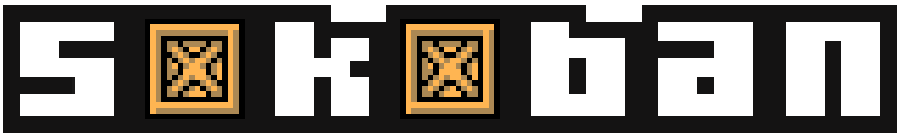
Universidade do Minho

Mestrado Integrado em Engenharia Informártica



Laboratórios de Informática I

Grupo li1g126

Miguel Magalhães (a77789)
Hugo Oliveira (a78565)

3 de Janeiro de 2016

Conteúdo

Resumo

Este documento, desenvolvido na unidade curricular de Laboratórios de Informática 1 (LI1), apresenta uma resolução de um jogo do tipo Sokoban.

Um dos objetivos da resolução deste problema, é tornar o jogo diferente do típico jogo Sokoban, como tal, foi acrescentado ao jogo a possibilidade de alterar o tema, de alterar os níveis de jogo, de utilizar Undo e Restart ainda de poder visualizar o Score.

O relatório baseia-se, fundamentalmente, nas partes mais importantes dos códigos desenvolvidos na resolução do jogo. Assim, este está dividido em 4 partes, sendo que as duas últimas são as mais importantes: Fase Inicial do jogo, Solver, Movimentação do Sokoban e ainda Interface Gráfica.

Capítulo 1

Introdução

Ao longo deste trabalho iremos explicar alguns dos passos fundamentais à realização do jogo *Sokoban*. Com isso pretendemos fazer um percurso objetivo e rigoroso sobre a resolução deste problema.

Todo o jogo foi desenvolvido na linguagem de programação funcional Haskell. É de notar, que neste relatório, nos incidimos sobretudo na realização da última tarefa, que consiste na realização da interface gráfica do jogo, onde recorremos à biblioteca *Gloss*.

Capítulo 2

Descrição do Problema

A realização do jogo tem por base a leitura do **mapa**, da **posição do boneco** e ainda das **coordenadas das caixas**. Todos estes dados contribuem para realização do jogo. Como tal, o **input** (mapa, posição do boneco e coordenadas das caixas) tem de obedecer a determinadas regras, tais como:

- Todos os mapas de jogo têm de ter sempre uma estrutura retangular;
- '#' representa uma parede;
- ' ' representa uma área livre;
- '.' representa um local de arrumação;
- logo após o mapa, as primeiras coordenadas representam a posição do boneco;
- as restantes coordenadas representam as posições das caixas;
- todas estas coordenadas não se podem sobrepor entre si nem se sobrepor com o mapa (tabuleiro).

Após a verificação de todas estas regras é necessário animar o *Sokoban*, partindo de movimentações nos eixos x e y para: cima, baixo, esquerda e direita.

Primeiramente, é analizada jogoda a jogada a possibilidade de movimentação no mapa e as suas consequências nos objetos para além do jogogador, nomeadamente na movimentação das caixas.

Por conseguinte, todo o jogo é convertido para uma *interface gráfica*, permitindo assim, o utilizador jogar. Nesta parte da tarefa, recorremos à biblioteca *Gloss*.

Capítulo 3

Concepção da Solução

3.1 Estruturas de Dados

Para podermos organizar da melhor forma o código, definimos alguns tipos, facilitando a tanto a escrita como a leitura.

- **type Input = [String]** - lista com todos os dados necessários;
- **type Tabuleiro = [String]** - lista com o mapa em si;
- **type NivelMapa = String** - tabuleiro numa string apenas;
- **type Coords = [String]** - lista de coordenadas;
- **type Posicao = (Int,Int)** - coordenada (x,y);
- **type TickCount = Int** - número de movimentos;
- **type Comandos = String** - string com os comandos, ex: "ULRD";
- **type Linha = String** - linha do tabuleiro;
- **type ComandoSimples = Char** - um comando, ex: 'U';
- **type Bitmaps = [Picture]** - lista com todos os bitmaps necessários para desenhar o mapa e aprensentar ao jogador;
- **type Bomba = Bool** - existência de uma bomba ou não;
- **type Contador = Float** - contador do tipo Float;
- **type Level = String** - numero de um nível ex: "3";
- **type Tema = String** - nome de um tema ex: "Lego";
- **type Highscore = String** - melhor pontuação obtida ex: "324";
- **type Mapa = (Bitmaps,NivelMapa,Comandos,Bomba)** - base do jogo.

