Projeto Buscas
CTC-17: Inteligência Artificial
Prof. Paulo André L. de Castro

I. Nome dos autores

- Fernando de Moraes Rodrigues, rodriguesfmr@ita.br
- Igor Amâncio Machado Dias, igoriamd@ita.br
- Lucas do Vale Bezerra, dovale@ita.br

II. Objetivo do Trabalho e Descrição das Implementações

O presente trabalho referente à disciplina *CTC-17: Inteligência Artificial* tem como objetivo aplicar os conhecimentos adquiridos nas aulas teóricas sobre Resolução de Problemas através de Busca de Melhoria Iterativa e sobre Problema de Satisfação de Restrições. Para isso, foram propostos três problemas distintos:

- Encontrar o menor caminho entre duas cidades
- Resolver o jogo Akari
- Encontrar o máximo global de uma dada função

As implementações dos três problemas citados foram feitas utilizando a linguagem Python, no IDE PyCharm, usando bibliotecas conhecidas, como pandas, numpy e math.

III. Resultados Obtidos

Os resultados e implementações de cada item pedido serão descritos nos tópicos a seguir.

A. Descrição da Solução do item 2.1 - Descrição do Arquivo de Dados

1) Implementação do Algoritmo da Busca A:* .
Para a realizar a implementação do algoritmo, necessitou-se construir um base de dados para facilitar a manipulação das informações. Para tal, com as informações passadas, criou-se dois dataframe: um contendo, para cada cidade, seus correspondentes três vizinhos, um em cada coluna (no caso daquelas que possuíam menos que três vizinhos, aplicou-se um zero quando necessário); outra contendo, para cada cidade, a distância para os correspondentes vizinhos, nas respectivas colunas iguais ao dataframe anterior. Para calcular a distância, usou-se da biblioteca geopy.

Com a base de dados pronta, pensou-se em dividir o problema três principais etapas, as quais são:

- Realizar o primeiro encontro de caminho, retornando o caminho, os possíveis caminhos e o peso encontrado
- Para os possíveis caminhos encontrados, checar se chega na cidade final com um peso menor que o peso já encontrado
- Caso ainda exista algum caminho que tenha melhor peso, volta à primeira etapa

Na primeira parte do problema, parte-se da busca a partir da cidade inicial requisitada. Dessa forma, para cada cidade, escolhe-se, dentre seus vizinhos, por meio da heurística daqueles que possuírem o menor valor de $f(Cidade)$. Tal função é calculada somando-se o peso acumulado até a cidade corrente (distância da cidade original até a cidade corrente, seguindo o caminho construído), a distância da cidade corrente para o vizinho analisado e a distância do vizinho até a cidade final. Escolhendo o vizinho, soma-se ao peso acumulado o valor da distância percorrida para o vizinho. Tal processo se repete, até chegar na cidade final, com um peso acumulado, sendo esse o peso da distância percorrida.

Vale salientar que alguns pontos precisam salientados nessa primeira parte:

- Como, no dataframe, algumas cidades ficarem com vizinhos marcados com zero, precisou-se evitar qualquer caso de cidade com ID igual a zero
- Necessita-se evitar que volte para uma cidade que já esteja no caminho. Isso é necessário para evitar que se entre em um loop e não consiga chegar na cidade final
- Além de verificar se a cidade já foi passada no path, também verifica se a cidade está presente em uma lista de cidades negras que não se pode visitar, pois ao visitá-las, acabaria em um caminho sem saída, pois todos os seus vizinhos já estariam no path, não podendo ir para nenhum canto.

A lista das cidades negras são geradas justamente quando se chega em uma situação sem saída. Nesses casos, precisa-se voltar no caminho assumido, retirando o valor do peso, e colocando a cidade na lista de cidade negras e mudando a cidade corrente. Assim, testa-se outros caminhos, voltando na lista até quanto for necessário, já que a lista de cidade negra estará atualizada. Ainda, pode ser que essa cidade saia da lista negra, já que o caminho fica constantemente se alterando. Caso se verifique que ao menos um vizinho dessa cidade negra não esteja no path nem na lista de cidade negras, então pode-se deixá-la disponível.

Terminando a primeira parte, para todos às vezes que foi necessário escolher qual caminho seguir, guarda-se as informações dos outros vizinhos numa lista, contendo: o caminho até chegar nesse vizinho descartado, o vizinho descartado, sua $f()$ e o peso acumulado assumindo que o caminho fosse por ela. Com o peso final, descarta-se todos os casos com $f()$ maior que o peso acumulado final encontrado ao chega na cidade final.

