Vinícius D'Avila Barros

Relatório referente à disciplina Estrutura de Dados e Algoritmos, do curso de Matemática Aplicada da FGV-RJ, ministrada pelo professor Alexandre Rademaker.

Rio de Janeiro

Junho de 2018

1. Autoavaliação

1.1. Quanto tempo por semana você dedicou para estudo do curso?

Durante a curso dediquei-me regularmente a atividades relacionadas aos temas apresentados, diria oito horas por semana, buscando estudar diferentes linguagens, implementar alguns algoritmos e solucionar problemas interessantes, mas poucas dessas horas, talvez duas, dedicadas especificamente aos problemas exigidos pelo curso e à programação em Common Lisp.

1.2. Você se preparava antes das aulas lendo o material que seria apresentado?

Meu estudo para a disciplina foi feito tipicamente atualizando-me à matéria já lecionada, ou seja, raras vezes me adiantei ao que seria ministrado na aula seguinte.

1.3. Como foi sua estratégia para fazer os projetos?

Buscamos planejar todos as funções necessárias já de início, de modo que cada uma realizasse uma tarefa bem específica, começando pela função que verifica se o objetivo do jogo foi atingido, depois implementando o movimento das peças e assim por diante.

1.4. Que material você usou para estudo?

O livro *Algorithms* de Papadimitriou, Dasgupta, e Vazirani, junto à consulta da Wikipédia inglesa em geral foram suficientes para a compreensão da parte teórica da matéria. Já na parte de

implementação e aprendizado de Common Lisp, *Practical Common Lisp* foi a melhor referência e mais frequentemente consultada.

1.5. Quais foram as partes mais dificeis do curso?

As duas maiores dificuldades do curso foram lidar com o péssimo ambiente de desenvolvimento *LispWorks*, e também com a carência de uma comunidade ativa programando em Lisp, se comparado a outras linguagens como o Python ou C++, o que dificulta na hora de pesquisar pelas soluções de erros (seja de compilação ou de execução) aparentemente comuns, mas que mesmo assim não possuem perguntas ou respostas satisfatórias em fóruns online e sites *Q&A* como o *StackOverflow*.

1.6. Quais foram as partes mais fáceis do curso?

A parte mais fácil foi o projeto em grupo para a A2. Não que tenha sido fácil, mas porque gerou um resultado final acabado de autoria própria, depois de tantas horas programando e tantas linhas de código, o que me deixou bastante satisfeito, e olhando para trás diria que foi divertido fazê-lo.

2. Avaliação do curso

2.1 O que você mais gostou sobre o curso?

Os melhores aspectos do curso são para mim o nível de abstração promovido para compreender a complexidade dos algoritmos, classes de problemas, paradigmas de programação, e além disso as discussões sobre como programar de maneira a tornar seu código mais legível, expansível, e adaptável, trazendo conceitos como compartimentalização e modularidade.

Vale registrar que antes de ingressar na disciplina havia tentado aprender programação funcional e tive bastante dificuldade. Porém logo nas duas ou três primeiras semanas do curso as soluções recursivas e por divisão e conquista para diversos problemas passou a fazer bastante sentido para mim, e inclusive em muitos casos vejo-as com mais naturalidade do que soluções procedurais.

2.2 O que você gostaria que fosse mudado no curso?

Acredito que valeria a pena mudar a linguagem do curso para Haskell ou Racket, por serem mais modernas e na minha opinião de acesso (em termos de IDE, comunidades online, etc.) um pouco mais fácil.

3. Projeto

O projeto foi realizado em dupla, com o aluno Matheus Assis, solucionando o *Sliding Puzzle*, isto é, dado um tabuleiro n x n com ($n^2 - 1$) peças numeradas de 1 a ($n^2 - 1$) em ordem qualquer, e sejam movimentos legítimos apenas aqueles que “arrastam” uma peça vizinha para a casa vazia do tabuleiro, retornamos todos os movimentos necessários para que essas peças sejam dispostas em ordem numérica.

