Anexo1

```
def main():
    print("Starting client!")
    ag = Agent()
    if ag.getConnection()!= -1:
        ag.run([])
```

Imagem 1. Agente inicialmente não conhece nenhum obstáculo invisível - lista vazia([ ]) nos parâmetros de run()

```
def run(self, invisible_obstacles, position=None):
    # Get the position of the Goal
    self.frontier_nodes.clear()
    self.visited_nodes.clear()
    self.goalNodePos = self.getGoalPosition()

    # Get information of the weights for each step in the world ..
    self.weightMap = self.getWeightMap()
    # Get max coordinates
    self.maxCoord = self.getMaxCoord()
    # Get the initial position of the agent
    self.state = self.getSelfPosition()
```

Imagem 2. `self.getSelfPosition()` retorna sempre a posição inicial e não a posição atual do agente.

```
# Adicionamos o clear para "limpar" a lista de frontier e de visited para, quando o agente "bater" numa parede
#  invisivel, volte a pensar desde o novo inicios, endo este a posição atual(antes de bater no obstáculo invisivel)
def clear(self):
    self.queue_data = []
```

Imagem 3. Criação de método clear na class Queue, serve no caso de ser a segunda ou mais vez que o run() corre o agente pensar tudo “do zero”.

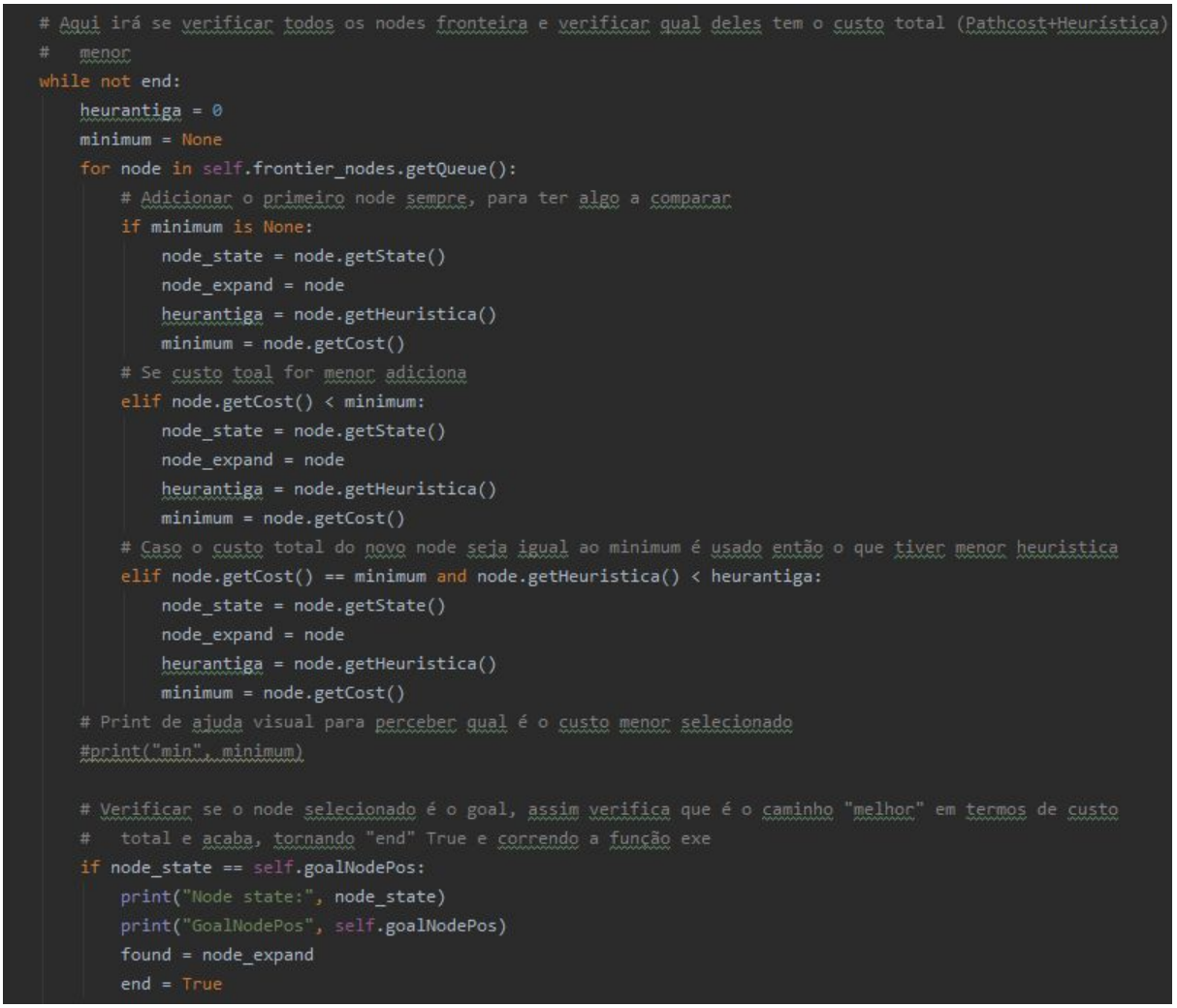
```
# Se não for a posição inicial, o state será a posição nova (acontece depois de bater no obstáculo invisível)
if position != None:
    self.state = position

# Se a lista não for vazia, ou seja, na primeira vez que o run corre onde não se bateu contra um obstáculo
#  invisível, irá correr
if invisible_obstacles != []:
    # Irá adicionar na lista de obstáculos o obstáculo invisível encontrado na tentativa anterior
    self.obst[invisible_obstacles[0][0]][invisible_obstacles[0][1]] = 1

# Start thinking
# Na primeira vez que corre serão os obstáculos visíveis, nas vezes que corre depois de serem encontrados
#  obstáculos invisíveis estes serão incluídos
obst = self.getObstaclesTotal()
i = 0
end = False
found = None
node_expand = None
node_state = None

# O valor da heurística vem da função dist, que recebe o X e Y do node atual e do goal
heuristica = dist(self.state[0], self.state[1], self.goalNodePos[0], self.goalNodePos[1])
```