Na segunda parte, usando-se das listas dos possíveis caminhos a se seguir, faz-se o mesmo processo da primeira parte, retirando a análise de guardar outros possíveis caminhos, mas mantendo a construção de caminhos e lista negra. Caso consiga chegar numa cidade final, com um peso acumulado menor que o peso encontrado na primeira tentativa, já tem-se um novo caminho a ser analisado. Tal processo de análise vai para todos os possíveis casos encontrados na primeira tentativa, retirando quando ultrapassa o peso já encontrado.

Já na última parte, existindo um caminho melhor a ser explorado, repete-se o processo passado nas duas primeiras etapas, encontrando o novo caminho, seu novo peso e novos possíveis casos. Para essa última etapa, salienta-se:

- Para facilitar, realiza-se a busca a partir desse vizinho que trouxe essa nova possibilidade. Com isso, para os resultados dos possíveis outros caminhos, é necessário fazer o append do caminho da cidade de origem com esse possível novo caminho, partindo desse vizinho analisado. Isso é feito ao passar a cidade analisada
- Para os possíveis outros caminhos que já existiam, é necessário checar novamente se os seus pesos estão condizentes com o novo peso encontrado. Uma vez checado, junta-se todos os possíveis caminhos

Assim, tem-se todo a parte de analisar os caminhos possíveis. Tal terceira etapa se repete até que não se tenha mais caminhos possíveis, já que o peso acumulado encontrado é menor do que todos as funções $f(ID)$ analisada. Assim, chega-se no menor peso acumulado possível, com o seu respectivo caminho ótimo.

2) Resultados: .
Aplicando o algoritmo A* acima descrito, partindo da cidade Alice Springs (ID 5) e querendo alcançar a cidade Yulara (ID 219), chegou-se no peso final teórico ótimo acumulado de 176811.01 quilômetros, passando pelos seguintes ID de cidade em ordem: [5, 6, 8, 10, 5, 6, 8, 10, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 101, 103, 105, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153, 155, 157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181, 183, 185, 187, 189, 191, 193, 195, 197, 199, 201, 203, 205, 207, 209, 211, 213, 215, 217, 219]. Plotando tais cidades em um gráfico Longitude x Latitude, tem-se o resultado da plotagem a seguir. Com isso, percebe-se o funcionamento do Algoritmo de Busca A*, salientando que é otimamente eficiente.

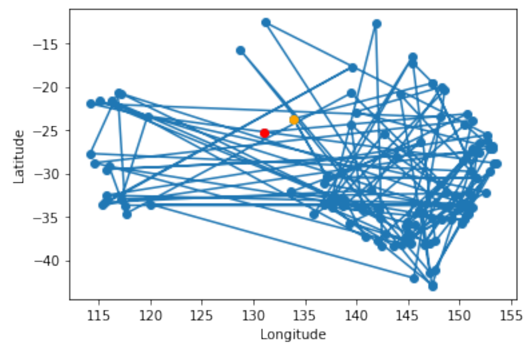


Fig. 1. Caminho ótimo seguido pelas cidades, com cidade de origem em laranja e destino em vermelho.

B. Descrição da Solução do item 2.2 - Light Up

1) Implementação de Satisfação de Restrições: .
Primeiramente, utilizou-se o link disponível para a obtenção de um jogo válido, colocando-o na forma de uma matriz 7x7, conforme exemplo das figuras 2 e 3 a seguir.

Para melhor compreensão, estabeleceu-se a seguinte legenda:

- 0,1,2,3,4: Casas numeradas restringindo a quantidade de lâmpadas adjacentes;
- P: Bloco preto;
- ”: Casa não iluminada;
- L: Lâmpada;
- I: Casa iluminada;
- X: Casa proibida, restrição.

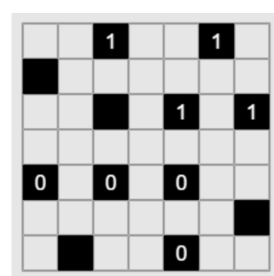


Fig. 2. Exemplo de um jogo gerado pelo link fornecido.

```
[['' '' '1' '' '' '1' '']
 ['P' '' '' '' '' '' '']
 ['' '' 'P' '' '1' '' '1']
 ['' '' '' '' '' '' '']
 ['0' '' '0' '' '0' '' '']
 ['' '' '' '' '' '' 'P']
 ['' 'P' '' '' '0' '' '']]
```

Fig. 3. Jogo na forma de matriz.

