UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

LUCAS RODRIGUES DE ALMEIDA

RENAN MOREIRA GOMES

Jogo da Bisca feito com a linguagem C

Estrutura de Dados

Espírito Santo - Vitória

2019

Sumário

Bibliotecas

Cards

A biblioteca **Cards** é a biblioteca responsável pela criação e manipulação de uma *estrutura de dados* que abstrai uma *mão de baralho*, ou seja, uma representação abstrata de uma lista de cartas na mão de um jogador.

Convenção Utilizada:

```
// suit 0 = Diamonds
// suit 1 = Spades
// suit 2 = Hearts
// suit 3 = Clubs
// number 8 = Jack
// number 9 = Lady
// number 10 = King
```

Elementos da representação de uma *carta de baralho*:

```
typedef struct dataCard {
        int suit;
        int number;
} DataCard;
```

1. **suit:** Corresponde ao naipe da carta em questão.
2. **number:** Número da carta (sendo que os números 8, 9, 10 correspondem respectivamente à valete, dama e rei).

Elementos de representação de uma mão de baralho:

```
typedef struct cards {
        DataCard data;
        struct cards* next;
} Cards;
typedef struct hand{
        int size;
        Cards *head;
} Hand;
```

1. **head:** Lista de cartas(encadeada).
2. **size:** Tamanho da lista de cartas.

Elementos de representação de uma mão de baralho *mapeada*:

```
typedef struct tablePlay {
        struct hand *h;
        int *id;
} TablePlay;
```

1. **h:** Lista de cartas.
2. **id:** vetor de índices respectivos à lista de cartas.

Funções relativas a manipulação da *mão de baralho*:

1. **Hand* createHand -** Cria uma mão de baralho.
2. **void destroyHand** - Destrói uma mão de baralho.
3. **void push -** Adiciona uma carta no início do baralho.
4. **void pop -** Remove uma carta do final do baralho.
5. **void printList -** Imprime as cartas presentes na mão de baralho.
6. **int isEmpty -** Retorna se a mão de baralho está vazia.
7. **Cards* atPos -** Retorna a carta que está em um índice específico.
8. **int indexOf -** Retorna o índice de uma carta específica.
9. **void erase -** Remove da mão de baralho a carta que está em um índice específico.
10. **Cards* erasePick -** Remove da mão de baralho uma carta em um índice específico e retorna a carta removida.
11. **void insert -** Insere uma carta na mão de baralho em um índice específico.
12. **void insertNode -** Insere uma carta(nó) já existente no final da mão de baralho.
13. **void xchgCards -** Troca a posição da carta A com alguma carta B presentes na mão do baralho.
14. **void fillAllCards -** Preenche o baralho com todas as cartas existentes da bisca.
15. **void shuffleCards -** tem a função de embaralhar o baralho.
16. **DataCard cutDeck -** Corta o baralho, retorna o trunfo e adiciona ele no final do baralho.
17. **TablePlay *sameSuit -** Dado uma carta e um baralho, retorna uma lista de cartas presente no baralho com seus respectivos identificadores(índices no baralho) que possuem o mesmo naipe correspondente a carta dada.
18. **int givePoints -** Retorna o total de pontos de um baralho e o esvazia.

Players

A biblioteca **Players** é a biblioteca responsável pela criação e manipulação de uma *estrutura de dados* que tem como função abstrair características e ações de um *Jogador em um jogo de bisca*.

Elementos da representação de um Jogador:

```
typedef struct player{
        int points;
        Hand *h;
        char *name;
} Player;
```

1. **points:** Corresponde ao número de pontos do jogador.
2. **h:** Lista de cartas na mão do jogador
3. **name:** Nome do jogador.

Elementos da lista de Jogadores em Jogo(Lista Circular):

```
typedef struct playersInGame{
        Player *p;
        struct playersInGame *next;
} PlayersInGame;:
```

1. **p:** Dados do jogador atual.
2. **next:** Próximo jogador.

