

Projeto de Laboratórios de Informática 1

Grupo g166

Mariana Miranda (a77782) Helena Poleri (a78633)

3 de Janeiro de 2016

Resumo

Para esta segunda fase do projeto de Laboratórios de Informática 1 (LI1), foi nos pedido que realizássemos três tarefas relacionadas com o *puzzle* Sokoban, exceto a segunda. Foi também a primeira vez que trabalhamos com a biblioteca gráfica *Gloss*, que utilizámos para as últimas duas tarefas, das quais uma consistia no desenvolvimento da *interface gráfica* do jogo *Sokoban*.

Conseguimos atingir os nossos objetivos ao finalizar as três tarefas. As duas que eram objeto de avaliação do *Mooshak*, foram totalmente aceites por este. Para além disso, e como a última tarefa de criação da interface gráfica era aberta, ainda conseguimos adicionar os extras que queríamos ao jogo.

Conteúdo

1	Introdução	3
2	Descrição do Problema	3
3	Concepção da Solução	4
3.1	Estruturas de Dados	4
3.2	Implementação	5
3.3	Extras	6
3.4	Testes	8
4	Conclusões	9

Lista de Figuras

1	Jogo a correr	6
2	Jogo antes e depois de um <i>undo</i>	7
3	Jogo antes e depois de um <i>shuffle</i>	8
4	Indicação de que os testes foram todos passados no terminal	9

Lista de Tabelas

1	Imagens usadas no jogo	4
---	----------------------------------	---

1 Introdução

Desenvolvida no âmbito da disciplina de Laboratórios de Informática 1 (LI1), da Licenciatura em Engenharia Informática da Universidade do Minho, esta segunda fase do projecto, desenvolvida na linguagem de programação funcional *Haskell*, continua a basear-se no puzzle *Sokoban*. Neste, controla-se um boneco numa arrecadação por intermédio de comandos e o objectivo é empurrar caixas para posições de arrumação.

Previamente, na primeira fase, tivemos que desenvolver pequenas aplicações em *Haskell*, relacionadas com este puzzle. A primeira pretendia-se que fosse capaz de validar um mapa fornecido, e verificar se este respeitava determinadas condições que nos foram dadas. Se estivesse correcto, o programa deveria devolver um OK e, em caso contrário, a linha onde encontrou a primeira falha. Depois de o mapa estar válido, a segunda deveria pô-lo pronto a ser usado para o puzzle e o seu output era, justamente, o mapa no seu formato final. A última tarefa da primeira fase, pedia-nos que fosse criado um programa que determinasse a posição do boneco após a execução de um único comando, sendo os comandos disponíveis L (*Left*), U (*Up*), R (*Right*) e D (*Down*).

A segunda fase consiste na realização de outras três tarefas, das quais se deve salientar a última, por consistir na criação da interface gráfica do puzzle *Sokoban*, à semelhança daquele existente no sítio sokoban.info, e cuja conceção pode ser encontrada na secção 3.

2 Descrição do Problema

A primeira tarefa (tarefa D) é muito semelhante à última tarefa da primeira fase, podendo ser considerada uma continuação desta. Isto é, enquanto que na tarefa C se pretendia implementar um programa que determinava qual a posição do boneco após a execução de um comando, nesta tarefa pretende-se executar toda uma sequência de comandos, até que o puzzle termine ou até que não existam mais comandos.

Sendo assim, o formato de entrada deverá ser um mapa seguido das coordenadas iniciais do boneco/caixas, e uma linha adicional que contém uma sequência de comandos (U, L, D ou R) não vazia, comandos estes que devem ser executados sequencialmente. Assim, e sendo que o objectivo do puzzle *Sokoban* é arrumar as caixas nos respectivos locais de arrumação, o programa deverá imprimir uma linha com o texto “FIM <tick_count>” onde <tick_count> representa o número de comandos que, efetivamente, provocaram alguma alteração no estado do jogo. Note-se também que, logo que for atingida a configuração final, o programa deve parar de executar comandos, mesmo que existam alguns ainda não processados.

Se, após a execução de todos os comandos, as caixas não estiverem arrumadas nos locais pretendidos, o programa deve imprimir o texto “INCOMPLETO <tick_count>” onde <tick_count> representa, uma vez mais, o número de comandos que, efetivamente, provocaram alguma alteração no estado do jogo.

