TÉCNICO LISBOA

PROJETO FINAL DE PROGRAMAÇÃO

MASTERMIND

Pedro Ventura, 93155
Rita Palma,93172

Programação 2018/2019 MEEC

1 Introdução

O projeto final consiste numa evolução do projeto intermédio. Em termos de funcionalidades pretende-se que o programa passe a ter a funcionalidade de descobrir uma chave secreta gerada pelo utilizador, ou pelo próprio programa.

O programa terá duas variantes, o modo teste e o modo interativo, neste último o utilizador pensa numa chave secreta e irá introduzir o número de pinos pretos e brancos consoante a sua chave pensada. Enquanto no modo teste, o computador recebe um ficheiro com todas as inicializações para puder correr o programa, e a cada execução irá criar uma lista de listas, como pedido no enunciando, existe uma lista de todos os jogos e cada jogo tem uma lista das tentativas. No final este registo será ordenado consoante o modo de execução do programa.

2 Implementação

2.1 Descrição das estruturas de dados

A estrutura de dados tem o propósito de tornar um programa mais rápido, simplificado e eficiente, armazenando melhor os diversos tipos de dados na memória.

No projeto utilizamos 3 estruturas do tipo typedef, de modo a não realizar sempre uma referência à struct, sendo assim definidas como init_dat, tentativas_pb e registo_jogo: ***Init_dat:*** Contém os valores lidos do ficheiro init, através de fscanf, assim como das inicializações feitas no modo interativo; ***Tentativas_pb:*** Nesta estrutura existem 5 variáveis: 1 string – tentativa - que guarda a tentativa da jogada atual, 3 inteiros – tent_atual, pretos e brancos – que representam, respetivamente, o número da jogada atual, o n.º de pinos pretos e o n.º de pinos brancos, e 1 apontador – nexttentativa - do mesmo tipo que esta estrutura que aponta para a estrutura da tentativa seguinte (formando uma lista); ***Registo_jogo:*** Guarda todos os valores que existem na init_dat, sendo que a maior parte das variáveis não são alterados ao longo do programa, exceto 3 variáveis que são alteradas ao longo de cada jogo, o número de jogadas, o tempo de jogo e a própria chave secreta. Existe ainda 1 apontador – nextjogo - para a estrutura registo_jogo que contém o jogo seguinte e 1 apontador – nexttentativa que aponta para a estrutura tentativas_pb que contém a primeira tentativa desse jogo. Estes 2 ponteiros permitem criar uma lista de listas (simplesmente ligadas) para guardar todos os jogos e as suas respetivas tentativas, para além disso existe um ponteiro do tipo registo_jogo, que é a head da lista que aponta para o primeiro "nó" da lista de listas.

Ao longo do projeto utilizamos memória dinâmica para alocar memória para guardar cada uma destas estruturas e as suas variáveis, fazendo free de cada uma delas no fim da sua utilização, de modo a tornar a utilização da memória mais eficiente.

2.2 Descrição das funções principais

O funcionamento do programa *Mastermind* está dividido em três secções: o algoritmo de jogo, o registo de jogo e a ordenação do ficheiro de histórico.

2.2.1 Algoritmo de Jogo

Primeiramente geram-se as tentativas aleatórias, tal como no projeto intermédio, e obtêm-se o número de pretos e brancos através do oráculo. Em seguida, executa-se o algoritmo. Este baseia-se em gerar todas as combinações possíveis, de acordo com os parâmetros definidos, caso uma combinação seja válida gera as suas permutações e

valida-as. Se for válida a permutação então passará a ser uma tentativa, sendo comparada com a chave secreta através do oráculo.

Para gerar todas as tentativas possíveis com repetição, começamos por gerar a tentativa inicial que colocará todas as posições a 'A' e a variável *j* com o valor de dimchave-1 (última posição do vetor char combinação[dimchave]). Para gerar a tentativa seguinte verificamos sempre a posição final: se for diferente à última cor possível, então incrementamos essa posição, caso contrário vamos incrementar a posição anterior ao *j* – variável auxiliar que passa sempre por referência para esta função e contem uma posição do vetor da combinação. Em seguida, colocam-se todas as posições à frente da incrementada (*j*-1) com valor igual à da posição incrementada e decrementa-se o *j*. Caso a posição *j* do vetor da combinação não tenha a cor máxima, então o *j* passa para a última posição (dimchave-1). No fim deste raciocínio a função retorna à função anterior.