Funções relativas a manipulação e ação do Jogador:

1. **Player* createPlayer -** Cria um jogador.
2. **void destroyPlayer -** Destrói um jogador.
3. **void showPlayerStats -** Mostra todos os status do jogador.
4. **void insertInTable -** Remove uma carta(pelo índice) da mão do jogador e insere na mesa.
5. **void pickInDeck -** Remove uma carta(do topo) do deck e insere na mão do jogador.
6. **void destroyPlayersInGame -** Destroi a lista circular de jogadores.
7. **void insertPlayerInGame -** Insere um jogador na lista de jogadores em jogo.

Game

A biblioteca **Game** é a biblioteca responsável pela implementação do jogo de baralho chamado bisca, utilizando exaustivamente os recursos da biblioteca **Cards** e **Players**.

Esta biblioteca não possui nenhuma estrutura, ela existe apenas para utilizar as outras bibliotecas e implementar toda lógica por trás da bisca.

Funções da biblioteca game:

1. **void botPlay -** Responsável por controlar a jogada de jogadores não-humanos(bots).
2. **void setupGame -** Verifica se os parâmetros de configuração são válidos se sim, configura o jogo e o inicia, caso contrário o programa é encerrado.
3. **void runGame -** Função principal que implementa de fato a lógica por trás da bisca.
4. **int whoWin -** Função que retorna quem ganhou a rodada de bisca.

Os parâmetros de configuração são especificados na **main**. Sendo necessário o programa ser executado com 3 argumentos.

Instruções para a execução do programa:

Modo:
(1) 2 Jogadores
(2) 4 Jogadores

Dificuldade:
(1) Facil
(2) Intermediário

Executar: ./main <Seu Nome> <Modo> <Dificuldade>
Exemplo: ./main Renan 1 2

Caso os parâmetros não forem respeitados, a execução do programa não ocorrerá de maneira alguma.

Convenção e Miscelânea de Representação das Cartas

Representação de Naipes

Os naipes são enumerados de 0 até 3, cada um representando um naipe em específico.

Naipe 0 = Ouro.
Naipe 1 = Espadas.
Naipe 2 = Corações.
Naipe 3 = Paus.

Contudo, só são representados assim no **back-end** para facilitar a implementação das funções. Quando o jogo é executado ele substitui esses números por caracteres especiais que desenham os naipes.

```
char suitString[4][7] = {"\u2662", "\u2660", "\u2661", "\u2663"};
```

Representação da Família real e Ases

As cartas valetes, dama e rei são representados de forma numérica sendo eles respectivamente as cartas de número 8, 9 e 10.

Os ases são por convenção, representados por cartas de número 1.

Obtendo e compilando o projeto(Makefile)

Obtendo o código fonte e instruções

O código fonte juntamente de arquivos de instrução, configurações do git(.gitignore, .git) e compilação podem ser obtidos no seguinte repositório: https://github.com/realrootboy/bisca.

Especificações do Makefile

O Makefile usa as seguintes flags de compilação: Wall, -g.

Compila as seguintes bibliotecas: Cards.h, Player.h, Game.h.

Tem como arquivo principal o **main.c**, gerando um executável chamado main.

E uma instrução de *limpeza* que remove todos os **.o** e **.gch** obtidos durante o processo de compilação.

Inteligência Artificial(Fácil)

Estratégia Bogo

A inteligência artificial implementada para jogadores não-humanos(bots) no modo fácil é bem simples, ela simplesmente escolhe aleatoriamente qual carta jogar. Essa implementação é feita facilmente utilizando **rand(time(NULL))**. Para garantir que o número obtido do *rand* não seja maior que o tamanho da lista, utilizamos o **resto da divisão** do rand dividido pelo tamanho da lista.

OBS:: Há uma chance absurdamente ínfima dos números escolhidos aleatoriamente gerarem uma sequência de jogadas absurdamente favoráveis para o bot. Mas no geral são apenas jogadas sem sentido e desfavoráveis para o mesmo.

Inteligência Artificial(Intermediário)

Estratégia Analista de Flags

A inteligência artificial de nível intermediário já é bastante mais inteligente que a do nível fácil.

Primeiramente foram criadas várias variáveis do tipo Int, para representar flags, índices e valores.