Tanto a tarefa E como a F pretendem fazer uso da biblioteca gráfica *Gloss*. A primeira, tarefa E, nada tem a ver com o puzzle *Sokoban*, mas permite-nos um maior entendimento do funcionamento da biblioteca usada, lidando com um tipo de dados fundamental dessa biblioteca: a *Picture*. O objectivo é que o programa determine as dimensões do menor retângulo envolvente de uma *Picture*, devolvendo uma única linha impressa com as dimensões deste, i.e. “<larg> <alt>”, onde <larg> e <alt> correspondem, respetivamente, ao arredondamento para inteiros (obtidos usando a função `round`) da largura e da altura do retângulo envolvente, separados por um espaço. Esta tarefa também deverá excluir os construtores `ThickCircle`, `Arc`, `ThickArc` e `Text`, já que, nestes casos, não é evidente qual o retângulo envolvente da figura.

Na última tarefa pretende-se criar a interface gráfica do jogo *Sokoban*, e tornar possível a interação com este, associando comandos a teclas específicas. O objectivo final é um jogo semelhante ao disponível no sítio sokoban.info. Sendo esta uma tarefa aberta, pretendemos adicionar alguns extras ao jogo, como, por exemplo, possibilitar ao jogador desfazer as últimas jogadas (`undo`), recomeçar o jogo (`restart`), ver quantas jogadas já fez no ecrã (número de `moves`) e guardar a sua melhor score (`best score`), podendo associar o seu nome a esta. Para além disso, o puzzle

vai dispor de múltiplos níveis com várias dificuldades tornando-se, assim, cada vez mais desafiante para o jogador. Para enriquecer a *interface gráfica* do jogo, criámos ainda imagens para os vários elementos do jogo (Tabela 1).







Descrição	Caracter	Imagem
Boneco	'o'	
Caixa	'H'	
Caixa Arrumada	'I'	
Chão	','	
Parede	'#'	
Local de Arrumação	','	

Tabela 1: Imagens usadas no jogo

3 Concepção da Solução

3.1 Estruturas de Dados

Na tarefa F, usámos a seguinte estrutura de dados:

```
data Mapa = Mapa {score :: (Int,Int)
  , memoria :: [(Float,Float)]
  , nome :: String
  , imagens :: [Picture]
  , coord :: [(Float,Float)]
  , mapi :: [String]
  , nivel :: Int
  , tempo :: (Int,Int)
  , sh :: Int}
```

Ao construtor *Mapa* foi associado, a cada um dos seus parâmetros, um nome (uma etiqueta). Deste modo, *Mapa* é constituído por nove parâmetros sendo cada um essencial para os diferentes aspectos da tarefa. Assim sendo,

- **score** dá-nos um tuplo, cujo primeiro elemento corresponde ao número de *moves* efetuados e o segundo elemento corresponde à melhor pontuação conseguida naquele nível, até ao momento;
- **memoria** guarda as posições que o boneco e as caixas ocuparam ao longo do jogo que vai, mais tarde, permitir fazer *undo* e *restart* no puzzle;
- **nome** que permite associar uma determinada pontuação o nome do jogador que a conseguiu;

- `imagens` é a lista de algumas imagens que são usadas ao longo do jogo;
- `coord` é a lista das coordenadas correspondente à posição do boneco e das caixas;
- `mapi` é o tabuleiro de jogo;
- `nivel` dá-nos informação sobre qual é o nível atual;
- `tempo` dá-nos um tuplo cujo primeiro elemento dá-nos o tempo que já passou e o segundo o tempo associado à atual melhor pontuação;
- `sh` diz-nos quantos shuffles ainda estão disponíveis (sendo que, foi estipulado que o número máximo de shuffles que o jogador pode fazer é 3).

É de notar que esta estrutura de dados e o uso de *Records* nos facilitou a escrita do código. Ao escrever o código assim, o *Haskell* automaticamente criou as funções `score`, `memoria`, `nome`, `imagens`, etc que ficaram prontas a usar desta maneira.

3.2 Implementação

Na tarefa F, começámos por usar como modelo o exemplo que nos foi fornecido na Blackboard. O primeiro objetivo e o mais simples, era por o jogo simplesmente a correr, sem nenhuns extras e, para isso, usámos funções que foram desenvolvidas em tarefas anteriores.

