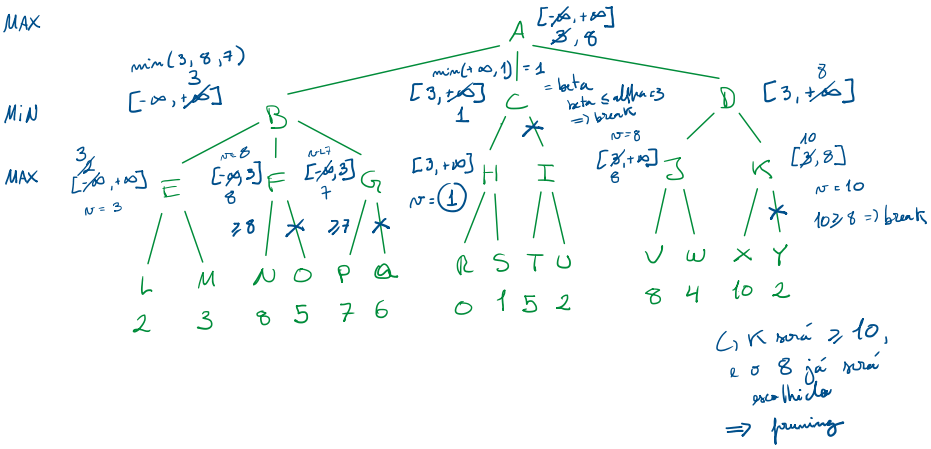
Tipos de problemas :
http://aimaterials.blogspot.com/p/problem-solving-agent_11.html

Ficha P. nº3

Tipo de problema : Procura clássica, estado único
(...)

Ficha TP nº4






```python
def minimax_2(self, depth, alpha, beta, is_maximizing):
    result = self.check_win()
    if result == self.bot:
        return 1
    if result == self.player:
        return -1
    if self.check_draw() or depth == 0:
        return 0

    if is_maximizing:
        max_score = -1000  # -inf
        for move in self.get_available_moves():
            self.board[move] = self.bot
            score = self.minimax_2(depth - 1, alpha, beta, False)
            max_score = max(max_score, score)
            self.board[move] = ' '
            alpha = max(alpha, score)
            if beta <= alpha:
                break

        return max_score
    else:
        min_score = 1000  # +inf
        for move in self.get_available_moves():
            self.board[move] = self.player
            score = self.minimax_2(depth - 1, alpha, beta, True)
            min_score = min(min_score, score)
            self.board[move] = ' '
            beta = min(beta, score)
            if beta <= alpha:
                break

        return min_score
```

Ficha prática nº5

def f_obj = x²

```
function HILL-CLIMB(problem):
  current = initial state of problem
  repeat:
    neighbor = highest valued neighbor of current
    if neighbor not better than current:
      return current
    current = neighbor
```

Variant	Definition
steepest-ascent	choose the highest-valued neighbor
stochastic	choose randomly from higher-valued neighbors
first-choice	choose the first higher-valued neighbor
random-restart	conduct hill climbing multiple times

Um estado fica numa é escolhido, apesar de muitos vezes ser necessário para evitar ficar preso em mínimos/máximos locais.

higher temperature ⇒ more likely to move to a worst neighbour
∧
energy : how worse/better)
↳ less likely to move to a very bad neighbour

```
function SIMULATED-ANNEALING(problem, max):
  current = initial state of problem
  for t = 1 to max:
    T = TEMPERATURE(t)
    neighbor = random neighbor of current
    ΔE = how much better neighbor is than current
    if ΔE > 0:
      current = neighbor
    with probability e^(ΔE/T) set current = neighbor
```

PROBABILITY
$e^{\Delta E/T}$

Procura Tabu

↳ Penalizar movimentos que levam a solução para espaços de procura visitados anteriormente.

- A Procura Tabu, no entanto, aceita de forma determinística soluções que não melhoram para evitar ficar presa em mínimos locais.
- **Estratégia:**
  - Ideia chave: manter a sequência de nós já visitados (Lista tabu);
  - Partindo de uma solução inicial, a procura move-se, a cada iteração, para a melhor solução na vizinhança, não aceitando movimentos que levem a soluções já visitadas, esses movimentos conhecidos ficam armazenados numa lista tabu;
  - A lista permanece na memória guardando as soluções já visitadas (tabu) durante um determinado espaço de tempo ou um certo número de iterações (prazo tabu). Como resultado final é esperado que se encontre um valor ótimo global ou próximo do ótimo global.