A solução será dividida em duas grandes partes. A primeira inicia-se preenchendo o entorno das casas numeradas, quando possível. Percorrendo toda a matriz e utilizando uma função que determina a quantidade de casas vazias e com lâmpadas em volta de uma casa numerada, coloca-se as lâmpadas em torno quando a quantidade de espaços vazios corresponder exatamente à quantidade de lâmpadas faltantes para satisfazer o respectivo número. Por exemplo, se há uma casa numerada com '3' com exatamente três espaços vazios disponíveis, as lâmpadas são colocadas corretamente.

Em seguida, percorre-se novamente a matriz indicando com um 'X' as casas onde não podem ser colocadas as lâmpadas, o que ocorre no entorno de um '0' ou quando o número máximo de adjacências de uma casa numerada foi atingido.

Para finalizar a parte inicial, procura-se casas que só podem ser iluminadas caso seja colocada uma lâmpada nela, como no exemplo da lâmpada do canto inferior esquerdo da figura 4 a seguir.

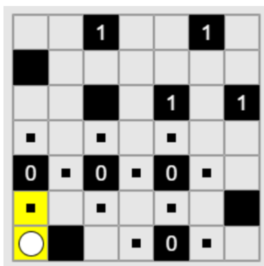


Fig. 4. Exemplo de casa cuja única possibilidade de ser iluminada é colocando uma lâmpada nela.

Os três passos dos parágrafos anteriores devem ser repetidos até que parem de ser colocadas lâmpadas ('L') ou restrições ('X').

Com isso, até o momento temos uma matriz parcialmente preenchida mas que certamente faz parte da solução, pois os caminhos que levaram ao preenchimento até aqui não são flexíveis.

Após esse preenchimento indubitável, deve-se estabelecer uma estratégia de escolher onde devem ser colocadas as novas lâmpadas e retornar caso haja alguma inconsistência, dando início à segunda grande parte da solução.

Foi criada uma função que percorre a matriz, identifica uma casa numerada que ainda não esteja com a quantidade de lâmpadas adjacentes corretas. Escolhe-se uma das casas vazias em seu entorno e coloca uma lâmpada, seguindo a ordem de tentativa cima, direita, baixo, esquerda. Salva-se em uma lista o tabuleiro antes da tentativa, bem como a casa onde foi colocada essa possível lâmpada, em uma segunda lista. Então, faz-se novamente o procedimento referente à primeira grande parte da solução.

Tem-se também uma função que identifica se o tabuleiro está completa e corretamente preenchido. Caso não esteja após o procedimento descrito anteriormente, utiliza-se a lista de tabuleiros para retornar e impede que a tentativa seja feita novamente na mesma casa, utilizando para isso a lista que contém as casas onde supôs-se que haveria uma lâmpada.

O procedimento segue até que a identifique a completude do tabuleiro e o jogo tenha sido finalizado corretamente.

2) Exemplos: Nos exemplos a seguir, na primeira imagem tem-se na matriz de cima o problema a ser resolvido e na matriz de baixo a solução encontrada. Na segunda imagem, tem-se a solução esperada encontrada pelo site fornecido no roteiro. Vale ressaltar que as matrizes encontradas como solução correspondem à matriz esperada para os três exemplos.

- Exemplo 1: Figuras 5 e 6

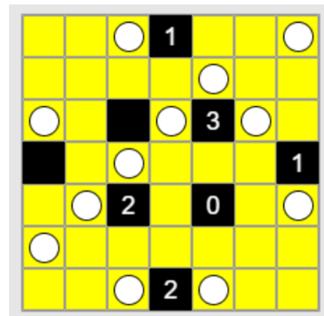
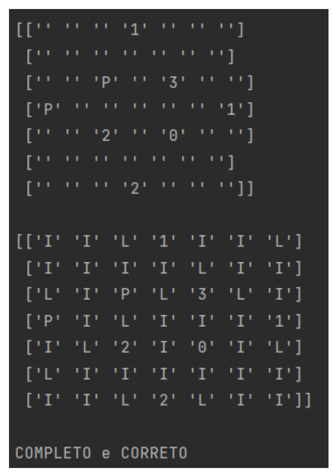