A **tarefa A** da 1ª Fase deu-nos bases para processamento do *input* que nos era dado e deu-nos conhecimento sobre as regras básicas do puzzle. Foram usadas na F, algumas funções desta tarefa, como por exemplo:

```
processaMapa :: String -> ([String],[(Float,Float)])
processaMapa linhas = let (x,y) = parteMapa $ inStr linhas
                      in (x, leCoordenadas y)
```

Esta função, `processaMapa`, é útil para a tarefa F porque, sendo que o input inicial é uma *string* com o mapa e as coordenadas todas juntas, esta divide-a na parte do mapa e na parte da lista das coordenadas. Para além disso, transforma a informação sobre as coordenadas em pares de coordenadas (x, y) .

Na **tarefa B**, usámos uma função que nos permitia remover os caracteres '#' redundantes de um tabuleiro (i.e. quando todos à sua volta são também '#'). Assim, esta função foi também usada na nossa tarefa F, sendo que os mapas que usamos vinham com os caracteres redundantes, que precisavam de ser removidos para não aparecerem na interface gráfica. A função que se encarrega de o fazer, chama-se `rmov`.

```
rmov :: [String]-> [String]
```

Da **tarefa D**, foi usada uma das tarefas mais básicas para o funcionamento do jogo, a `move`. Esta função recebe um comando, as coordenadas do boneco e das caixas e o tabuleiro e devolve as coordenadas do boneco e das caixas depois das alterações resultantes desse comando. É esta função que vai, na nossa tarefa F, servir como resposta ao clique nas teclas das setas. Por exemplo, se a tecla pressionada for a da Esquerda, o move vai ser "ativado" com o caracter L e mover, se possível, o boneco para a esquerda.

```

move :: Char -> [(Float,Float)] -> [String] -> [(Float,Float)]
move x l l1 | equiv (coordenadas '.' (reverse l1)) l2 = ((a,b):l2)
            | x == 'L' && moveM (a-1,b) (a-2,b) l2 l1 = ((a-1,b):(moveCaixa (a,b) (a-1,b) l2))
            | x == 'R' && moveM (a+1,b) (a+2,b) l2 l1 = ((a+1,b):(moveCaixa (a,b) (a+1,b) l2))
            | x == 'U' && moveM (a,b+1) (a,b+2) l2 l1 = ((a,b+1):(moveCaixa (a,b) (a,b+1) l2))
            | x == 'D' && moveM (a,b-1) (a,b-2) l2 l1 = ((a,b-1):(moveCaixa (a,b) (a,b-1) l2))
            | otherwise = ((a,b):l2)
              where (a,b) = head l
                    l2 = tail l

```

Na tarefa F, pode-se dizer ainda que existem funções base essenciais que nos permitiram criar a interface gráfica e interagir com esta. Destas destacam-se:

- **joga** que interage com os diferentes componentes do jogo e as funções principais (desenhaMapa, reageEvento, reageTempo) e cria-o;
- **desenhaMapa** que a partir de um *Mapa* cria a interface gráfica (IO Picture);
- **reageEvento** que a partir da interação com um teclado ou um rato, reconhece o pressionar de um tecla e reage de acordo com esta, alterando o seu estado;
- **reageTempo** que altera o estado do *Mapa* com o passar do tempo.

Entretanto, descobrimos ainda um problema com a eficiência do nosso programa. As imagens, usadas ao longo do jogo, estavam a ser carregadas todas no início, assim como todos os níveis, o que era totalmente desnecessário. Para criar o jogo, estávamos a fazer uso do módulo *Graphics.Gloss.Interface.Pure.Game*, mas como tínhamos que resolver o problema da eficiência, começámos a usar o módulo *Graphics.Gloss.Interface.IO.Game*. Com algumas alterações no código, este permitiu-nos escrevê-lo duma maneira em que as imagens e os diferentes níveis fossem só carregados quando necessários.



Figura 1: Jogo a correr

3.3 Extras

Com o mínimo feito, decidimos adicionar alguns extras ao puzzle para melhor o programa final.

- **Undo** (Tab) e **Restart** (Enter)

Estas são, talvez, as duas funcionalidades adicionadas mais importantes. À semelhança do puzzle existente no sítio Sokoban.info, adicionámos a possibilidade de o jogador desfazer um movimento (*Undo*) e de recommençar o nível (*Restart*). Antes disto, se o jogador se enganasse a clicar nas teclas, teria que fechar a aplicação e abri-la de novo. Dado que ao desfazer um movimento, o número de moves x passa a $x - 1$ (ver Figura 2), o jogador pode usar esta funcionalidade para considerar um novo caminho, no qual efetue menos movimentos, e assim

conseguir uma melhor pontuação. Para tal, ao longo do jogo as posições do boneco e das caixas são guardadas e quando se retrocede uma jogada ou se recomeça apenas se vai buscar ,respetivamente, a última ou a primeira lista das posições que estes ocuparam.