Imagem 4. Dos parâmetros que se seguem, o "i" é o parâmetro que irá dizer quantas casas irão ser arredondadas nos prints explicativos que irão seguir, e o "end" é inicializado a "False" e só colocado a "True" quando o melhor caminho é encontrado.

```
# Criamos obstacles aqui porque teremos que mudar estes ao encontrar obstáculos invisíveis.
#  Estes, depois de encontrados, são então adicionados aos obstáculos
self.obst = self.getObstacles()
```

```
# Precisamos de obter os obstáculos a partir do self e não do ficheiro, por isso temos esta função
#  Isto porque os obstáculos irão ser mais do que os iniciais ao serem adicionados os invisíveis encontrados
def getObstaclesTotal(self):
    return self.obst
```

Imagens 4.1. e 4.2. Criação de obstáculos



```
# Aqui irá se verificar todos os nodes fronteira e verificar qual deles tem o custo total (Pathcost+Heurística)
#   menor
while not end:
    heurantiga = 0
    minimum = None
    for node in self.frontier_nodes.getQueue():
        # Adicionar o primeiro node sempre, para ter algo a comparar
        if minimum is None:
            node_state = node.getState()
            node_expand = node
            heurantiga = node.getHeuristica()
            minimum = node.getCost()
        # Se custo toal for menor adiciona
        elif node.getCost() < minimum:
            node_state = node.getState()
            node_expand = node
            heurantiga = node.getHeuristica()
            minimum = node.getCost()
        # Caso o custo total do novo node seja igual ao minimum é usado então o que tiver menor heuristica
        elif node.getCost() == minimum and node.getHeuristica() < heurantiga:
            node_state = node.getState()
            node_expand = node
            heurantiga = node.getHeuristica()
            minimum = node.getCost()
    # Print de ajuda visual para perceber qual é o custo menor selecionado
    #print("min", minimum)

    # Verificar se o node selecionado é o goal, assim verifica que é o caminho "melhor" em termos de custo
    #   total e acaba, tornando "end" True e correndo a função exe
    if node_state == self.goalNodePos:
        print("Node state:", node_state)
        print("GoalNodePos", self.goalNodePos)
        found = node_expand
        end = True
```

Imagem 5. Na imagem podemos visualizar o “minimum” que irá guardar o valor de custo menor depois de percorrer toda a lista de “frontier_nodes” e indo sempre comparando valores.

```
#estado atual será o do node selecionado
self.state = node_state
# Remover da fronteira o node que será espandido
self.frontier_nodes.remove(node_expand)
#test
# Verificar que não está na lista de obstacles
if obst[self.state[0]][self.state[1]] == 0:
    print("Node's position (expand):", self.state)
    # Insere na lista de nodes visitadas só depois de confirmar se não existe obstáculo
    self.visited_nodes.insert(node_expand)
    list_visited = []
    for n in self.visited_nodes.getQueue():
        list_visited.append(n.getState())
    # Tenta expandir para todas direções a node
    for dir in ["north","east","west","south"]:
        new_node = self.getNode(node_expand, dir,self.goalNodePos)
        # Verificar que não é repetido (node já visitada)
        if new_node.getState() not in list_visited:
            # Verificar que não está na lista de obstacles
            if obst[new_node.getState()[0]][new_node.getState()[1]] == 0:
                # Insere na fronteira as nodes que poderão ser exploradas(que não são obstáculos visíveis)
                self.frontier_nodes.insert(new_node)
```