Fig. 5. Solução gerada pelo site para o primeiro exemplo.



```
[['' '' '' '1' '' '' '']
 ['' '' '' '' '' '' '']
 ['' '' 'P' '' '3' '' '']
 ['P' '' '' '' '' '' '1']
 ['' '' '2' '' '0' '' '']
 ['' '' '' '' '' '' '']
 ['' '' '' '2' '' '' '']]

[['I' 'I' 'L' '1' 'I' 'I' 'L']
 ['I' 'I' 'I' 'I' 'L' 'I' 'I']
 ['L' 'I' 'P' 'L' '3' 'L' 'I']
 ['P' 'I' 'L' 'I' 'I' 'I' '1']
 ['I' 'L' '2' 'I' '0' 'I' 'L']
 ['L' 'I' 'I' 'I' 'I' 'I' 'I']
 ['I' 'I' 'L' '2' 'L' 'I' 'I']]

COMPLETO e CORRETO
```

Fig. 6. Primeiro exemplo e solução encontrada.

- Exemplo 2: Figuras 7 e 8

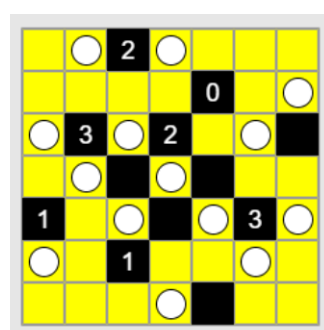


Fig. 7. Solução gerada pelo site para o segundo exemplo.

```
[['' '' '2' '' '' '' '']
 ['' '' '' '' '0' '' '']
 ['' '3' '' '2' '' '' 'P']
 ['' '' 'P' '' 'P' '' '']
 ['1' '' '' 'P' '' '3' '']
 ['' '' '1' '' '' '' '']
 ['' '' '' '' 'P' '' '']]

[['I' 'L' '2' 'L' 'I' 'I' 'I']
 ['I' 'I' 'I' 'I' '0' 'I' 'L']
 ['L' '3' 'L' '2' 'I' 'L' 'P']
 ['I' 'L' 'P' 'L' 'P' 'I' 'I']
 ['1' 'I' 'L' 'P' 'L' '3' 'L']
 ['L' 'I' '1' 'I' 'I' 'L' 'I']
 ['I' 'I' 'I' 'L' 'P' 'I' 'I']]

COMPLETO e CORRETO
```

Fig. 8. Segundo exemplo e solução encontrada.

- Exemplo 3: Figuras 9 e 10

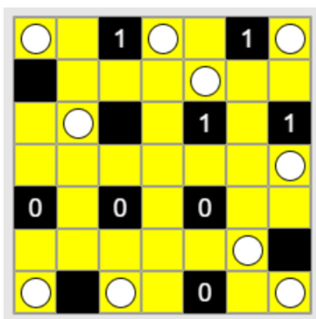


Fig. 9. Solução gerada pelo site para o terceiro exemplo.

```
[['' '' '1' '' '' '1' '']
 ['P' '' '' '' '' '' '']
 ['' '' 'P' '' '1' '' '1']
 ['' '' '' '' '' '' '']
 ['0' '' '0' '' '0' '' '']
 ['' '' '' '' '' '' 'P']
 ['' 'P' '' '' '0' '' '']]

[['L' 'I' '1' 'L' 'I' '1' 'L']
 ['P' 'I' 'I' 'I' 'L' 'I' 'I']
 ['I' 'L' 'P' 'I' '1' 'I' '1']
 ['I' 'I' 'I' 'I' 'I' 'I' 'L']
 ['0' 'I' '0' 'I' '0' 'I' 'I']
 ['I' 'I' 'I' 'I' 'I' 'L' 'P']
 ['L' 'P' 'L' 'I' '0' 'I' 'L']]

COMPLETO e CORRETO
```

Fig. 10. Terceiro exemplo e solução encontrada.

C. *Descrição da Solução do item 2.3 - Melhoria Iterativa*

1) Visualização: Inicialmente, ao se deparar com a função $f(x, y) \in \Re^3$ abaixo, foi necessário visualizá-la graficamente para se obter uma melhor visão dos máximos locais e globais presentes na função.

$$f(x,y) = 4\,e^{-(x^2+y^2-2(x+y-1))} + e^{-((x-3)^2+(y-3)^2)} + e^{-((x+3)^2+(y-3)^2)} + e^{-((x-3)^2+(y+3)^2)} + e^{-((x+3)^2+(y+3)^2)}$$

Fig. 11. Definição da função f(x,y).