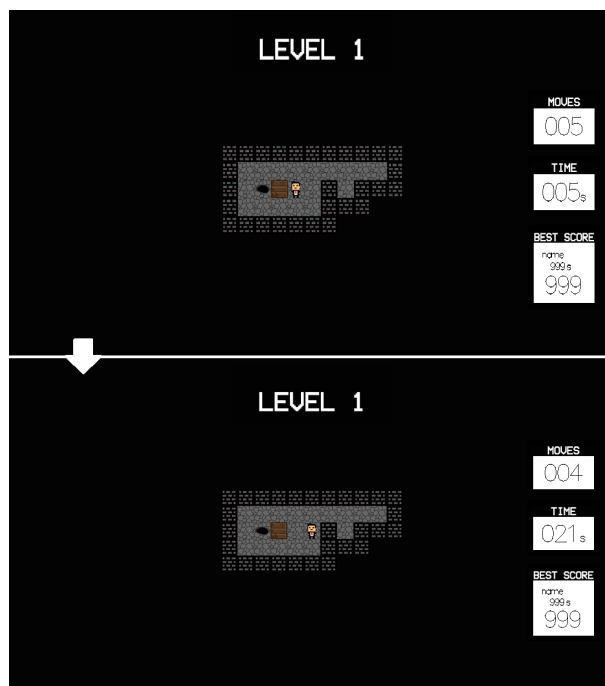


Figura 2: Jogo antes e depois de um *undo*

- Níveis

Sendo o *Sokoban* um puzzle, pode ter vários mapas, com diferentes dificuldades. Assim, adicionámos diferentes níveis (30 ao todo) ao nosso, com diferentes dificuldades e diferentes números de caixas, sendo possível avançar e retroceder de nível.

Tanto avançar de nível (F2) como retroceder (F3) tem métodos muito similares. Como *Mapa* dá-nos a informação do nível atual, ao pressionar F2 a função **nextlevel** carrega o nível seguinte, exceto no caso de se estar atualmente no nível 30 (último), ou o jogador ainda não ter carregado Enter para iniciar o jogo. Do mesmo modo a função **previouslevel** carrega o nível anterior, exceto se o jogador estiver no primeiro nível. Ambas as funções invocam a função **constroimapa** que, com a informação do nível, constrói um novo *Mapa*.

- *Shuffle* (F1)

A funcionalidade de *Shuffle* (Figura 3) permite ao jogador, em cada nível, mudar a posição das caixas. Sendo que foi estipulado que o número máximo de *shuffles* que um jogador pode fazer por nível é 3, isto implica que, para além da configuração inicial das caixas, o jogador pode tentar jogar o mesmo nível com outras três configurações. Para tal, ao carregar F1 o programa lê um ficheiro com as posições possíveis que as caixas podem ocupar - excluindo assim os lugares em que a finalização do nível seria impossível - e, através da função **randomRIO** escolhe aleatoriamente novos locais para as caixas, tendo em consideração aquelas que já estão num local de arrumação, ou seja, estas não são reposicionadas.