Imagem 6. Expansão dos nodes fronteira e visited.

```
def remove(self, elem):
    return self.queue_data.remove(elem)
```

Imagem 6.1. Remover um elemento específico (menor custo) [class Queue]

```
        # Código para mostrar todas possíveis soluções(não tem em conta obstáculos invisíveis)
        # Torna a procura mais demorada, visto que tem que percorrer mais uma vez a lista e dar print das
        # soluções todas(que em alguns casos são muitas(Exponencial))
        """
        # Verifica se exite o goal entre os frontier nodes, ignorando o custo este apresenta todas a soluções
        #  encontradas
        for node in self.frontier_nodes.getQueue():
            if node_state == self.goalNodePos:
                final_node = node
                actual_dir = self.getDirection()
                actual_pos = self.getSelfPosition()
                actual_step = None
                steps = []
                actual_node = final_node
                # Follow from the goal leaf to root...
                while actual_node.getPathCost() != 0:
                    steps.insert(0, [actual_node.getState(), actual_node.getPathCost()])
                    actual_node = actual_node.getParent()
                steps.insert(0, [actual_pos, 0])
                #print("Possível solução", steps)
        """

        # test
        self.printNodesPoderoso("Frontier", self.frontier_nodes, i)
        self.printNodesPoderoso("Visitied", self.visited_nodes, i)
if end == True:
    self.exe(found)
else:
    print("Not Found the GOAL!!!")
input("Waiting for return!")
```

Imagem 7. Código que percorre todos o nodes da fronteira e mostram todas as soluções encontradas para chegar ao Goal; isto faz o software “correr” exponencialmente mais lentamente(quanto maior a procura).

```
def exe(self, final_node=None):
    actual_dir = self.getDirection()
    actual_pos = self.getSelfPosition()
    actual_step = None
    steps = []
    actual_node = final_node
    #Follow from the goal leaf to root...
    while actual_node.getPathCost() != 0:
        steps.insert(0, [actual_node.getState(), actual_node.getPathCost()])
        actual_node = actual_node.getParent()
    steps.insert(0, [actual_pos, 0])

    print("Final Path", steps)

    actions =[]
    fim = False
    i = 0
    print("Length of steps:",len(steps))
    while fim == False:
        actual_step = steps[i]
        next_step = steps[i + 1]
        print("Actual step:", actual_step)
        print("Next step:", next_step)
        next_dir = self.getNextDirection(actual_step[0],next_step[0])
        turns = self.getTurns(actual_dir, next_dir)
        for turn_action in turns:
            actions.append(turn_action)
        actions.append("forward")
        i = i + 1
        if i >= len(steps) - 1:
            fim = True
        else:
            actual_dir = next_dir
    print("Actions:",actions)
    self.c.execute("command", "set_steps")
```

Imagem 8. “Steps” são inseridos ordenadamente e mostrados através do primeiro print da imagem; para criar as actions é tido em conta a direção do agente.

```
    i = -1
    for action in actions:
        # Se a ação for uma de movimento e não de mudar de direção, corresponde a uma ação dos steps
        # Só neste caso, andamos "1" para a frente na lista, porque não queremos percorrer ações de direção
        if action == "forward":
            i += 1
        self.c.execute("command", action)

        # Após ser executada a ação irá percorrer a seguinte função, incluindo a posição atual(actual_pos),
        #  a posição que deveria estar depois de "andar"(steps[i][0], por isso temos aqui o i sendo apenas nas
        #  ações de "andar") e temos a action
        x = self.checkStuff(actual_pos, steps[i][0], action)

        # Se a condição for dirente de False então encontrou-se obstáculo
        if x != False:
            # Assim, irá correr o run denovo, devolvendo o obstáculo encontrado e a posição onde se encontra
            self.run(x, actual_pos)
        actual_pos = self.getSelfPosition()


def checkStuff(self, actual_pos, step, action):
    # Atualizamos a posição atual
    actual_pos2 = self.getSelfPosition()
    # Se a posição atual antes do step for igual à nova, sendo a ação feita de "andar"("forward") e se a
    # posição atual for diferente do step(posição que deveria ter), insere o step como um obstáculo invisível
    if actual_pos == actual_pos2 and action == "forward" and actual_pos2 != step:
        invisible_obstacles = []
        invisible_obstacles.append(step)
        return invisible_obstacles
    else:
        return False
```

Imagem 9. Função criada(checkStuff()) recebe a posição atual antes de ser sido efetuado o “step”, os “steps[i][0]” que remete à posição que deveria ter depois de uma ação(tendo em conta que i é só incrementado se for feita uma ação de movimento e não de mudança de direção) e a action para verificar se foi uma ação de movimento(forward).