3.2 Implementação

3.2.1 Fase Inicial

Na fase inicial de jogo foi necessário perceber quais eram os *inputs* válidos. Estes inputs numa primeira abordagem consistiam apenas num mapa seguido de coordanadas, das quais a primeira seria a posição do boneco e as restantes as posições das caixas.

O jogador irá recorrer às teclas 'W', 'A', 'S' e 'D' para mover o Sokoban. Consequentemente, para melhor perceber o mecanismo de jogo, começamos por desenvolver um código com base na **Tarefa4**. Este código, permitiu a visualizaçãp no terminal da movimentação do Sokoban. Isto é, após o jogador premir uma tecla de jogo, é possível ver a posição final do Sokoban depois de este comando ser executado.

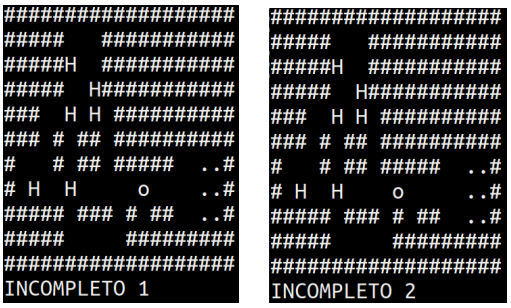


Figura 3.1: Após premir W e A respetivamente

Para que fosse possível adicionar os comandos 'U', 'L', 'D' e 'R' no *input* criámos um programa que permite interagir com ficheiro *tab1.in* (mapa de teste) e com uma adaptação da tarefa 4 que resulta num jogo interativo no terminal.

```
main = do
  hSetEcho stdin False
  hSetBuffering stdin NoBuffering
  c <- getChar
  if c == 'q'
    then do return ()
    else do addcmd c
            hSetEcho stdin True
            callCommand "clear"
            callCommand "./tarefa4_modified < tab1.in"
            main
   where addcmd c
           | c == 'w' || c =='W' = appendFile "tab1.in" "U"
           | c == 'a' || c =='A' = appendFile "tab1.in" "L"
           | c == 's' || c =='S' = appendFile "tab1.in" "D"
           | c == 'd' || c =='D' = appendFile "tab1.in" "R"
           | otherwise           = appendFile "tab1.in" ""
```

3.2.2 Solver

Numa tentativa de aplicar no jogo a funcionalidade do jogador obter uma resolução de um certo nível, foi desenvolvida o programa *solver*.

Para isso é gerada uma série de soluções por ordem crescente de tamanho. São testadas todas as combinações possíveis até que seja encontrada a solução do nível. Como as soluções testadas por ordem de crescente de tamanho, sempre que é encontrada uma solução é garantido que esta é a melhor solução para o mapa que está a ser testado.

A função *comands* gera os comandos que são posteriormente testados pela *trySolution* que vai testando os comandos gerados até encontrar a solução.

```
1 trySolution tab n
2   | jogada tab (comands !! n) == "1" = (comands !! n)
3   | otherwise = trySolution tab (n+1)
```

```
1 comands = [0..] >>= flip replicateM "URLD"
```

Dado que para níveis de elevada complexidade a função leva demasiado tempo a encontrar uma solução, esta mesma funcionalidade não foi adicionada ao jogo. Isto por ser um solver por *brute force*. No entanto, encontra-se em baixo um exemplo para um tabuleiro de pouca complexidade.

```
######
#   ##
#   ##
#   ##
#   .#
######
1 1
3 2

> ./solver < tab1.in

      UURRDLDRR
      51.178735s
```

Figura 3.2: Exemplo de execução da função Solver

3.2.3 Movimentação do Sokoban

Adaptámos a **Tarefa4** de modo a funcionar como um "esqueleto do jogo". Todo este código diz respeito à movimentação do Sokoban.