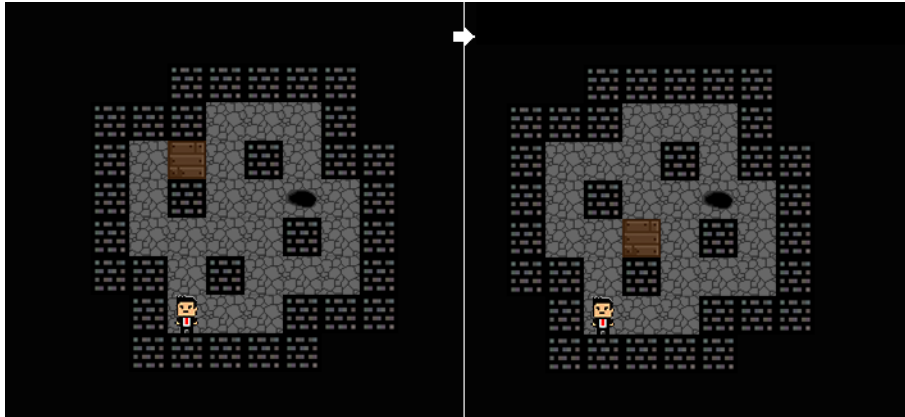


Figura 3: Jogo antes e depois de um *shuffle*

- *Best Score*, nome e tempo

No *Sokoban*, a *score* corresponde, por padrão, ao número de movimentos efetuados por um jogador. Quanto menor for o número de movimentos, melhor é o score (i.e uma pessoa que completou um puzzle com 6 movimentos tem um melhor score que uma que usou 7 para o finalizar). Deste modo, assim que o jogador atingir uma pontuação melhor que a atual (antes de qualquer jogo a pontuação está em 999), o jogador pode associar o seu nome a ela. Se esse nível for jogado outra vez, essa pontuação vai ser mostrada, juntamente com o nome do jogador que a obteve, num retângulo com o nome de *Best score* na interface do jogo, até que outro jogador, ou o mesmo, faça melhor.

Como é possível que o jogador faça o mesmo *score* duas vezes, foi adicionado tempo à janela de jogo também. Quando o nível começa, o tempo começa a contar. O tempo servirá, portanto, como um "desempate" no caso de o jogador obter a mesma *score* duas vezes. Assim, se o número de movimentos for igual, a pontuação que conta como melhor, é aquela à qual está associada um menor tempo.

Para facilitar foram criadas 3 funções : **concluido** que verifica se já terminou o nível, **worse** e **best**, que verificam respetivamente se o desempenho foi pior ou melhor, do que aquele que obteve melhor pontuação.

```
concluido m = equiv (coordenadas '.' (mapi m)) (tail (coord m))
worse m = (concluido m && (fst (score m)) == (snd (score m)) && fst (tempo m) > snd (tempo m))
          || (concluido m && fst (score m) > snd (score m))
best m = (concluido m && fst (score m) == snd (score m) && fst (tempo m) <= snd (tempo m))
          || (concluido m && fst (score m) < snd (score m))
```

3.4 Testes

Para que tivéssemos a certeza de que as aplicações que nos foram pedidas estavam a funcionar bem e tinham o desempenho necessário, foram desenvolvidos testes para estas. Tanto para a função D como para a função E, usámos dois tipos de *testing*: *black-box* (tipo mooshak) e *white-box* (HUnit).

Para os teste do tipo *black-box*, tivemos que definir ficheiros de *input* e *output*. No ficheiro de *output* púnhamos o *output* esperado e ao testar, o programa iria comparar o *output* do programa com o esperado. O *white-box testing*, por sua vez, testa com conhecimento total do código. Como é possível testar funções individuais no código (testes unitários) e não apenas o *input* e *output* finais, proporciona uma maior cobertura do código.

Por exemplo, para a tarefa D, nos testes do tipo *black-box*, tínhamos que criar um ficheiro .in com o mapa do jogo, seguido das coordenadas e da *string* com os movimentos e um ficheiro .out

Quanto aos testes unitários, por exemplo na tarefa E, tínhamos uma lista de testes para a função principal, para podermos testar com todos os tipos de *Pictures* do *Gloss*. Um exemplo de um teste unitário para a Tarefa E é o seguinte:

Depois dos testes todos definidos, era só correr a função `main` dos testes no terminal que nos era dito a quantos testes o nosso código tinha passado (Figura 4).

Figura 4: Indicação de que os testes foram todos passados no terminal

Neste projeto, pretendíamos resolver as três tarefas que nos foram propostas, nomeadamente uma na qual tínhamos que criar a interface gráfica do jogo *Sokoban*, e tornar possível a interação com este, associando comandos a teclas específicas (semelhante ao disponível em sokoban.info).

Por sua vez, a última tarefa era aberta. Pretendíamos, para além do básico, adicionar alguns extras ao jogo, e conseguimos com que fosse possível ao jogador desfazer as últimas jogadas (*undo*), recomeçar o jogo (*restart*), ver quantas jogadas já fez no ecrã (número de *moves*), e guardar a sua melhor score e associar o seu nome a esta. Para além disso, conseguimos com que o nosso jogo dispusesse de vários níveis com várias dificuldades, para poder dar uma melhor experiência de jogo a quem o jogar. Como atingimos todos os nossos objetivos, achámos que o resultado final foi muito positivo.