Segunda entrega do projeto de laboratório de programação sobre testes de software e otimização de código

Erick de Brito Sousa Lima

Universidade Federal do Cariri (UFCA), curso de graduação em Ciências da Computação, -

Juazeiro do Norte, CE

erick.brito@aluno.ufca.edu.br

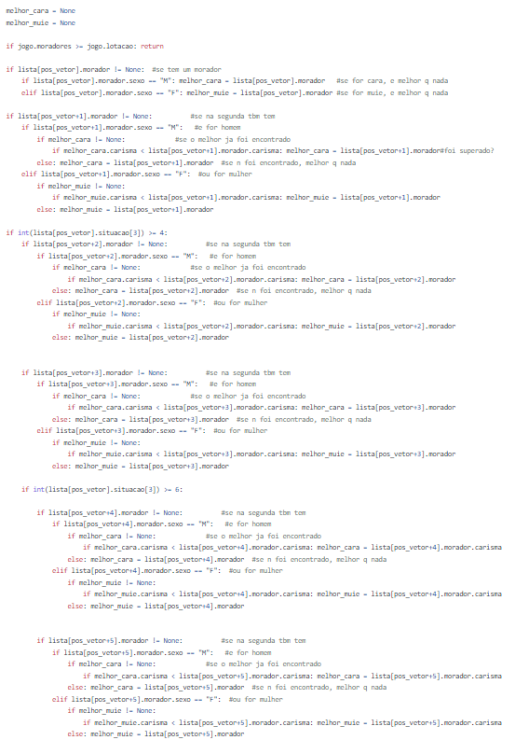
https://github.com/erickxbliv/Jogo-pygame

Resumo. *Este relatório é apenas uma contextualização sobre os métodos utilizados para a segunda entrega deste projeto.*

1. Mudanças estruturais

Este tópico tem como objetivo implementar alguma mudança no código original que possa permitir e facilitar alterações futuras, simplificar grandes funções, evitar duplicação e etc.

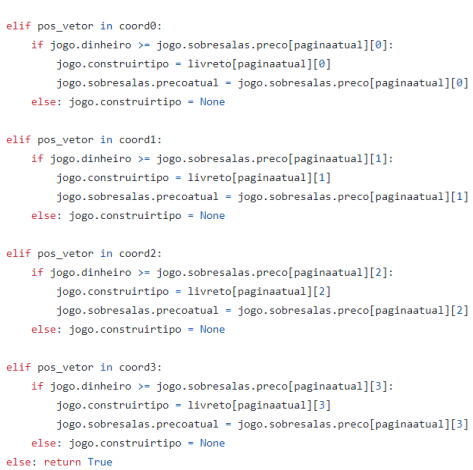
1.1. Primeira alteração, simplificando funções complexas



```
max = int(lista[pos_vetor].situacao[3]) - 1
contador = 1
while contador <= max:
    competicao(pos_vetor,lista,melhor_cara,melhor_muie,contador)
    contador += 1
```

Para evitar a repetição, troquei esse pedaço de código que estava muito complexo e coloquei em uma função, e assim usando um loop para alterar os dados nas vezes que ele aparecia. É uma função que verifica se um filho pode nascer, ao serem encontrados um homem e uma mulher nos quartos, e se houver mais de um ou uma, verificar quais tem os melhores Carismas. Esta cadeia de condições foi colocada dentro de uma função chamada competição. Está presente no arquivo dwellers.py, na linha 186.

1.2. Segunda alteração, evitando repetição de código



```
elif pos_vetor in (coord[0] or coord[1] or coord[2] or coord[3]):
    i = 0
    while i <= 3:
        if pos_vetor in coord[i]: mostrar_pag(jogo,livreto,paginaatual,i)
        i += 1
else: return True
```

Para corrigir este problema, apenas transformei 4 listas em uma matriz, e assim pude acessar seus dados em um loop de forma bem fácil. O código que se repetia está na função mostrar_pag, e do arquivo menu.py vemos a linha 280. Desta forma, agora está até mais fácil também para implementações futuras,

pois caso eu quisesse adicionar mais itens a loja, consequentemente precisando de uma nova página como vitrine, não haveria mais problema se eu tivesse itens demais.

2. Suítes de teste

Nosso próximo objetivo será criar duas suítes de teste, uma para caixa preta e outra para caixa branca, as duas contendo pelo menos 5 testes cada, para assim fazer uma melhor verificação do código e utilização dos conhecimentos adquiridos em sala.

