Relatório do Trabalho de Programação III – Minesweeper

Trabalho realizado pelos alunos:

Jorge Couto	Nº58656	IACD
Miguel Grilo	Nº58387	IACD

No âmbito da cadeira de Programação III, foi-nos pedido a realização de um programa Prolog ou OCaml que contenha o código necessário para criar e jogar o famoso 'Minesweeper'. O Minesweeper é um jogo digital no qual o jogador tem o objetivo de decifrar as posições das 'minas' dentro de um campo quadrado, dividido em várias posições, e marcar os espaços em que elas estão com uma bandeira. A posição de cada mina pode ser assumida pelo jogador, que, ao 'pisar' uma casa, poderá (ou não, no caso de este não ser o caso) receber um número nessa casa, de 1 a 8. Esse número define quantas minas existem nas casas à volta. Por exemplo, uma casa com o número 3 terá 3 minas à sua volta, em quaisquer posições.

O código para o Minesweeper foi escrito em OCaml que é, entre as duas linguagens sugeridas, aquela em que vimos menos problemas quanto a pensar na estrutura geral.

Falando, agora, na estrutura interna e externa de facto. Como cada posição dentro do tabuleiro de jogo varia no estado (pode ser uma posição marcada, ou uma mina 'escondida', por exemplo), definimos o tipo posição, que permite dizer as posições possíveis no jogo: Um espaço que esteja ainda por ser pisado, e não tenha sido marcado, pode dizer-se como 'Escondido'. Por outro lado, um espaço que esteja ainda por ser pisado e não tenha sido marcado, no entanto tenha uma mina lá dentro, pode chamar-se de 'Mina_Escondida'. 'Marcado' é o nome dado para espaços que tenham uma bandeira (tenham sido marcados), e 'Mina_Marcada' são os espaços que têm uma mina e foram marcados com sucesso. Uma mina que tenha sido 'encontrada' (i.e. pisada) pode-se chamar de 'Mina_Encontrada', e um espaço que tenha sido revelado e tenha minas à volta é um Número (o Número sendo quantas minas existem à volta). 'Vazio' é o nome dado a espaços que tenham sido revelados, mas não tenham um Número associado por não terem minas à volta. Para além do tipo 'pos', que representa as posições de jogo, temos também o tipo tabuleiro, que representa o tabuleiro em si. Representamos o tabuleiro como um array de array (matriz) de posições.

Quanto a funções, comecemos pelas funções principais (i.e. as funções pedidas pelo professor). A função empty cria um tabuleiro vazio de NxN posições através da função make_matrix do módulo Array, de dimensões x = n e y = n e tipo inicial de todas as posições como Escondido. Afinal, nenhum dos espaços foi ainda revelado, e não temos Minas ainda. A função principal para colocar minas no tabuleiro (todavia não a única) é a função Random, que antes de tudo verifica se k (o número de minas a ser colocadas, pedido no input) é menor que n*n (impedindo, assim, que alguém crie um tabuleiro que tenha apenas minas, ou tente colocar 26 minas em um tabuleiro de 25 espaços, por exemplo). Se a condição for respeitada, usamos um ciclo que itera k vezes para preencher posições aleatórias no tabuleiro. A escolha da posição é feita pela função int do módulo Random, que escolherá um n aleatório para a e outro para b e, se a posição não for já uma mina escondida (ou seja, se a posição estiver 'livre'), muda o tipo da posição para uma mina escondida. A segunda função que permite colocar uma mina no tabuleiro é a função mine, que coloca uma mina na posição r c (dada no input), sendo r o número da linha e c o número da coluna. Antes de tentar colocar a mina, a função verifica se a posição é válida, retornando a sua invalidade se o r e/ou o c forem menores a 0 ou, por outro lado, maiores que a dimensão máxima de cada linha/coluna. A verificação seguinte é se o espaço dado, ou seja, a posição (r,c) do tabuleiro já têm uma mina escondida. Se já tiver, será

retornada invalidade por já haver uma mina nesse espaço. Finalmente, se as duas condições anteriores se verificarem e a posição dada for escondida, a função mudará o tipo da posição para uma Mina_Escondida, dando a resposta 'ok'.