3.1. Por que você escolheu o tema que escolheu?

A escolha do tema se deu principalmente por um interesse maior que compartilhamos por desafios envolvendo jogos de tabuleiro (notavelmente, o xadrez).

3.2. O que você precisou ler e estudar para implementar o projeto?

Os materiais utilizados foram o 8-puzzle assignment do curso *Algorithms and Data Structures* da Universidade de Princeton, o artigo da Wikipédia inglesa sobre o algoritmo A* search, a biblioteca CL-HEAP, além do livro *Practical Common Lisp*.

3.3. Quais foram as partes mais complicadas? Quais as mais fáceis?

O mais complicado foi lidar com problemas de falta de memória inicialmente, porque havíamos definido a fila de prioridade, denominada *game-tree*, como uma variável global, mas depois da optimização do *is-grandparent* (brevemente descrita mais adiante) e da limitação do escopo da *game-tree* para dentro da função solve, deixamos de ter dificuldades maiores.

3.4. Quanto tempo você dedicou ao projeto? Como você 'atacou' o problema?

Estimo que o tempo total gasto puramente programando, debugando e pensando na solução concentra, ou seja, não contabilizando as horas para redigir este relatório, tenha sido no total aproximadamente 20 horas para cada aluno da dupla. O item 1.3. explica nossa abordagem para solucionar o problema.

3.5. Descrição da 'arquitetura' da solução. Que funções foram implementadas e por que? Dar a complexidade de cada uma delas.

A solução do desafio envolve a implementação do algoritmo *A* search*. Para compreendê-lo podemos começar pensando em qual seria a solução ingênua do problema: gerar uma árvore de tabuleiros de forma que para cada movimento um novo tabuleiro seja acrescentado até que eventualmente o tabuleiro cujo estado é o desejado (ou seja, todas as peças são dispostas em ordem numérica) seja atingido. Devido ao fator de ramificação, isso seria muito pouco viável, tanto em virtude do tempo quanto da memória gasta.

Ao usar o *A* search*, o que faremos é, também, gerar uma árvore e um conjunto de tabuleiros a partir do inicial (que serão no mínimo 2 e no máximo 4), porém associar um peso a cada tabuleiro gerado, e com esse peso associado inserir os tabuleiros em uma fila de prioridade. Depois disso, fazemos o *dequeue*, o que nos retornará o tabuleiro de menor peso, e assim inseriremos na fila de prioridade os filhos apenas desse tabuleiro. Durante todo o processo, que se repete até atingirmos o estado desejado, apenas os filhos dos tabuleiros que tem o menor peso dentro da fila de prioridade são efetivamente criados, o que é bem mais econômico do que a solução de busca exaustiva em largura (que denominamos de ingênua).

A complexidade do algoritmo é dada em função do fator de ramificação *r* e da profundidade *p* da solução, isto é, do número de movimentos necessários para alcançar o estado final, na forma $O(r^p)$. Não é trivial estimar o fator de ramificação devido às diversas heurísticas e optimizações possíveis para cada implementação; no pior dos casos, para o nosso problema do *Sliding Puzzle*,

sem contar as optimizações nosso *r* seria aproximadamente igual a 3 (média dos números máximo e mínimo de filhos por tabuleiro), e *p* depende do tamanho do tabuleiro.

Para representar o tabuleiro, criamos uma classe chamada *board*, cujo atributo principal é *state*, onde armazenamos um *array* de tamanho n^2 que indica as peças e suas posições, e onde o número 0 representa uma casa vazia. *Father* é um atributo que recebe um outro objeto da classe board, de onde aquela instancia originou-se, útil para quando chegarmos ao estágio final do algoritmo e quisermos saber quais foram os movimentos necessários para chegar até lá. *Distance* e *movecount* estão relacionados à determinação de um peso ao inserir o tabuleiro na fila de prioridade, *movecount* simplesmente guarda o número de movimentos feitos para chegar até aquele estado. *Piece* guarda qual peça foi movida em relação ao tabuleiro-pai para chegar a tal estado, e finalmente *zeropos* indica a posição da casa vazia no tabuleiro.

