FUNDAÇÃO GETÚLIO VARGAS

MATHEUS DE MONCADA ASSIS

TRABALHO DE ESTRUTURA DE DADOS E ALGORITMOS

SOLUÇÃO PARA O SLIDING-PUZZLE PELO ALGORITMO A* SEARCH

RIO DE JANEIRO - RJ
2018

MATHEUS DE MONCADA ASSIS

TRABALHO DE ESTRUTURA DE DADOS E ALGORITMOS

Relatório do trabalho junto ao Curso
de Estrutura de Dados e Algoritmos
da EMAp-FGV como requisito para
conclusão do curso do 3º período.

Professor: Alexandre Rademaker
Monitor: Henrique Muniz

RIO DE JANEIRO – RJ
2018

AUTO AVALIAÇÃO NO CURSO

Meu nome é Matheus Assis, tenho 19 anos e fui aluno no curso de Estrutura de Dados e Algoritmos durante o semestre. Durante esse período, fui capaz de conhecer diversas noções de computação que nunca me foram apresentadas, além de ter sido capaz de criar e programar diversas ideias interessantes e que podem ser usadas para diversos códigos durante minha vida. Desse modo, é importante ressaltar a maneira como o curso foi levado durante esse tempo, para que fique claro todo o trabalho realizado.

Durante os primeiros meses do ano, muitos estigmas foram colocados sobre o curso – seja por veteranos, seja pela maneira como o curso foi apresentado –, fazendo com que um certo medo fosse instaurado. Entretanto, durante o decorrer, tal medo passou a se tornar um desafio maior, fazendo com que eu dedicasse algumas horas a mais para estudo do curso. De maneira mais específica, no início minhas horas dedicadas eram bem próximas de uma hora por semana, sem ser o material prévio, mas após alguns meses, passou a ser de duas horas ou mais, com leitura prévia e busca por outras vertentes do curso.

Outra situação recorrente durante o semestre foi a realização das listas, que, em diversos momentos, se mostrou um desafio. É importante evidenciar a dificuldade das mesmas no começo, em que nós estávamos com pouca noção da própria linguagem – Common Lisp, que antes não fora apresentada à nossa turma –, e com a alocação de tempo das matérias. Entretanto, durante o decorrer, tais listas se adaptaram às nossas demandas, fazendo com que elas se tornassem mais produtivas.

No decorrer do semestre, utilizei do livro-texto da matéria para o estudo, além de utilizar de muitas pesquisas na internet pelo próprio site do Common Lisp – http://clhs.lisp.se/. Diante disso, encontrei uma falta de informações sobre diversos problemas que surgem durante o código, o que fez com que a insatisfação aumentando. Encarando algumas listas mais difíceis que o normal, com uma carga grande de exercícios, acredito que o curso ficou sobrecarregado em diversos momento, gerando lacunas em outros.

Portanto, acredito que um dos pontos mais difíceis do curso foi a alocação do tempo com as demandas excessivas em determinados momentos do semestre. Entretanto, o aprendizado da matéria foi enriquecedor, acrescentando muito, não somente ao currículo, mas às minhas habilidades de programação. Em suma, o curso foi um dos que mais me rendeu frutos, mas ainda tem alguns pontos negativos que podem ser superados.

AVALIAÇÃO DO CURSO

De maneira geral, o curso apresentou matérias interessantes, enriquecedoras e de fácil acesso. Entretanto, deve-se reiterar a falha do livro em apresentar os pseudocódigos, tendo em vista que esses não ajudavam na programação do algoritmo selecionado.

Creio que uma das mudanças pode ser um modelo com mais trabalhos, principalmente em dupla, tendo em vista o quão enriquecedor são. Por mais que seja necessário ter um controle do aprendizado pelo aluno, é importante também ter meios de melhorar o ensino. Um trabalho é capaz de melhorar a aceitação de ideias distintas para a abordagem de um problema, além de motivar mais ainda os alunos, já que eles podem alcançar resultados maiores.

Logo, é de suma importância ter em mente como o último trabalho foi marcante no semestre. Pessoalmente, foi um dos trabalhos que mais me deu aprendizado, e que mais me fez querer aprender o que fora apresentado em sala de aula. O uso de listas é interessante, mas não é tão interativo quanto um trabalho, que utiliza os algoritmos que aprendemos durante as listas.