Dessa forma, obteve-se a seguinte figura:

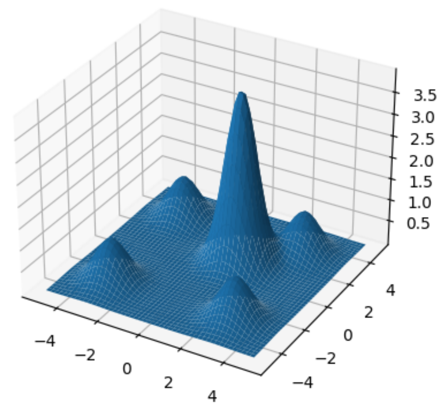


Fig. 12. Visualização gráfica da função f(x,y).

2) Implementação: A partir da visualização, visou-se uma melhor ambientação aos algoritmos do problema e ao entendimento da Busca por Melhoria Iterativa. Dessa forma, baseando-se no pseudo-código do Hill-Climbing apresentado em aula, implementou-se o algoritmo e foi possível ver que a cada vez que o algoritmo executava, um máximo local diferente era descoberto, mas pouquíssimas vezes o máximo global.

Por isso, mostrou-se necessário o aprimoramento por meio da implementação do algoritmo Simulated Annealing, também apresentado em aula. Utilizando-se de algumas referências, bem como da lógica de pseudo-código apresentada pelo Professor, foi possível obter os resultados abaixo. Nesse algoritmo, considerou-se um decaimento na temperatura de 1% a cada iteração.

3) Resultados: Inicialmente, com 1000 iterações e temperatura inicial $T = 4$, obteve-se o resultado abaixo:

Além disso, foram testados maiores e menores números de iterações, bem como diferente valores de temperatura inicial.

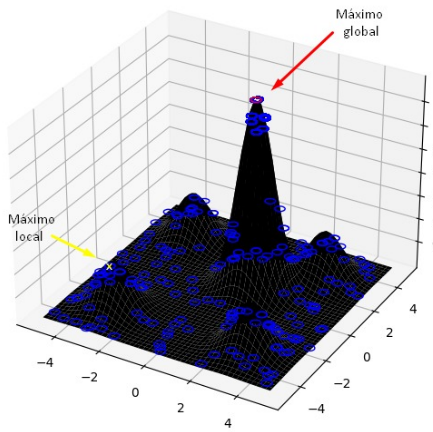


Fig. 13. Resultado para 1000 iterações e temperatura inicial $T = 4$.

O algoritmo mostrou-se eficiente para encontrar o máximo global em todos os casos, bem como para encontrar alguns máximos locais no percurso. Em certos casos, para números grandes de iterações e altos valores de temperatura inicial, o algoritmo encontra todos os máximos locais e o máximo global.

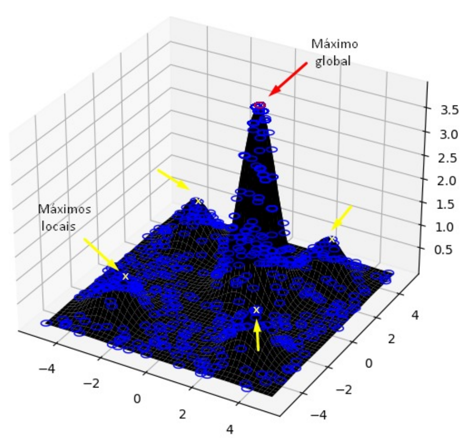


Fig. 14. Resultado para 10000 iterações e temperatura inicial $T = 100$.

IV. Conclusões

Os três problemas abordados no presente trabalho elucidaram bem ao grupo a aplicação das técnicas aprendidas nas aulas teóricas, principalmente por serem desafiadores e exigirem soluções criativas e distintas. Foram problemas que demandaram um certo tempo e bastante raciocínio dos integrantes, o que fez com que a experiência fosse bastante engrandecedora para nosso desenvolvimento acadêmico, além de proporcionar grande satisfação ao encontrar a solução esperada.

Referências

[1] Departamento de Computação – ITA, Prof. Paulo André L. de Castro, Buscas de Melhoria Iterativa - Vídeo Aula e Slides de CTC-17

[2] Simulated Annealing with Python. [Online] Disponível: https://www.youtube.com/watch?v=XNMGq5Jjs5w

[3] Aula 6 - Meta-heurísticas (Simulated Annealing). [Online] Disponível: https://www.youtube.com/watch?v=4VGt0jN73fc