A função *checkMov* exibe o tabuleiro de jogo após todas as movimentações e as implicações destas movimentações terem sido realizadas.

```
checkMov :: Tabuleiro -> TickCount -> Comandos -> Posicao -> Tabuleiro
checkMov l p c b
    | existemH tabCBoneco == 0  = tabCBoneco ++ ["FIM " ++ show p]
    | null c  = tabCBoneco ++ ["INCOMPLETO " ++ show p]
    | b == move l b [head c] && [head c] /= "B" = checkMov l p (tail c) b
    | otherwise  = checkMov movimentacao (p+1) (tail c) (move l b [head c])
    where tabCBoneco  = colocCaixas (rmBoneco l) [b] 1
          movimentacao = if [head c] == "B"
                         then bomba l b
                         else (mvBoxAux tabCBoneco b [(head c)])
```

Quando é efetuado um movimento, o Sokoban pode alterar de posição ou então permanecer no mesmo local. Para isso, é necessário saber, tendo em conta a posição do Sokoban e a direção para onde se vai mover, qual irá ser o obstáculo que irá encontrar e de que forma proceder de acordo com essa análise.

Foi desenvolvida a seguinta função que permite, para qualquer **movimento**, testar a posição final do Sokoban e, desse modo, fornecer essa mesma posição. (Segue-se um exemplo para o caso de o moviemento ser *Up*, para os restantes é análogo.)

```
move :: Tabuleiro -> Posicao -> Movimento -> Posicao
move tab (a,b) "U"
    | up == '#' || (elem up "IH" && elem up2 "#IH") = (a,b)
    | otherwise                                     = (a,b+1)
    where up = tab !! (length tab - b - 2) !! a
          up2 = tab !! (length tab - b - 3) !! a
    (...)
```

Para verificar se o jogo terminou e, por isso, se todas as caixas estão nos locais de arrumação, a função *existemH* contabiliza os 'H' (caixas) do tabuleiro/mapa sempre que é efetuada uma jogada, sendo que caso se atinja o fim, não são realizadas as jogadas posteriores.

```
existemH :: Tabuleiro -> Int
existemH = foldr ((+) . existemHAux) 0
existemHAux :: Linha -> Int
existemHAux [] = 0
existemHAux (h:t)
    | h == 'H'  = 1 + existemHAux t
    | otherwise = existemHAux t
```

Atendendo que no jogo implementamos a possibilidade de o jogador em determinados mapas poder usar uma *Bomb*, foi necessário acrescentar ao mapa uma parede exterior. Esta parede permite, no caso de o jogador utilizar o recurso *Bomb*, que o jogador não fique com uma abertura no tabuleiro de jogo (*addOutsideWall*).

```
addOutsideWall :: Tabuleiro -> Tabuleiro
addOutsideWall tab = map (++"#") $ reverse $ map (++"#") $ ([head tab]
++ reverse tab ++ [head tab])
```

Para depois na parte gráfica do jogo ser apenas visivel o tabuleiro/mapa, decidimos "remover"partes do mapa desnecessárias. Não se trata necessariamente de uma remoção, mas sim de uma modificação do mapa para que depois seja possivel tornar esta parte do mapa que foi removida num *Blank*, tornando o jogo mais apelativo. Para isto, recorremos a função *transformCar* da **Tarefa2**, que transforma os cardinais redundantes em @, esta função é auxiliada pela *checkCar* que verifica se em torno de cada posição no tabuleiro apenas existem cardinais.

```
transformCar :: Tabuleiro -> Posicao -> Tabuleiro
transformCar tab (a,b)
    | a == (length $ head tab) = transformCar tab (0,b+1)
    | b == (length tab) = tab
    | checkCar tab (a,b) = transformCar correctionT (a+1,b)
    | otherwise = transformCar tab (a+1,b)
    where line = tab !! b
          correctionT = (take b tab) ++ [(take a line) ++ ['@'] ++ (drop (a+1) line)]
        ++ (drop (b+1) tab)
```

```
checkCar :: Tabuleiro -> Posicao -> Bool
checkCar t (a,b) = t !! b !! a == '#' && and [dir, esq, cima, baixo, cimD, cimE,
↪baixD, baixE]
    where dir    | r > 0 && r < w = t !! b !! r `elem` "#@"
                 | otherwise      = True
          (... análogo nas outras direções)
          w      = length (head t) - 1 -- width
          r      = a+1 -- right
```

3.2.4 Interface Gráfica - *Gloss*

O *game.hs* que funciona como num "tradutor"para *Bitmaps* para além de permitir a interação com o jogador. Assim, para além de permitir a visualização de todo o jogo, permite que este seja "jogável".