ESCOLHA DO PROJETO

O tema escolhido pelo meu trabalho foi o Sliding-Puzzle Solver. Eu e Vinícius tínhamos em mente um modelo que poderia nos ajudar a criar tal código e durante conversas, concluímos que esse era o trabalho que queríamos criar.

Durante a criação do código, encontramos alguns sites – alguns bons, outros nem tanto – de pesquisa. Dentre eles, o mais importante era a documentação dada pelo "http://www.cs.princeton.edu/courses/archive/spring18/cos226/assignments/8puzzle/index.html". Além dele, também acessamos os sites de consulta do Common Lisp, "http://clhs.lisp.se/", e um site para solucionar um problema encontrado de inversões dentro de uma matriz https: "//www.geeksforgeeks.org/counting-inversions/". Além dos sites, também tivemos que estudar o algoritmo A* search para realizar a busca ótima.

As partes mais fáceis do nosso projeto consistiram na criação de funções que dependiam exclusivamente de parâmetros simples, como um array. Dessa maneira, funções como o hamming-dist e is-goal podem ser caracterizadas como desse tipo. Entretanto, algumas outras funções foram um tanto quanto complicadas, principalmente para visualização do que acontecia dentro delas, tais como o is-solvable e o manhattan-dist – o segundo um pouco mais simples, mas de difícil visualização no começo.

Eu e Vinícius dedicamos tempos similares, tendo em vista que passamos a maior parte do tempo realizando o projeto juntos. Diria que passamos três/quatro dias criando o código, enquanto dois desses dias foram de trabalho contínuo na própria FGV. Durante os dias na FGV, chegamos de manhã na faculdade e só parávamos de programar no meio/final da tarde.

Para solucionar o problema, começamos pensando em tudo que precisaríamos e criamos a função solve. A partir dela, passamos a criar outras funções, que determinavam outras partes do problema para que, enfim, alcançássemos o resultado desejado. Vale ressaltar que no início, tínhamos em mente a montagem de um código que fosse apresentado em formatos de matrizes nxn, mas durante a realização dele, percebemos que trataríamos alguns problemas de forma trivial caso utilizássemos uma matriz $1xn^2$.

CÓDIGO

```
(ql:quickload :cl-heap)
  (def class board()

        ...)
```

Primeiramente, percebemos que deveríamos ter uma classe capaz de armazenar o estado atual do jogo, juntamente com o jogo anterior à ele. Além disso, precisávamos armazenar o peso do jogo, o número de movimentos até então, a peça movida – uma das últimas coisas que adicionamos na classe –, e a posição do zero dentro do jogo. Além disso, também chamamos a biblioteca cl-heap, que nos fornecerá a cl-heap:priority-queue, ou fila de prioridades.

```
(defun is-goal (board-array)
                ...)
(defun unroll (board-obj &optional mov res)
                ...)
```

A primeira função – is-goal – é responsável por checar se o jogo atual é o jogo solução, enquanto a segunda – unroll – nos auxilia na hora de responder o problema. Ambas as funções são simples, e a complexidade de ambas é O(n).

```
(defun hamming-dist (board-array)
                ...)
(defun manhattan-dist (board-array)
                ...)
```

Ambas as funções são necessárias para a determinação de um peso para o jogo. Enquanto a Hamming Distance aumenta em um o peso para cada peça fora do lugar correto, a Manhattan Distance aumenta o peso no número de posições de distância do lugar correto. Dessa maneira, o Manhattan Distance é o meio mais eficiente de se alcançar uma resposta, já que utilizamos uma fila de prioridades para a observação dos estados do jogo. Além disso, a complexidade de ambas é O(n).

```
(defun make-move (board-obj move)
                        ...)
    (defun is-granparent (parent-obj child-obj)
                        ...)
(defun enqueue-child (queue board-obj move function)
                        ...)
  (defun gen-neighbors (board-obj queue function)
                        ...)
```