Para gerar as combinações sem repetição, o raciocínio é semelhante em termos de incrementação da última posição sempre que esta é diferente da cor máxima. Caso seja, entra num loop infinito que só sai quando gerada a combinação seguinte. Nesse loop, se combinação[*j*-1]=combinação[*j*]-1, então *j* decrementa (e volta a fazer o loop). Caso contrário, incrementa o valor da posição *j*-1 e coloca todas as posições seguintes com o valor da posição anterior+1, coloca *j*=dimchave-1 e sai do loop.

Para gerar as permutações, primeiramente corremos todas as posições da combinação, para verificar a posição na qual se encontra a letra mais elevada, se na posição atual esse char for superior à seguinte então, irá permutar essas duas posições, utilizando uma variável auxiliar, depois utilizamos um qsort de modo a que não haja permutações repetidas.

Para a validação, no caso de cada combinação considerámos que a soma do número de pinos brancos e pretos do resultado da comparação da combinação com todas as tentativas anteriores tem de ser igual, respetivamente, à soma de pinos brancos e pretos da comparação de cada tentativa com a chave secreta. Assim, apenas as combinações que sejam válidas vão sofrer permutações e respetivas validações, otimizando o algoritmo e tornando-o mais eficiente. Para cada permutação considerámos que o número de pinos brancos e pretos do resultado da comparação da combinação com todas as tentativas anteriores tem de ser igual, respetivamente, à aos pinos brancos e pretos da comparação de cada tentativa com a chave secreta.

2.2.2 Registo de Jogo

No registo de jogo, são utilizadas as estruturas apresentadas na declaração das estruturas e a alocação dinâmica necessária para cada estrutura e respetivas variáveis, de modo a rentabilizar ao máximo a memória e fazendo assim uma gestão mais eficiente da mesma, uma vez que apenas se aloca memória estritamente necessária.

No início de cada jogo é criado uma estrutura do tipo registo_jogo que recebe os valores da estrutura init (valores de inicialização), que é colocada na lista de listas. Para isso são percorridos todos os nós da lista sendo sempre incrementados no sentido do nextjogo. Quando for encontrado o nextjogo que aponta para NULL, significa que o espaço para o jogo seguinte a esse ainda não foi preenchido com nenhum jogo e a estrutura do novo jogo é colocada nessa posição, ficando o seu ponteiro nextjogo a apontar para

NULL e o ponteiro nextjogo do nó anterior fica a apontar para este novo nó. No caso do primeiro jogo a head, ponteiro que aponta para o início da lista está a NULL, logo o novo nó é colocado nessa posição. Em seguida, são geradas todas as tentativas, sendo colocadas na lista de listas depois de ser gerada cada tentativa válida. O raciocínio para inserir uma nova tentativa é idêntico ao do jogo. É encontrada a última posição ocupada por um jogo e em seguida percorrem-se todas as tentativas incluídas nesse jogo e quando se encontrar a nexttentativa a apontar para NULL coloca-se nesse nexttentativa o novo nó.

Após todos os jogos é colocado o registo de jogo no ficheiro histórico pela função fprint da biblioteca de C. Começando na cabeça da lista e através de dois ponteiros auxiliares, um de cada struct que corre cada lista, o registo é escrito no final do ficheiro através do modo "a", quando é chamada a função fopen.

2.2.3 Ordenação do Histórico

A ordenação é feita após o registo do jogo ser colocado no ficheiro histórico, onde o ficheiro é aberto e lido, através da função load_lista, que lê linha a linha o ficheiro. Quando a linha tem 9 argumentos, então significa que estamos perante um jogo, logo iremos guardar as variáveis em questão no registo de jogo, utilizando as funções cria_jogo e insere_jogo, explicadas anteriormente, alocando assim as variáveis do registo de jogo na lista. Quando o número de argumentos é diferente de 9, estamos perante as tentativas do jogo guardado anteriormente, como tal iremos chamar a função criar_tentativa e inserir_tentativa, para guardar as variáveis na lista a cada tentativa encontrada na leitura do ficheiro, correspondente ao jogo em questão. Após todas as linhas serem guardadas na lista, iremos aceder à função sort_lista, que irá organizar a lista por categorias e ordenar cada categoria de acordo com o parâmetro introduzido no modo de execução short/fast.