Nesta parte da tarefa, como já referimos, aumentamos o nível de complexidade do jogo,

isto é, acrescentámos a possibilidade do jogador realizar **Undo** e **Restart**, visualizar o **Score**, poder alterar o **tema**, o recurso à **Bomb** quando disponível, e ainda a navegação entre os diferentes **níveis** do jogo.

Figura 3.3: Interface do jogo Sokoban

Numa primeira fase começamos por desenhar a interface do jogo, isto é, a janela que abre ao executá-lo. Para isso definimos a função *desenhaMapa*, onde estão definidos todos os constituintes da interface. No final é DEVOLVIDA uma picture.

```
1 desenhaMapa :: Mapa -> Picture
2 desenhaMapa (a,tab,cmd,n) = Pictures [border,bground,tabuleiro,tabela,title,
3                               bottombar,lastcmds,hidecmds]
4     where (...)
```

Uns dos constituintes da interface gráfica é o tabuleiro, e é aqui que o jogador mais se vai focar. A função *mapshow* permite que após receber um tabuleiro de jogo converte-lo num conjunto de pictures, formando assim uma picture final que contem apenas o mapa. Para isso são usados *Bitmaps*.

```
1 mapshow :: Tabuleiro -> Bitmaps -> Comandos -> Picture
2 mapshow tab bmps cmd = scale 1.2 1.2 (Pictures [Pictures
3                    (translateLines (mapshowaux (reverse tab) bmps cmd) 0),
4                  if elem 'B' cmd
5                  then Pictures (translateLines (bombshow (reverse tab) bmps cmd) 0)
6                  else Blank])
```

Dado que os *Bitmaps* escolhidos têm uma resolução de 20x20, após todo o tabuleiro ser transcrito para *Bitmaps*, é necessário posicioná-lo linha a linha. Para isso usamos a função *translateLines*, que começa por pegar na linha que constitui a base do mapa e aplica uma translação de 20 unidades por cada nível de linha de forma sucessiva.

```haskell
translateLines :: [Picture] -> Contador -> [Picture]
translateLines [] _ = []
translateLines (h:t) r = ((Translate (0) (r*20) h) : (translateLines t (r+1)))
```

O mapa de jogo é constituido por '#', por 'o', por 'O', por '@', por ' ', por '.' e ainda 'I'. Como tal, é necessário converter este mapa para *Bitmaps*. Para isso aplicámos a função *mapshowaux* que recorre a função *assignbmp* que "traduz"cada um dos constituintes do mapa para o respetivo *Bitmap*.

```haskell
mapshowaux :: Tabuleiro -> Bitmaps -> Comandos -> [Picture]
mapshowaux [] _ _ = []
mapshowaux (h:t) bmps cmd = (Pictures (assignbmp 0 bmps h cmd):(mapshowaux t bmps
↪cmd))
```

Sendo assim, a função *assignbmp* aplica singularmente um *Translate* ao respetivo constituinte. Sendo que é tida em conta a orientação do boneco, de acordo com a última movimentação realizada. Em baixo encontra-se um exerto da função.

```haskell
assignbmp :: Contador -> Bitmaps -> NivelMapa -> Comandos -> [Picture]
assignbmp _ _ [] _ = []
assignbmp r bmps@[pu,pd,pl,pr,w,s,f,b,bs,psu,psd,psl,psr,bomb,logo] (h:t) cmd
| h == 'o' && last cmd == 'U' = ((Translate (r*20) (0) pu ):(assignbmp (r+1) bmps t
↪cmd))
(...)
| h == 'O' && last cmd == 'R' = ((Translate (r*20) (0) psr):(assignbmp (r+1) bmps t
↪cmd))
| h == '#' = ((Translate (r*20) (0) w ):(assignbmp (r+1) bmps t cmd))
| h == 'H' = ((Translate (r*20) (0) b ):(assignbmp (r+1) bmps t cmd))
| h == 'I' = ((Translate (r*20) (0) bs):(assignbmp (r+1) bmps t cmd))
| h == '@' = (assignbmp (r+1) bmps t cmd)
| last cmd == 'B' = (assignbmp (r+1) bmps t cmd)
```