↳ LISTA DE VISITADOS

```
public Solution run(Solution initialSolution) {
        Solution bestSolution = initialSolution;
        Solution currentSolution = initialSolution;

        Integer currentIteration = 0;
        while (!stopCondition.mustStop(++currentIteration, bestSolution)) {

                List<Solution> candidateNeighbors = currentSolution.getNeighbors();
                List<Solution> solutionsInTabu = IteratorUtils.toList(tabuList.iterator());

                Solution bestNeighborFound = solutionLocator.findBestNeighbor(candidateNeighbors, solutionsInTabu);
                if (bestNeighborFound.getValue() < bestSolution.getValue()) {
                        bestSolution = bestNeighborFound;
                }

                tabuList.add(currentSolution);
                currentSolution = bestNeighborFound;

                tabuList.updateSize(currentIteration, bestSolution);
        }

        return bestSolution;
}
```

```
@Override
public Solution findBestNeighbor(List<Solution> neighborsSolutions, final List<Solution> solutionsInTabu) {
        //remove any neighbor that is in tabu list
        CollectionUtils.filterInverse(neighborsSolutions, new Predicate<Solution>() {
                @Override
                public boolean evaluate(Solution neighbor) {
                        return solutionsInTabu.contains(neighbor);
                }
        });

        //sort the neighbors
        Collections.sort(neighborsSolutions, new Comparator<Solution>() {
                @Override
                public int compare(Solution a, Solution b) {
                        return a.getValue().compareTo(b.getValue());
                }
        });

        //get the neighbor with lowest value
        return neighborsSolutions.get(0);
}
```

```
descendente(A,B) :-
        filho(A,B).
descendente(A,B) :-
        filho(A,X),
        descendente(X,B).
```

↳ com variáveis:

```
descendente(A,B,1) :-
        filho(A,B).
descendente(A,B,G) :-
        filho(A,X),
        descendente(X,B,N),
        G is N+1.
```

- Em PROLOG o incremento de uma variável nunca pode ser feito como N is N+1 (is é a atribuição numérica)
  - Se N não estiver instanciado ocorre uma falha quando se tenta avaliar N+1
  - Se N estiver instanciado não poderemos obrigar a mudar o seu valor
  - Pode ser usado N1 is N+1
- Quando num facto ou regra não interesse o valor de uma variável, esta pode ser substituída por _

- Adição X + Y
- Subtracção X - Y
- Multiplicação X * Y
- Divisão X / Y
- Divisão inteira X // Y
- Resto da divisão inteira X mod Y
- Potência X ^ Y
- Simétrico - X

- ip(X) parte inteira de X
- ln(X) logaritmo natural de X
- log(X) logaritmo decimal de X
- max(X,Y) máximo entre X e Y
- min(X,Y) mínimo entre X e Y
- rand(X) gera um número aleatório entre 0 e X
- sign(X) sinal de X
- sin(X) seno de X (graus)
- sqrt(X) raiz quadrada de X
- tan(X) tangente de X (graus)

- Igualdade X==Y
- Diferença X\==Y
- Maior X>Y
- Menor X<Y
- Menor ou igual X=<Y
- Maior ou igual X >= Y

- = para a atribuição simbólica X=a
- is para a atribuição numérica X is 5

- A atribuição simbólica é bidireccional

- Para X=Y
  - Se X não está instanciado e Y está então temos X←Y
  - Se X está instanciado e Y não está então temos X→Y
  - Se nenhum está instanciado então atravessam a ser a mesma variável
  - Se ambos estão instanciados com o mesmo valor então há sucesso
  - Se ambos estão instanciados com valores diferentes então ocorre uma falha

- A atribuição numérica é unidireccional
- Do lado direito do *is*, se estiverem envolvidas variáveis, elas devem estar instanciadas
- Do lado esquerdo a variável não deve estar instanciada, senão ocorre uma falha
- Do lado direito as variável que apareçam devem estar instanciadas
- Em PROLOG N is N+1 nunca tem sucesso

```
par(N) :- 0 is N mod 2.
```

(⇒ `N mod 2 is 0` does not work!)