A seguinte função 'principal' é a função dump, que mostra o tabuleiro, dando print no 'cabeçalho' de jogo (os números das colunas) através de um ciclo que itera n vezes (sendo n a dimensão da linha). Terminada a criação da tabulação das colunas', o código passa para uma linha abaixo e começa a iterar sobre cada linha, dando print da tabulação da linha, e para essa linha cada coluna: Se a posição for escondida de qualquer modo, será um "_"; Se for fazia, é um espaço em branco; Se for uma mina que foi encontrada, é um "+" (a ser usado caso o jogador perca o jogo); Se tiver um numero associado, dá print desse mesmo número e, se for uma posição marcada de qualquer modo, será "#". Cada linha é terminada com ")", tal e qual começa com "(".

Depois, temos a função Step, que, tal como o nome indica, permite ao jogador 'pisar' o espaço indicado no input. As coordenadas da linha que o jogador pretende pisar, r e c, são antes de tudo verificadas se são válidas, de modo igual a como é feito na função mine. Se forem, então um de quatro casos poderá acontecer (separados com o 'match'): Se o espaço escolhido for uma mina escondida, é retornado a mensagem 'boom' e declara-se o fim de jogo com a função auxiliar game_over (que, simplesmente, dá ao jogador a mensagem 'tabuleiro final:', e retorna o tabuleiro final na sua totalidade com um dump); Se o espaço foi marcado, é pedido que escolha outra posição, visto que um espaço já marcado não deve ser pisado; Se o espaço for escondido (mas não uma mina escondida), será usada a função auxiliar 'Sonar' nessa mesma posição para verificar as casas à volta (a função Sonar será explicada em breve); Finalmente, em todos os outros casos (i.e. for vazio ou tiver um número associado), será retornada a mensagem que a posição já foi revelada.

Antes de seguirmos para as demais funções, devemos falar sobre a função sonar, uma função auxiliar que implementamos. Como sabemos quando jogamos Minesweeper, quando encontramos várias posições sem mina adjacentes o próprio jogo limpa-as por nós, para não termos que limpar cada espaço vazio um por um. O objetivo da função é esse mesmo, limpar todos os espaços vazios, e para a recursidade quando o espaço tiver número. Ela cria uma variável n igual ao tamanho do tabuleiro para verificar, como de costume, se a posição em que foi feito o step anterior é válida. Se for, então fará o match da posição do tabuleiro em que foi feito o step com dois possíveis casos, o caso da posição ser escondida e a posição ser de qualquer outro tipo. Se for escondida, usará a função auxiliar conta_minas para contar a quantidade de minas adjacentes (usando a variável contador), alterando a posição para vazio se não houver minas adjacentes (contador = 0), ou para um número (o numero de minas adjacentes) se houverem, de facto, minas adjacentes (contador diferente de zero). Se a posição for qualquer outro tipo de posição, seja um número, uma posição marcada, enfim, a função sonar não fará nada. A função 'mover' é implementada em simultâneo à sonar a partir do 'and', visto que dependem uma da outra. O objetivo de 'mover' é o de, como o nome indica, mover a posição ao redor de uma casa vazia, fazendo-o através de duas iterações de a e b, entre -1 e 1, que representam movimento: Por exemplo, na iteração a = 0 e b = -1, a função 'mover' levará o sonar a ser testado na posição (r, c-1). Adicionamos também uma verificação que indica à função sonar para ser novamente testada nessa posição, contando que não seja a exata mesma posição (a = 0 e b = 0), para permitir criar o efeito de 'procura' desejado.