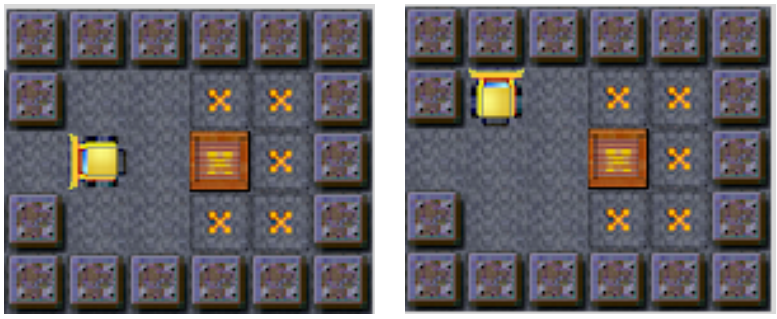


Figura 3.4: Interface do jogo Sokoban

Caso o jogador pretenda utilizar a *Bomb* e caso esta esteja disponível nesse nível, é necessário apresentar a *Bomb* no jogo. Para isso utilizámos a função *bombshow* com o

auxílio da função *placeBomb*, que permite colocar a *Bomb* em cima do resto do tabuleiro na posição do jogador.

```
bombshow :: Tabuleiro -> Bitmaps -> Comandos -> [Picture]
bombshow [] _ _ = []
bombshow (h:t) bmps cmd = (Pictures (placeBomb 0 bmps h cmd) : (bombshow t bmps cmd))
```

```
placeBomb :: Contador -> Bitmaps -> NivelMapa -> Comandos -> [Picture]
placeBomb _ _ [] _ = []
placeBomb r bmps@[pu,pd,pl,pr,w,s,f,b,bs,psu,psd,psl,psr,bomb,logo] (h:t) cmd
    | (h == 'o' || h == 'O') && last cmd == 'B'
  = ((Translate (r*20) (0) bomb ):(placeBomb (r+1) bmps t cmd))
    | otherwise = (placeBomb (r+1) bmps t cmd)
```