1. int have1 = 0; - Flag de existência de ases na mesa.
2. int have7 = 0; - Flag de existência de 7 na mesa.
3. int have8 = 0; - Flag de existência da valete na mesa.
4. int have9 = 0; - Flag de existência de dama na mesa.
5. int have10 = 0; - Flag de existência de rei na mesa.
6. int nonHighTrump = 0; - Flag de não-existência de um trunfo na mesa maior que o do bot em questão.
7. int i_have_trump = 0; - Flag de existência de um trunfo na mão do bot.
8. int my_trump_number = -1; - Flag de controle de número do trunfo na mão do bot.
9. int trumpIndex = -1; - Flag de controle do índice do trunfo especificado acima.
10. int highCardValue = 0; - Flag de controle de número de uma carta com número alto na mão do bot.
11. int highCardIndex = -1; - Flag de controle do índice da carta com número alto especificada acima.
12. int significantHigh = 1; - Flag de controle de carta na mesa com valor maior que a maior carta do bot.
13. int insignificantCardMinValue = 7; - Flag de controle de número de carta com número pequeno na mão do bot.
14. int insignificantCardIndex = 0; - Flag de controle do índice da carta com número pequeno especificada acima.
15. int i = 0; - Variável contadora.

Logo após a definição dessas variáveis de controle é feita uma varredura com uma série de verificações na **mão do bot**.

Para cada carta da mão do bot:

1. É feita a verificação se essa carta é um trunfo, se sim a flag de ter trunfo na mão é marcada como 1, ou seja, verdadeiro. É guardado também o número desse trunfo, e o seu índice.
2. É feita a verificação se essa carta é maior que 8 ou maior que a carta de maior valor, se sim é feita a verificação se já foi jogada alguma carta na mesa, se sim é feita uma verificação se a primeira carta dessa mesa tem o mesmo naipe dessa carta(Para tentar eventualmente encartar), se sim é guardado o valor dessa carta na variável responsável, e seu índice também na variável responsável por isso.
3. É feita uma verificação se essa carta tem um número baixo(maior que 1 que representa as e menor que 7 que tem alto valor na contagem de pontos), se

sim é guardado o valor dessa carta na variável responsável, e seu índice também na variável responsável por isso.

Em seguida, após concluída a varredura na **mão do bot** é feita uma varredura nas **cartas da mesa**.

Para cada carta da mesa:

1. É feita uma busca por cartas de valor:
    a. É feita uma verificação se essa carta é um ás, se sim, habilita a flag responsável.
    b. É feita uma verificação se essa carta é um 7, se sim, habilita a flag responsável
    c. É feita uma verificação se essa carta é um valete, dama ou copa, se sim, habilita as flags responsáveis.
2. É feita uma verificação se a carta é um trunfo, se sim, é verificado se o trunfo em questão tem um valor menor que o trunfo na mão do bot

Regras da Bisca

Quem ganha a rodada?

Essa tarefa é dividida em fases. A função responsável por retornar o vencedor é a função `int whoWin(Hand* table, DataCard trumpInfo, int mode)` da biblioteca **Game.h**.

A função retorna um inteiro para permitir encontrar o jogador vencedor sem passar uma estrutura que relaciona quem jogou a carta com a carta em sí. Essa busca é feita por meio da seguinte estrutura de repetição:

```
for( i = 0 ; i < dataWin ; i++)
current = current->next;
```

Assim podemos percorrer a **lista circular** que representa a ordem de vez dos jogadores e passar a vez para o jogador correto(o jogador que ganha a rodada é o primeiro a jogar na próxima rodada).

Temos de seguir as seguintes regras:

1. Caso há um trunfo na mesa, o jogador que jogou o trunfo ganha a partida.

2. Caso há mais de um trunfo na mesa, ganha o jogador que jogou o trunfo de maior número.
3. Caso não haja trunfo e haja naipes diferentes do naipe de quem jogou primeiro, quem jogou primeiro ganha.
4. Caso não haja trunfo e haja uma carta que foi jogada após a primeira tendo o mesmo naipe da primeira porém com valor diferente, ganha quem joga a maior carta.

Assim o ganhador ganha todas as cartas que foram jogadas na rodada e as mesmas são convertidas em pontuação para o jogador. Ganha o jogador que no final da partida tiver o maior número de pontos.

Sistema de Pontuação

A contagem de pontos é feita pela função int givePoints(Hand *h) da biblioteca **Cards.h**.