2.1. Caixa preta

```
def carregou_corretamente(dwellers):
    contagem = 0
    while contagem < len(dwellers):
        if dwellers[contagem] != None:
            assert dwellers[contagem].sexo != None
            assert dwellers[contagem].vida > 0
        contagem += 1
def emprego_certo(lista,dwellers):
    contagem = 0
    while contagem < len(dwellers):
        if dwellers[contagem] != None:
            assert dwellers[contagem].celula == lista[dwellers[contagem].celula.id-1]
            contagem += 1
def ampulheta(jogo):    #esta funcao zera o horario e troca o dia
    horario = 1.0   #mas ha uma funcao antes que executa comandos
    sair = False    #a meia noite, sendo preciso garantir que 24:00 acontece
    while not sair:
        horario += jogo.passagem
        if horario > 24.3:
            assert (horario - jogo.passagem) >= 24.0
            sair = True
            horario = 1.0
def inicializacao(jogo):
    assert jogo.dados.dificuldade != None
    assert jogo.dados.carregar != None
    assert jogo.dados != None
    assert jogo.sobresalas != None
    assert jogo.moradores > 0
def posicoes(lista):
    contagem = 0
    while contagem < 147:
        assert lista[contagem].coordenadas != None
        contagem += 1
```

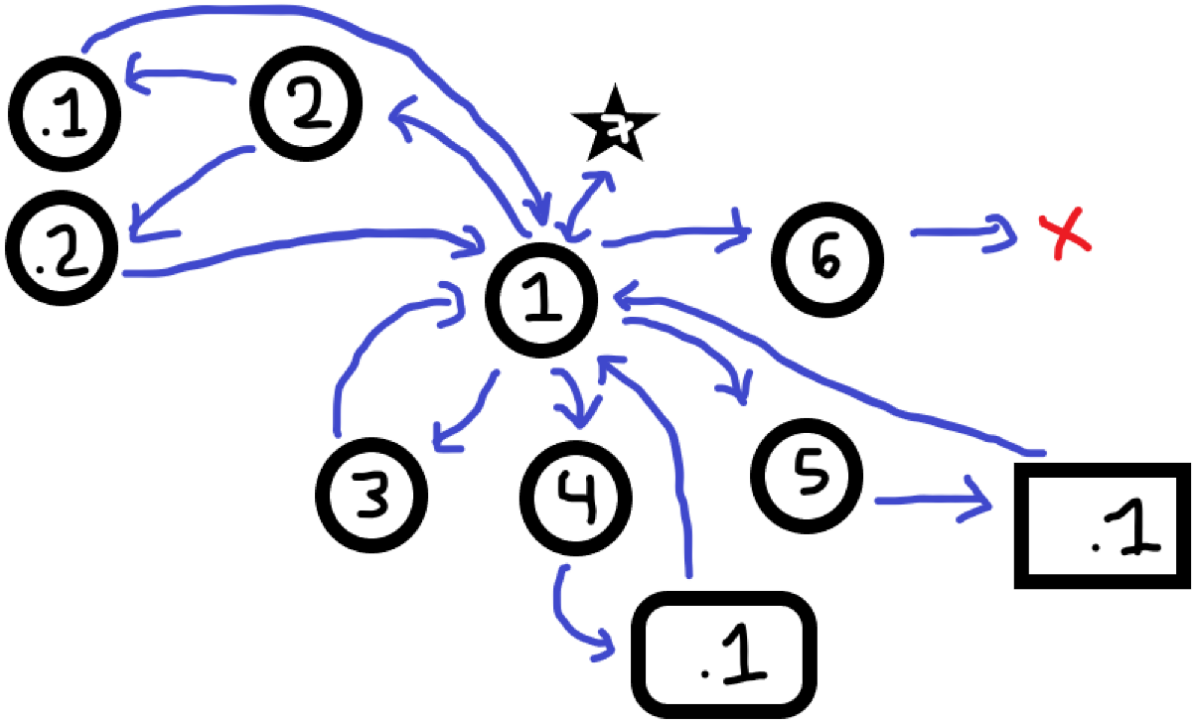
Aqui podemos ver que eu construí um arquivo com 5 testagens para serem feitas, além de uma função extra que abre essas vistas na imagem acima. Essa função “mãe” é chamada logo antes do loop infinito do jogo ser iniciado, para testar se tudo está funcionando corretamente antes de começar. Cada uma tem importância, pois indica que os dados foram preenchidos corretamente para o início do jogo, com exceção de ampulheta, que apenas verifica se a passagem de tempo está com um valor suficientemente bom para que o horário de meia-noite seja constatado durante o jogo, e não imediatamente pulado caso entre nessa condição que zera o valor do horário atual após as 24 horas.

O comando assert é de fato muito conveniente, pois pelo fato do python ser uma linguagem interpretada, às vezes o erro passa despercebido por nem ter sido lido ainda, e quando é, temos que verificar várias vezes o valor das variáveis para ver onde está dando erro, falta e falha, seja através de printar na tela ou depurando.

2.1. Caixa branca

Agora, vamos analisar vários resultados possíveis para diferentes tipos de inputs no código durante funcionalidades diferentes. A caixa branca é mais da parte de implementação, então separei esses dois blocos para testarmos os caminhos possíveis e se funcionam.