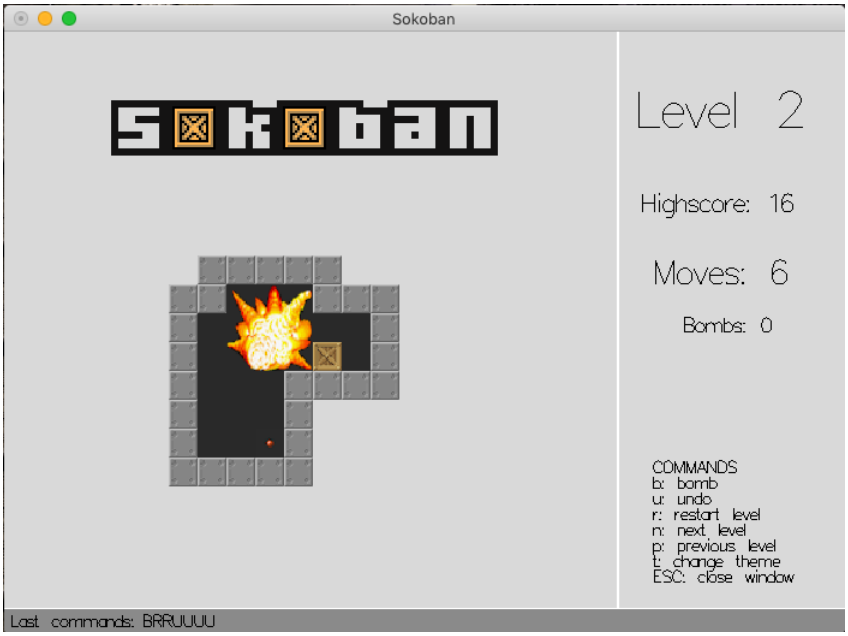


Figura 3.5: Utilização do recurso Bomb

Para a movimentação propriamente dita dentro do tabuleiro é usada a *makemove* que faz a conexão entre a interface gráfica e o esqueleto do jogo (*move*). Em primeiro lugar, é adicionado ao ficheiro *cmds.in* o novo comando. Após isto o programa *move* (referido anteriormente como esqueleto do jogo), processa os comandos no tabuleiro correspondente ao nível atual. Recorrendo ao ficheiro *tab.temp* gerado pela *move* é devolvido o novo tabuleiro para ser "traduzido"pela *desenhaMapa* de modo a ser apresentado ao jogador.

```haskell
makeMov :: Mapa -> ComandoSimples -> IO Mapa
makeMov (bmp,tab,cmds,False) 'B' = return (bmp,tab,cmds,False)
makeMov (bmp,tab,cmds,n) c =
            do appendFile "content/levels/cmds.in" [c]
               callCommand ("./move < " ++ level 'a' ++ " > content/levels/tab.temp")
               tabn <- readFile "content/levels/tab.temp"
               if tab == tabn
                 then return (bmp,tabn,cmds,n)
                 else do if endgame tabn
                           then highscore ((length $ rmBombListMoves cmds) + 1)
                           else return ()
                         if c /= 'B'
                           then do runCommand "afplay content/sounds/move.aiff"
                                   return (bmp,tabn,(cmds++[c]),n)
                           else do runCommand "afplay content/sounds/bomb.mp3"
                                   return (bmp,tabn,(cmds++[c]),False)
```

Na função *makemove* é também verificado o *highsore* caso o jogo tenha acabado, ou seja, é comparado o *score* do jogador com um ficheiro local criado exclusivamente para o efeito que contém o *highscore* anterior. Caso a pontuação obtida seja melhor é alterado esse ficheiro com a pontuação correspondente a esse nível; caso o ficheiro não exista é criado o ficheiro com a nova pontuação obtida.

```haskell
highscore :: Int -> IO ()
highscore a
  | not $ unsafePerformIO (doesFileExist ("content/levels/highscore" ++ lvl a ++ ".sc"))
= writeFile ("content/levels/highscore" ++ lvl a ++ ".sc") (show a)
  | (read (unsafePerformIO $ scores a) :: Int) > a
= do writeFile "content/levels/highscore.sc" (show a)
     renameFile "content/levels/highscore.sc" ("content/levels/highscore" ++ lvl a ++ ".sc")
  | otherwise = return ()
```

Uma das características principais do nosso jogo, é a possibilidade de expansão do jogo por parte do utilizador. Tanto a nível de níveis como a nível de temas. Bastando seguir as instruções para o efeito. Assim, não existem limitações do número de níveis total nem dos temas que o utilizador pretende adicionar.

Para os níveis basta colocar o novo nível na pasta *levels* como "tabN.in"onde N é o número do novo nível. Para um adicionar um tema basta adicionar uma pasta com os bitmaps na directoria correta. Tanto os níveis como os temas são automaticamente adicionados ao jogo assim que este é aberto.

Para uma melhor organização criámos um ficheiro *settings.txt* na pasta *content* onde se encontram todos os dados referentes aos níveis e temas. No caso de o jogador pretender seguir ou regressar para um nível, será alterado o ficheiro *settings.txt* com o nível

Figura 3.6: Diferentes níveis de jogo

correspondente à intenção do utilizador (o nível seguinte ou o anterior). Para isso são utilizadas as funções *nextLvl* e *prevLvl* (análoga à nextLvl) respetivamente.

```
nextLvl :: Mapa -> IO Mapa
nextLvl m = if lvl 'a' >= maxLevel
              then exitSuccess
              else do writeLvl 'n'
                      restart m 1
```

```
changelvl :: Char -> [Level]
changelvl a = (head $ settings a):(head (words $ settings a !! 1) ++ " "
       ++ upLevel ((words $ (settings a) !! 1) !! 1) a) : (tail $ tail $ settings a)
    where upLevel a 'n' = show ((read a :: Int) + 1)
          upLevel a 'p' = show ((read a :: Int) - 1)
```