Para cada carta da lista de cartas passadas para a função é:

1. Somado 11 pontos se for um ás.
2. Somado 10 pontos se for um 7;
3. Somado 2 pontos se for um valete.
4. Somado 3 pontos se for uma dama.
5. Somado 4 pontos se for um rei.
6. As demais cartas não têm valor.

Modos de jogo(2 jogadores e 4 jogadores)

O modo de jogo de 2 jogadores consiste em um jogador 1 e um jogador 2. Cada um busca a vitória individualmente.

Um jogador ganha quando ele ultrapassa 60 pontos, pois o número máximo de pontos é 120.

Já o modo de jogo de 4 jogadores consiste em um jogador 1, um jogador 2, um jogador 3 e um jogador 4 que por mais que busquem a vitória individualmente, estão divididos em dois times. Time 1 : Jogador 1 e jogador 3. Time 2: Jogador 2 e Jogador 4. Um time ganha quando a soma dos pontos de cada jogador do time ultrapassa 60 pontos, pois no modo de 4 jogadores o máximo de pontos também é 120.

Estratégias de Desenvolvimento

Desenvolvimento das bibliotecas

O foco inicial do desenvolvimento do projeto é criar todas as bibliotecas em **ordem de dependência** para poder implementar a parte que interessa com tranquilidade, que é toda a parte lógica da bisca.

Inicialmente é criada a biblioteca **Cards.h** pois é o módulo mais *atômicos*, ou seja, não necessita de outras bibliotecas para funcionar.

Logo em seguida é criada a biblioteca **Player.h**, pois é um módulo quase *atômico*, que tem como requisito para funcionar apenas a biblioteca **Cards.h**.

As duas bibliotecas citadas acima são de extrema importância, tendo como prioridade criar todo um **background** para que as principais rotinas de um jogo de bisca possam ser feitas, que são:

1. Cortar o baralho
2. Jogar cartas na mesa
3. Pegar cartas
4. Embaralhar
5. Cortar o baralho e definir o trunfo

Só assim podemos construir a **Game.h** que implementa a lógica de um jogo de bisca, utilizando **exaustivamente** as funções acima para garantir o funcionamento do jogo.

Uso de memória

É obrigatório o tratamento de vazamento de memória neste projeto. Tudo que é alocado dinamicamente é consequentemente liberado em alguma parte do programa.

Rodando o **valgrind** na arquitetura **amd64** para 2 jogadores obtemos que o uso de memória total do programa é de cerca de **13,500 bytes**. Para 4 jogadores o resultado é semelhante.

Análise de Custo Computacional

Escolha das funções a serem analisadas

Faz sentido analisar o custo das funções mais **importantes**(Primordiais para o funcionamento correto do programa) e **recorrentes**(Em geral, as utilizadas em cada rodada), a fim de obter informações úteis sobre a otimização do projeto.

As funções elegíveis para a análise que cumprem as características citada acima são as seguintes:

1. `void shuffleCards` - Embaralhar.
2. `void pickInDeck` - Jogador pegar uma carta do baralho.
3. `void insertInTable` - Jogador jogar uma carta na mesa.
4. `void printList` - Listar um baralho.
5. `TablePlay* sameSuit(Hand *h, DataCard c)` - Retornar cartas do mesmo naipe.
6. `int whoWin(Hand* table, DataCard trumpInfo, int mode)` - Calcular o ganhador da rodada.

OBS:: Algumas dessas funções usam outras funções dentro delas, cujo o custo também será analisado.

Embaralhar

A função embaralhar contém os seguintes laços de repetição:

```
for( i = 0 ; i < h->size ; i++ )
      for( j = 0 ; j < h->size ; j++ ){
            xchgCards(h, atPos(h, rand() % val_max),
                  atPos(h, rand() % val_max));
      }
```

Portanto, a complexidade da função seria **O(n²)**. Observe atentamente que a função `Cards* atPos` também é chamada duas vezes como argumento da função `void xchgCards`. Essa função chamada de duas vezes, é a função de pegar o elemento na posição **index**, ou seja, o pior caso é **O(index)** ou **O(n)**. Estando

aninhada dentro do for muda a complexidade da função de embaralhar para **O(n³)**.