As próximas funções a se falar são a Mark e a Unmark, sendo uma delas uma adição que fizemos ao jogo para o melhorar. A função Mark simplesmente muda o tipo da posição especificada para uma posição marcada. Se for uma posição escondida, passa a ser marcada, e se for uma mina escondida passa a ser uma mina marcada. Posições de qualquer outro tipo dão fail, visto que não é

possível, por exemplo, marcar uma posição vazia. A Unmark faz o inverso, mudando uma posição marcada para escondida ou, se for uma mina marcada, para uma mina escondida. Sem a Unmark, se o jogador acidentalmente marcasse a casa errada não poderia voltar atrás, deixando o jogo assim mais simples, e não sendo necessário reiniciar depois de qualquer erro. A seguir devemos falar da função 'verificar_done', a penúltima função principal escrita no código. Apesar do nome pedido no trabalho ser done, o OCaml não permite que a função seja chamada de Done pelo nome já ser da base do OCaml, daí mudarmos o nome para 'verificar_done'. A função verificar_done verifica se a quantidade de posições do tipo minas_escondidas e do tipo marcados são 0, e se existe um número diferente de 0 de minas_marcadas. Se não houver espaços marcados, então nenhuma bandeira foi mal colocada (restrição colocada para impedir que o jogador possa vencer o jogo ao colocar todos os espaços como marcados), e se existem minas marcadas enquanto não existe uma única mina escondida então podemos concluir que todas as minas foram marcadas, podendo assim retornar a resposta 'ok'. 'Fail' é retornado se essas condições não forem verificadas.

Por fim, devemos falar da função header, a função minesweeper. A função minesweeper é chamada automaticamente quando se usa o ficheiro sweep.ml, criando a seed de jogo para garantir a aleatoriedade das minas na colocação e iniciando o tabuleiro inicial. A partir daí, o minesweeper faz um match automático de qualquer input dado, permitindo correr os comandos de jogo automaticamente e sem pontos e vírgulas. Por exemplo, dizer "empty 5" apenas fará o empty com N = 5, permitindo jogar com o tabuleiro 5x5.

Todavia, mesmo depois de falarmos da função header, existem mais duas a mencionar, duas funções que já tínhamos intenções de criar, mas apenas acabamos por criar terminado o código de jogo geral: As funções guardar e carregar. Digamos que o jogador esteja a meio de um jogo de Minesweeper, mas têm que sair. Apesar de ter que sair, ele não quer perder o progresso. É para isso que servem as funções que, como o nome indica, permitem guardar o tabuleiro de jogo atual em um ficheiro e carregá-lo para continuar a jogar em outro momento. Começando pela função guardar, ela abre o ficheiro no qual o tabuleiro será guardado (um ficheiro passado para o código no input) e escreve o tamanho do tabuleiro no início do arquivo, sendo o tamanho dado pela função length do módulo Array. Então, iterando sobre cada linha e coluna do tabuleiro, a função escreve cada posição do tabuleiro no ficheiro, representando o tipo da posição de modo diferente. Espaços escondidos como E, minas escondidas como M, marcados com X, minas marcadas com MX, encontradas com +, números como o próprio número e vazios como 0. As posições escritas são divididas por espaço, mas para evitarmos a função carregar de tentar ler mais do que foi escrito impedimos que fosse escrito um espaço no fim. Quando a função finalmente acaba de escrever todo o tabuleiro, fecha o ficheiro e retorna a mensagem de sucesso. A função carregar têm uma lógica semelhante, todavia inversa à de guardar. Primeiramente, ela abre o ficheiro fornecido no input quando chamada e, chamando n à variável representada pelo valor do comprimento escrito no início do ficheiro, cria um tabuleiro nxn de modo igual ao usado na função empty. Depois, o código itera sobre cada linha do ficheiro, dividindo-a em posições separadas por espaço, para que em leitura sejam apenas considerados os caracteres usados anteriormente. Antes de começar a ver qual o tipo de cada posição, é feita uma verificação para garantir que o número de posições é igual ao tamanho, para desse modo impedir que haja mais posições do que o tamanho permite e vice-versa. Caso a condição se verifique (o que deverá verificar, a não ser que tenha surgido um erro a guardar), cada espaço lido será comparado agora a cada espaço do tabuleiro com o match, e o tipo de posição desse mesmo espaço do tabuleiro será mudado conforme o caracter (ou, no caso de mina marcada, caracteres) lido no ficheiro. Quando a iteração for terminada, e todo o ficheiro lido, o ficheiro é fechado, e é enviada mensagem de sucesso, permitindo ao jogador continuar o jogo que tinha guardado anteriormente.