Iremos criar duas variáveis auxiliares que avançaram sempre em conjunto e de seguida ao longo de toda a lista, de modo a conseguir organizar toda a lista. Este raciocínio está dentro de um loop com 1 if e diversos else's if encadeados, sendo que só sai deste loop caso não entre em nenhuma destas condições. Cada vez que entra num destes casos, trocam-se as posições dos 2 nós na lista, as variáveis auxiliares voltam ao início da lista e repete-se o loop. Caso contrário segue para a seguinte condição (1). Primeiramente, iremos verificar se a dimensão da chave do registo de jogo que estamos a verificar é superior à próxima posição [1]. Em seguida, verificamos o caso da dimensão da chave ser igual à próxima e o número de cores ser superior à seguinte [2]. Depois verificamos se as dimensões de chave e as cores são iguais e a repetição de cor diferente [3]. Seguidamente verifica se a posição em questão e a seguinte tem exatamente a mesma repetição de cor, dimensão da chave e número de cores [4]: se assim for, verifica se estão ordenadas segundo a ordem indicada na execução [4.1] – se não estiverem trocam-se as posições da lista; caso estejam ordenadas iremos executar [4.2]: consoante a ordem de organização, se o valor dessa ordem for superior à da posição seguinte e todas as restantes variáveis tenham o mesmo valor, então executa-se a troca. Caso contrário incrementa as posições e recomeça o ciclo.

Para colocar o registo de jogo no ficheiro, são utilizados 2 for's, 1 que começa em (ptr=head), avançando no sentido ptr=ptr->nextjogo, percorrendo todos os jogos até apontar para NULL, fazendo fprintf de cada estrutura de cada jogo ao longo desse ciclo. Dentro desse for fazemos outro que começa em (ptr2=ptr->nexttentativa), avançando no

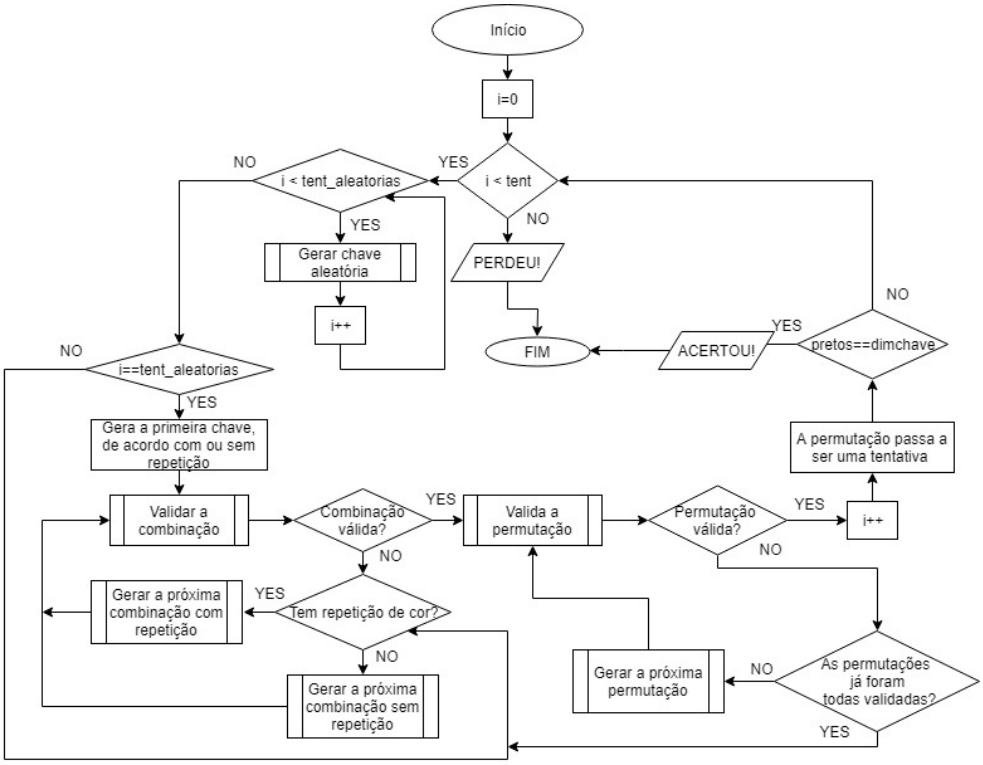
Notas:
(1) Considere [n], as várias condições consecutivas deste programa

