Inteligência Artificial

Thiago Henrique Leite da Silva, RA: 139920

Aula 05 - Exercício prático Busca informada (Prática)

1) (1,0) Pesquise um algoritmo empregando A* num problema real (pode ser um jogo como xadrez, quebra cabeça de 8 peças, circuito de robô, percurso em mapa, etc.)

- Submeta o código no classroom.

- Cite a fonte utilizada, link e autor onde você pegou o código.

- Num arquivo pdf descreva o funcionamento do algoritmo, a lógica usada pelo autor para resolver o problema com A*, compile o código e insira o print da saída gerado

O código escolhido que implementa a busca A* foi este aqui:

- https://github.com/C-Collamar/8-Puzzle-Solver

Este código é uma implementação que resolve o famoso problema do quebra-cabeça de 8 peças, o desenvolvedor do código fez algo bem interessante, a implementação da solução com a busca A* e também com a busca em largura, imprimindo ao final o desempenho de cada uma, nos mostrando dados como o número de comparações, memória utilizada e tempo de execução.

O código funciona da seguinte maneira, de início, inserimos a posição inicial em que queremos que as peças se encontrem, depois, inserimos qual o resultado queremos obter. Após essas atribuições, o script calcula a solução com a busca A estrela e depois com a busca em largura, ao final, imprime os resultados obtidos.

Focando no funcionamento da busca A*, ela funciona da seguinte maneira:

De início, passamos os estados inicial e final fornecidos pelo usuário para a função.

A heurística definida pelo programador foi a distância que um vértice está do seu estado final, sendo definida por um valor inteiro, onde quanto menor ele é, mais próximo o vértice está de sua posição correta. A função que determina a heurística de um nó é a ***manhattanDist***, que calcula a distância de Manhattan do nó até seu destino; a distância de Manhattan funciona muito bem para movimentos horizontais e verticais, como no problema, portanto é uma heurística admissível.

```c
int manhattanDist(State * const curr, State * const goal) {
    int x0, y0; //used for indexing each symbol in `curr`
    int x1, y1; //correspoinding row and column of symbol from curr[y0, x0] at `goal`
    int dx, dy; //change in x0 and x1, and y0 and y1, respectively
    int sum = 0;

    //for each symbol in `curr`
    for(y0 = 0; y0 < 3; ++y0) {
        for(x0 = 0; x0 < 3; ++x0) {
            //find the coordinates of the same symbol in `goal`
            for(y1 = 0; y1 < 3; ++y1) {
                for(x1 = 0; x1 < 3; ++x1) {
                    if(curr->board[y0][x0] == goal->board[y1][x1]) {
                        dx = (x0 - x1 < 0)? x1 - x0 : x0 - x1;
                        dy = (y0 - y1 < 0)? y1 - y0 : y0 - y1;
                        sum += dx + dy;
                    }
                }
            }
        }
    }

    return sum;
}
```

Sabendo disso, agora podemos começar inserindo nosso nó raiz na fila, que será o elemento da posição (0,0) da matriz inicial pré-determinada pelo usuário.

Enfim começamos a busca, enquanto tivermos nó para expandir, vamos desenfileirando eles da fila e verificando se ele está em sua posição final, lembrando que cada nó tem uma posição final específica de acordo com o que foi determinado.

Caso ele esteja na posição correta, passamos para o próximo, caso contrário, expandimos seus adjacentes e adicionamos eles na mesma fila que estávamos olhando anteriormente só que na posição ordenada crescentemente pelo custo, para assim pegarmos sempre o menor custo primeiro.