Existem duas funções de possível uso para auxiliar no cálculo do peso. A mais simples é a *hamming-dist*, que simplesmente soma o número de peças na posição errada. Já a *manhattan-dist* soma as distâncias verticais e horizontais de cada peça em relação à posição correta no tabuleiro. O primeiro método não leva em consideração o quão longe uma peça está de sua posição ideal, por isso é bem menos eficiente. O valor da prioridade a ser usado é justamente a soma dos atributos *movecount* e *distance*, sendo que a última pode ser resultado da *hamming-dist* ou *manhattan-dist*, a ser passada como argumento da função principal *solve*.

Ao chamarmos a função solve com um estado inicial, antes de tudo decidimos, em *is-solvable*, se tal estado é sequer passível de solução. Para isso precisamos considerar o número de inversões presentes no tabuleiro, isto é, o número de peças *i* e *j* tais que $i < j$ mas a peça *i* aparece depois do *j* na representação linear do tabuleiro (justamente aquela utilizada no atributo *state*, com um *array* de tamanho n^2). A complexidade temporal da contagem do número de inversões é $O(n^4)$, porém só é computada uma vez no início do programa e satisfaz as exigências do *assignment*.

Tendo armazenado o número de inversões em um tabuleiro, temos que separar a solubilidade do problema em dois casos, quando o tabuleiro tem dimensões ímpares ou pares. Quando é ímpar, então cada movimento muda o número de inversões por um número par, e já que queremos que o número de inversões seja 0, então necessariamente, para que o tabuleiro seja resolvível, então já desde o início deve haver um número par de inversões. Já se o tabuleiro tem dimensão par, então temos que considerar também a linha em que a casa vazia está presente e adicionar ao número de inversões, verificar se é par para então decidir a solubilidade.

Uma vez concluído que o tabuleiro é resolvível, geramos seus filhos em *gen-children*, calculamos seu peso/prioridade e enfileiramos cada um deles na *game-tree*. Fazemos o *dequeue*, que nos dá o tabuleiro com menor peso, verificamos se esse é justamente o tabuleiro cujas posições são as corretas, e novamente enfileiramos os filhos do tabuleiro que foi retirado, exatamente como na descrição do *A* search* previamente. Porém com uma optimização, checamos em *is-grandparent* se o estado do filho não é o mesmo que o do pai de seu pai, se for o caso, sequer enfileiramos o objeto, de modo a evitar desperdício de tempo, memória, e talvez até problemas com loops infinitos.

Uma vez alcançado o estado final, chamamos a função *unroll*, que recursivamente percorre os atributos *father* e *piece* dos tabuleiros até chegar no início da *game-tree*, o que nos dá cada movimento realizado para solucionar o *puzzle*.

3.6 Manual do usuário: como usar seu código, passo-a-passo que você daria para sua avó executar o sistema que você implementou! ;-)

Uma vez que tenha-se baixado e compilado a biblioteca CL-HEAP, preferencialmente com o uso do quicklisp, para testar é necessário representar o tabuleiro linearmente como uma lista, de modo que o quadro a seguir seja escrito na forma ‘(0 1 3 4 2 5 7 8 6).

Também é necessário escolher entre *hamming-dist* ou *manhattan-dist*. Feito isso, basta salvar, compilar e carregar o arquivo *sliding-puzzle.lisp*, e chamar a função *solve* da forma:

CL-USER 10 : 1 > (solve '(0 1 3 4 2 5 7 8 6) #'manhattan-dist)

#(1 0 3 4 2 5 7 8 6)

#(1 2 3 4 0 5 7 8 6)

#(1 2 3 4 5 0 7 8 6)

#(1 2 3 4 5 6 7 8 0)

Número de movimentos: 4

Ordem de peças a serem movidas: (1 2 5 6)

Uma lista de testes com tabuleiros de diferentes dimensões foi fornecida no repositório do projeto no Github, dentro do arquivo *testes.lisp*.