1. Após entrar no loop do jogo, o usuário está em controle de onde vai clicar.
2. Para remover uma pedra. .1 Sim, gastando dinheiro se houver, e .2 não
3. Para confiar um novo visitante a ser um dos moradores do alojamento
4. Acessar os subsistemas encontrados ao clicar em uma sala
5. Acessar o sistema principal, que controla várias funções
6. Clicar para fechar o jogo
7. para coletar a produção de um morador

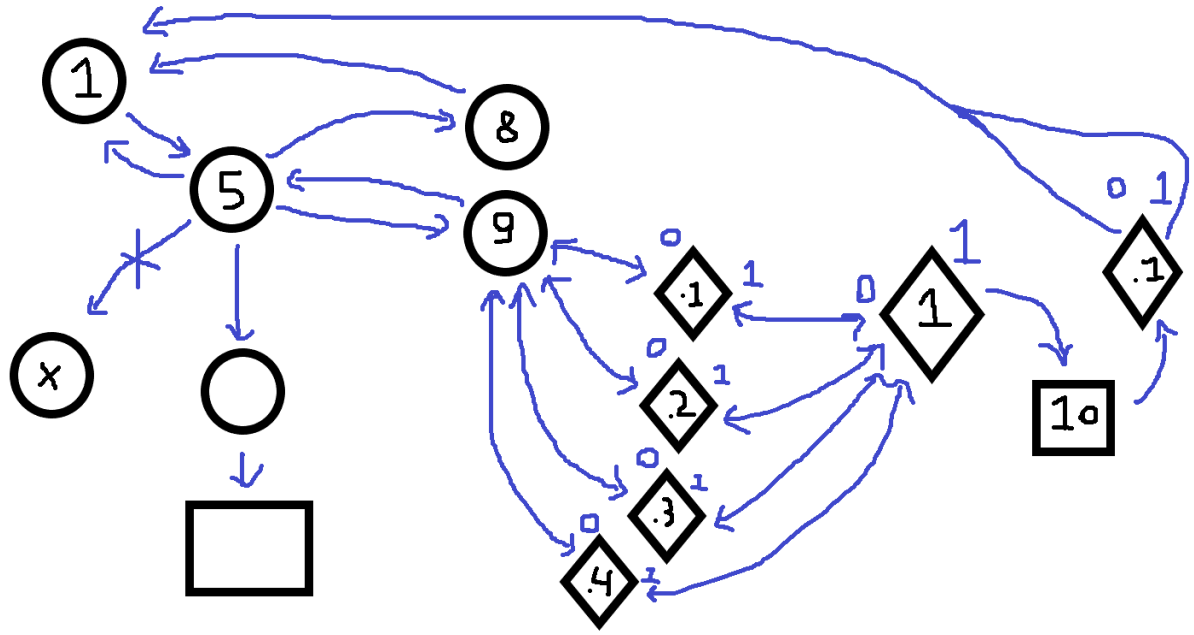


situação	**desenvolvimento**
para sair do jogo	{1,6}
dinheiro insuficiente para pedra	{1,2,2.2,1}
realizar várias atividades	{1,7,1,3,1,3,16}
Abrir subsistemas	{1,5,...,1}

Agora vamos explorar mais os subsistemas do código. Mais especificamente ao entrar no nó Nº 5 é visto o sistema e entre as opções nós podemos escolher para construir novas

instalações. A ramificação é bem sensível e circular também, então vamos analisar a corretude: Já conhecemos 1 e 5, agora temos,

8. É a opção de fechar o sistema
9. Entramos para a seleção de salas, é nos mostrado um livreto com 4 opções de salas para construir e uma opção para voltar ao sistema.
   Ao selecionar uma das salas, .1 .2 .3 ou .4, somos levados de volta ao 1, mas desta vez em forma de condição, além da opção voltar, é necessário verificar se o local que está sendo escolhido para construir a sala selecionada é válido.
10. se o lugar é válido seremos levados para a função que constrói a sala
    10.1. agora será verificado se esta sala pode se fundir com vizinhas, e então retorna




```
if jogo.construirtipo != None:
    voltou = funcoes.preparar_obra(jogo,lista)
    if voltou == False:
        return False
    else: jogo.construirtipo = None
```

```
if executar:
    erguer(lista,pos_vetor,jogo,fundir)
    jogo.modo = "espectador"
    pretendencia(lista,pos_vetor,False)
    pretendencia(lista,pos_vetor+1,False)
    jogo.construirtipo = None
    return False
```

```
if fundir != None: fusao(jogo,lista,pos_vetor,fundir)
jogo.sobresalas.calcconsumo()
sucesso =  pygame.mixer.Sound(path.join('sons','obrafinalizada.wav'))
pygame.mixer.Sound.play(sucesso)
```

Quase todas as funções circulares retornam um bool, se o usuário tiver apenas voltado ele só precisa andar uma aresta para trás, se ele tiver completado a tarefa ele retorna falso, e a função que recebeu esse retorno também retorna falso, entrando em uma cadeia para voltar para o jogo principal (nó Nº 1).

situação	**desenvolvimento**
indecisão sobre a sala	{1,5,9,9.3,1,9,9.2,1,9}
cancelando a operação	{1,5,9,9.1,1,9.1,9,5,1}

dinheiro insuficiente	{1,5,9,9.1,9,5,1}
finalizando a operação	{1,5,9,9.4,1,10,1}
não há localidades válidas para construir	{1,5,9,9.3,1,9.3,1,9.3,1}