Essas funções são as responsáveis por gerar todos os jogos vizinhos – ou filhos – e colocar dentro da fila de prioridade que criamos. Vale dizer que a função is-granparent garante uma otimização crítica ao nosso código, fazendo com que não visitemos o jogo, caso ele seja o avô do atual – pai do pai do jogo atual. Nesse caso, como elas estão conectadas, sua complexidade está em O(n), enquanto a is-granparent é O(1).

```
(defun is-solvable (board-array)
                ...)
```

Uma das funções mais complicadas que criamos. Ela se baseia em determinar, antes mesmo de começarmos a solucionar o problema, se é solucionável ou não. Para isso, utilizamos uma técnica em que, para matrizes nxn com n ímpar, como o número de inversões realizados nela é invariante, caso tenhamos um número par de inversões, ela é solucionável. Caso contrário, não. Para valores n pares, temos que determinar o número de inversões, somado ao número da linha em que o zero (0) está presente na matriz. Caso esse número seja ímpar, temos uma proporção invariante e ela é solucionável. Caso contrário, não.

Tal código é $O(n^2)$, sendo o mais trabalho, mas realizado apenas uma vez durante todo o problema, já que se estamos em um estado inicial, qualquer estado que alcançamos a partir dele poderá fazer o caminho de volta, garantindo que se é solucionável no início, será sempre solucionável.

É válido explicar que o número de inversões que estamos contado é, para todos os valores da matriz ordenados linearmente – esse foi um dos motivos principais para termos mudado a formatação do nosso problema –, colocamos nosso ponteiro na primeira caso e obtemos um valor i. Caso esse valor i esteja a frente dos valores menores que i e diferentes de zero, a quantidade desses valores será armazenada na quantidade de inversões.

```
(defun solve-aux ()
       ...)
 (defun rec-ans ()
       ...)
  (defun solve ()
       ...)
```

Finalmente, as últimas funções e, consequentemente, as que determinam todo o processo. Para o funcionamento da solve, criamos a fila de prioridades com peso de acordo com a função hamming-dist ou manhattan-dist, o jogo inicial e passamos a chamar a solve-aux, caso o sistema seja solucionável. Dentro da solve-aux, chamamos as funções importantes, retirando o primeiro elemento da fila de prioridades e passando como parâmetro para essas funções.

Para fins de apresentação de resposta, a rec-ans é responsável por mostrar os resultados na tela de maneira bem acessível, com todos o caminho percorrido até a solução, juntamente com o número de movimentos necessário e a ordem das perças a serem movidas no tabuleiro.

Tais funções, por dependerem de outras, carregam a complexidade delas, mas não acrescentam complexidades, por não utilizarem loops e afins. A única que aumentará a complexidade é a rec-ans, que aumentará a complexidade de acordo com o número de movimentos.

Rodando o código, receberemos o resultado caso seja solucionável, e “Unsolvable” caso contrário.

MANUAL DO USUÁRIO

Seja muito bem-vindo!

É com muito prazer que venho apresentar à você como utilizar o código solve, capaz de resolver qualquer sliding-puzzle, um brinquedo antigo, mas que se tornou popularmente conhecido durante os tempos.

Para isso, basta utilizar nosso código, carregar a biblioteca cl-heap e passar como parâmetros da função solve, a lista do que você pretende resolver. Ou seja, caso seu jogo esteja nesse formato:

Passaremos a lista na forma **‘(0 1 3 4 2 5 7 8 6)**

Ademais, temos um outro parâmetro! Esse deverá ser escrito em um desses formatos: #’manhattan-dist ou #’hamming-dist. Vale ressaltar que esses dois métodos definem a velocidade com que o programa irá rodar, sendo o Manhattan Distance mais eficiente e, consequentemente, mais rápido. Para 15-puzzles ou maiores, recomendamos a utilização do Manhattan.

No final, dentro da IDE escolhida por você, a chamada será apresenta dessa maneira, utilizando o exemplo da lista acima:

```
(solve ‘(0 1 3 4 2 5 7 8 6) #’manhattan-dist)
```

Sua reposta será gerada e irá conter todos os estados do jogo, juntamente com o número de movimentos e, por fim, as peças que você deve mover para alcançar a resposta.

Muito obrigado por conhecer nosso código!

Criadores: Matheus Assis e Vinícius D’Avila