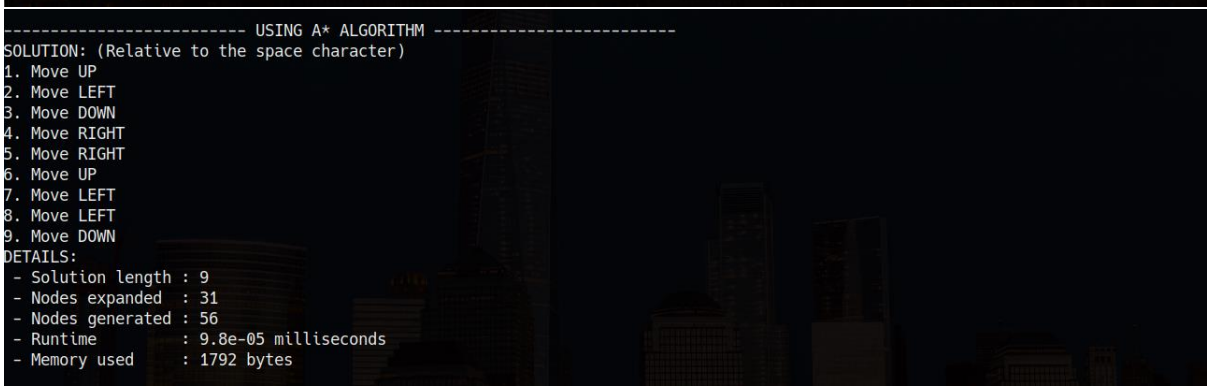
```
SolutionPath* AStar_search(State *initial, State *goal) {
    NodeList *queue = NULL;
    NodeList *children = NULL;
    Node *node = NULL;

    //start timer
    clock_t start = clock();

    //initialize the queue with the root node of the search tree
    pushNode(createNode(0, manhattanDist(initial, goal), initial, NULL), &queue);
    Node *root = queue->head->currNode; //for deallocating generated tree

    //while there is a node in the queue to expand
    while(queue->nodeCount > 0) {
        //pop the last node (tail) of the queue
        node = popNode(&queue);

        //if the state of the node is the goal state
        if(statesMatch(node->state, goal))
            break;

        //else, expand the node and update the expanded-nodes counter
        children = getChildren(node, goal);
        ++nodesExpanded;

        //add the node's children to the queue
        pushListInOrder(&children, queue);
    }

    //determine the time elapsed
    runtime = (double)(clock() - start) / CLOCKS_PER_SEC;

    //get solution path in order from the root, if it exists
    SolutionPath *pathHead = NULL;
    SolutionPath *newPathNode = NULL;

    while(node) {
        newPathNode = malloc(sizeof(SolutionPath));
        newPathNode->action = node->state->action;
        newPathNode->next = pathHead;
        pathHead = newPathNode;

        //update the solution length and move on the next node
        ++solutionLength;
        node = node->parent;
    }

    --solutionLength; //uncount the root node

    //deallocate the generated tree
    destroyTree(root);

    return pathHead;
}
```

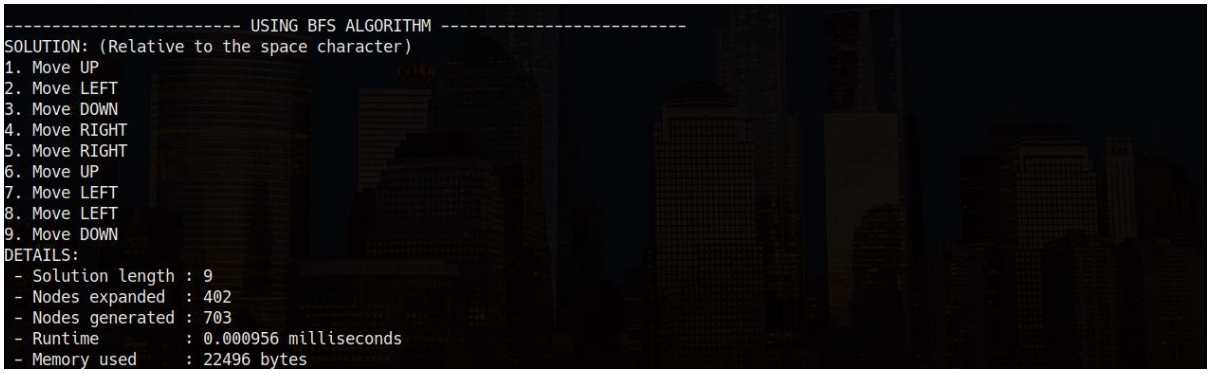
Após o laço principal, as ações que deveriam ser tomadas são gravadas em uma lista, que é retornada ao usuário.

Em resumo, a forma com que o desenvolvedor encontrou para aplicar a busca A* no problema foi utilizando a heurística como uma forma de organizar os nós em uma fila de prioridade, onde os nós

com menor custo, ou seja, mais pertos do seu objetivo, tem preferência na hora de serem expandidos.

Código em execução:





Após analisar os resultados obtidos, podemos perceber que a busca A* foi extremamente melhor do que a BFS, tanto em memória quanto em tempo de execução, além de ter expandido muito menos nós do que o algoritmo A*.

O que nos mostra o quão poderoso é utilizarmos uma heurística admissível na resolução de um problema.

O autor do código explicou mais a fundo os resultados que obteve neste PDF:

- https://github.com/C-Collamar/8-Puzzle-Solver/blob/master/8-Puzzle%20Solver%20Documentation%20and%20Report.pdf

Achei bem interessante, vale a pena dar uma olhada professora!