sentido ptr=ptr->nexttentativa e que percorre todos as tentativas até apontar para NULL, fazendo fprintf da estrutura de cada tentativa ao longo desse ciclo.

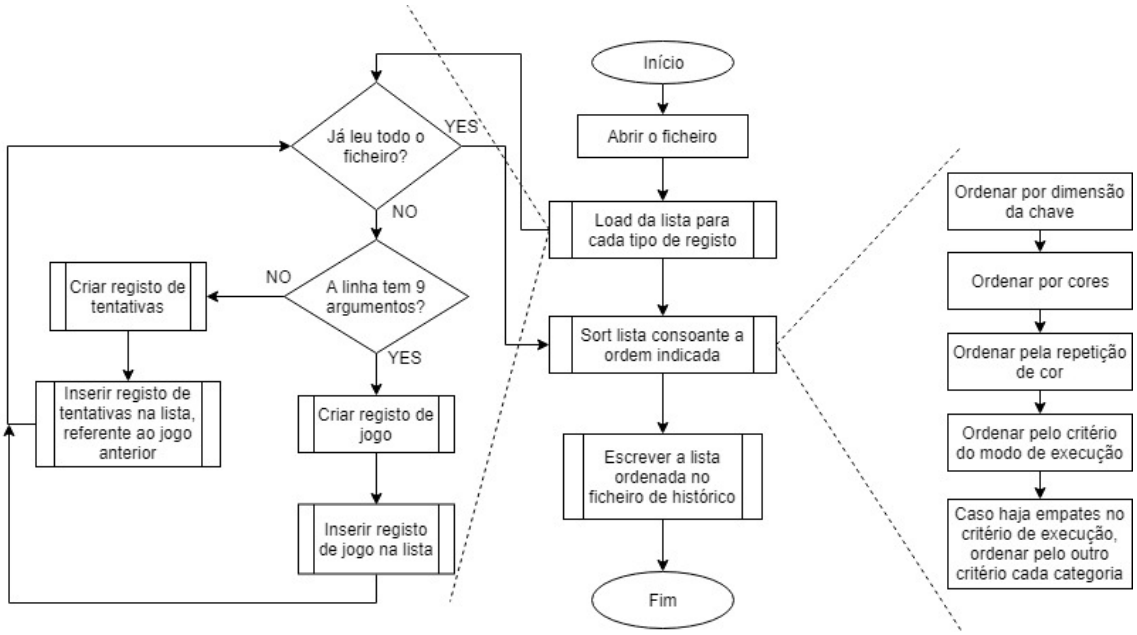
3 Fluxogramas

3.1 Fluxograma das funções principais

3.1.1 Algoritmo de Jogo



3.2.3 Ordenação do Histórico



4 Conclusão

Neste projeto foi proposto a implementação de um algoritmo de jogo do *Mastermind* na linguagem C e um registo de jogo do mesmo, criando um histórico ao longo de todos os jogos.

Após a conclusão do programa, é possível referir que conseguimos implementar um algoritmo 100% correto em termos de jogabilidade e eficiência, acrescentado uma otimização do mesmo. Para além disso, conseguimos obter os resultados esperados, estando incluídas no nosso programa todas as funcionalidades que nos foram propostas.

O programa tem a capacidade de acertar a chave secreta, quer seja no modo interativo ou no modo teste, ordenando o histórico de uma forma rápida e precisa, tendo em conta as categorias e o critério de ordenação

Além disso, conseguimos utilizar de uma forma eficaz alocação dinâmica, obter o número previsto de frees, não possuir erros no valgrind, fazendo uma boa utilização da memória, e criar as listas de listas consoante o que nos foi pedido no enunciado do projeto final.