O jogador tem ainda a possibilidade de poder alterar o *theme* do jogo. No caso do jogador pretender fazer esta alteração, todos os *Bitmaps* irão ser subtituidos e, para isso, utilizámos as funções *nextTheme* e *changeTheme*. Mais uma vez, recorremos ao ficheiro *settings.txt* e alteramos aí o tema selecionado.

```
nextTheme :: Mapa -> IO Mapa
nextTheme (bmp,tab,cmds,n) = do newTheme 'a'
            pu  <- loadBMP (theme 'a' ++ "/playerU.bmp")
            pd  <- loadBMP (theme 'a' ++ "/playerD.bmp")
            ...
            ...
            bomb<- loadBMP "content/bitmaps/bomb.bmp"
            logo<- loadBMP "content/bitmaps/logo.bmp"
            return ([pu,pd,pl,pr,w,s,f,b,bs,psu,psd,psl,psr,bomb,logo],tab,cmds,n)
```

```haskell
changeTheme :: t -> ContadorInt -> [Tema]
changeTheme a n
    | n == (length allThemes - 1) = changeToTheme $ allThemes !! 0
    | themeName a == allThemes !! n = changeToTheme $ allThemes !! (n+1)
    | otherwise = changeTheme a (n+1)
    where changeToTheme b = (take 2 $ settings b)
                            ++[head (words $ settings b !! 2) ++ " " ++ b]
                            ++(drop 3 $ settings b)
```

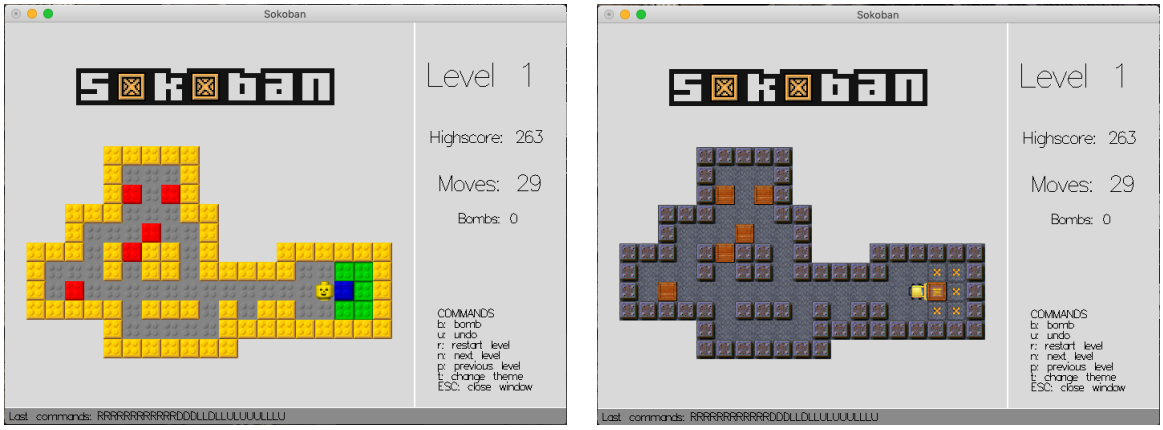


Figura 3.7: Diferentes temas disponíveis

Capítulo 4

Conclusões

Todo o trabalho foi desenvolvido de raiz, sendo que só foi possível perceber o mecanismo de jogo com base na tarefa4.

Após a realização da Movimentação do Sokoban todo o jogo se baseou na interface gráfica. A utilização da ferramenta Gloss permitiu criar, para além do jogo Sokoban, funcionalidades tais como alteração do tema, alteração dos níveis de jogo, utilização do Undo e Restart ainda da visualização do Score.

O código *Solver* foi criado apenas com efeitos de "proof of concept"e, como tal, não teve qualquer funcionalidade no jogo.

Concluímos, que este projecto vai além de apenas um jogo Sokoban, permitiu-nos obter novos conhecimentos tanto a nível de programação e formas de pensar como também de tudo aquilo anexo, isto é, o repositório SVN, a documentação e os testes para o código desenvolvido.