Observe o trecho de repetição na atPos:

```
for( i = 0 ; i < index ; i++ ) current = current->next;
```

Jogador pegar uma carta do baralho

A função de pegar a carta no baralho é uma função que não possui nenhum laço de repetição, nem chamada recursiva, nem nada do tipo. Porém ela chama funções que possuem essas características citadas.

São essas as `Cards* erasePick` e `void insertNode`. Que fazem respectivamente, pegar uma carta do baralho removendo-a da lista original e inserir uma carta em um baralho x na posição n. Ambas funções possuem complexidade **O(n)**, sendo **n** no pior caso, o tamanho do baralho. Portanto a função de pegar a carta no baralho também é **O(n)**.

Observe o trecho de código presente na função insertNode:

```
while( aux->next != NULL ){
      aux = aux->next;
}
```

A função erasePick usa repetição quando chama a função atPos, que teve sua complexidade explicada acima(página 12, Embaralhar).

Jogador jogar uma carta na mesa

Assim como a função de pegar uma carta no baralho, inserir uma carta na mesa não é diferente, também chama as mesmas funções que possuem as mesmas complexidades citadas acima, portanto também **O(n)**. Pode-se dizer que a função de de jogar uma carta na mesa é **“inversa”** a de pegar uma carta no baralho, pois elas são quase iguais, o que muda é o destino e o remetente da carta em questão.

Listar um baralho

A função de listar(ou imprimir) um baralho na tela é uma função bem simples, que não chama nenhuma outra função, tendo apenas um laço de repetição que percorre o tamanho do baralho, ou seja **O(n)** sendo n o tamanho do baralho.

Observe:

```
while( current != NULL ){
printf("%d - [Suit: %s, Number: %d]\n",
      i,suitString[current->data.suit], current->data.number);
      current = current->next;
      i++;
 }
```

Retornar cartas do mesmo naipe

A função de retornar cartas do mesmo naipe, semelhante a de listar um baralho, obrigatoriamente tem que percorrer todo o baralho, já que ele precisa comparar o naipe repassado carta a carta. Isso leva a crer que também é **O(n)** sendo n o tamanho do baralho passado para a comparação. Conclusão que pode deixar o indivíduo que analisa em dúvida, pois dentro da função de comparação de naipe, tem uma inserção no final de uma nova lista de cartas com o mesmo naipe, tendo complexidade também **O(n)**. Porém isso não torna a função **O(n²)**, pois o "n" é diferente. Seria algo como: for custo O(n) tendo como *"filho"* com for custo O(x), sendo que x é sempre menor ou igual(sendo igual apenas uma vez, ou seja, na última repetição do laço.

Observe que nem no pior caso ocorre **O(n²)**:

```
for( i = 0 ; i < h->size ; i++ ){
      if( aux->data.suit == c.suit ){
            insert(sameCards->h, aux->data, sameCards->h->size);
            sameCards->id[j] = i;
            j++;
      }
      aux = aux->next;
}
```

Calcular o ganhador da rodada

Por fim, temos a função de calcular o ganhador da rodada, que por sua vez possui seus próprios laços e chama eventualmente algumas funções **O(n)**, mas nunca de forma aninhada, mantendo assim a função de calcular jogador com complexidade **O(n)**.

Observe os trechos:

```
while( aux != NULL ){
      if( aux->data.number > max->data.number ){
            max = aux;
            y = sameTrump->id[i];
      }
      aux = aux->next;
      i++;
}
```

```
TablePlay *sameTrump = sameSuit(table, trumpInfo);
```

```
destroyHand(sameTrump->h);
```

```
while( aux != NULL ){
       if( max->data.suit == aux->data.suit )
            if( aux->data.number > max->data.number ){
                v.y = v.x;
                max = aux;
            }
      v.x++;
      aux = aux->next;
   }
```

Note que todos trechos acima usam laços que repetem no máximo n vezes, sendo o pior caso **O(n)**.

Repetição ainda não mostrada na função destroyHand(recursiva):

```
void destroyHand( Hand *h ){
      (instruções de liberar e mudar elemento…)
      destroyHand( h );
}
```