Rush-hour

PSPACE, ANSI, State

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

Jan 05^{th}, 2023

Rush-hour

`https://www.thinkfun.de/products/rush-hour/` Rushhour ist *PSPACE-complete* und damit wahrscheinlich nur in exponentieller Zeit lösbar

State Monad

Die State erlaubt es "mutable" Variablen zu haben (ohne IO)

```
1  data StateT s m a = StateT { runStateT :: s -> m (a,s) }
2
3  instance Functor (StateT s m) where
4    fmap f m = StateT $ \s ->
5      fmap (\ (a,t) -> (f a,t)) $ runStateT m s
6
7  instance Monad (StateT s m) where
8    return a = StateT $ \s -> return (a,s)
9    (>>=) :: m a -> (a -> m b) -> m b
10   m >>= k = StateT $ \s -> do
11     (a,t) <- runStateT m s
12     runStateT (k a) t
```

Monad Transformer

- Man beachte das (kuriose) `m` in `data StateT s m a`
- Hier kann State zusammen mit anderen Monaden eingesetzt werden
- Hierdurch lassen sich komplexe "Echtwelt"-Probleme in logische Bestandteile zerlegen
- Programme geben nun in einzelnen Funktionen "Constraints" an

```
 1  data Player = P { health :: Int, magic :: Int }
 2  data DoA = Dead | Alive
 3  monster :: (Monad m, MonadState Player m) => Int -> m DoA
 4  monster c = do
 5    s <- gets health
 6    if s<c
 7    then do put $ P 0 0
 8            return Dead
 9    else do modify (\P h m -> p (h-c) m)
10            return Alive
```

Generische Puzzle-Spiele

Es handelt sich hierbei um Spiele für die Folgendes gilt:

- Spiele laufen in diskreten Zügen ab
- zu jedem beliebigen Zeitpunkt gibt es wohldefinierte Spielzustände
- aufeinander folgende Zustände lassen sich jeweils exakt beschreiben

Rush-hour ist so ein Spiel, aber auch Sudoko, Schach, go, Mensch ärgere dich nicht, und viele mehr. Wir interessieren uns nur für "1-Spieler" Spiele, da dann nicht "reagiert" werden muss.

Generische Puzzle-Spiele

Gegeben den aktuellen Zustand S eines Spiels, sind drei Probleme zu lösen (teilweise mittels Moves M):

solved $S \to \mathbb{B}$ is *wahr* wenn S ein Lösungszustand ist

move $S \to M \to S$, gegebn S und M wird ein neuer Zustand S generiert

moves $S \to \{M\}$, gegeben S, generiere alle legalen Moves $M_1 \ldots M_S$ für S

Allerdings hängen die Typen von S und M vom jeweiligen Spiel ab: wie organisieren wir das?

Typklassen, Datenfamilien, und Typfamilien

- Typklassen kennen wir schon: `class Puzzle a where`
- bisher aber nur mit zugehörigen Funktionen
- einer Typklassen können aber aber Datentypen zugeordnet werden
- wir können also Funktionen bauen die gegeben einen Typkonstruktor einen anderen Typkonstruktor berechnen
- Damit können dann Datentypen überladen werden

```
1  data family XList :: * -> *
2
3  data instance XList Char = XCons Char (XList Char) | XNil
4
5  data instance XList () = XUnit Int
```

Die Kombination von Klassen und Familien

Insgesamt "5 Elemente" reichen um unsere Puzzles zu beschreiben. Achtung: wir müssen hier *nicht* wissen, dass wir Rush-hour lösen wollen

```
1  class Puzzle p where
2    -- | Der aktuelle Zustand des Puzzles
3    data PState p :: *
4    -- | Ein Schritt zwischen zwei Zustaenden
5    data Move p :: *
6
7    -- | Alle moves von einem State aus
8    moves :: PState p -> [Move p]
9    -- | State nach State mittels Move
10   move :: PState p -> Move p -> PState p
11   -- | Ist das Problem geloest?
12   solved :: PState p -> Bool
```

Es gibt noch einige Hilfstypen

```haskell
-- | Pfade die zu einem State fuehren
type Path p = ([Move p], PState p)

-- | Die "Grenze" sind alle bisher erkundeten Pfade,
-- die potentiell noch zum Ziel fuehren koennen.
type Frontier p = [Path p]

-- | Groesse der aktuellen Frontier
newtype FrontierSz = FSz Int
  deriving (Eq,Ord,Show,Num)

-- | @search@ erlaubt BFS und DFS Suchen.
data SearchTy = BFS | DFS
  deriving (Eq,Ord,Show)
```

und Hilfsfunktionen

```
1  -- | Gegeben einen 'Path', berechne alles 'moves'
2  -- von @q@ aus, wodurch eine Liste von 'Path'
3  -- entsteht. Jeweils mit einem eigenem PState,
4  -- dieser wurde durch 'move' erreicht.
5
6  succs :: Puzzle p => Path p -> [Path p]
7  succs (ms,q) = [(ms ++ [m], move q m) | m <- moves q]
```

und Hilfsfunktionen

```
1  -- | Wir koennen Constraints auch zusammen fassen
2  type SolveTy m p = ( Monad m, MonadState FrontierSz m
3                     , Puzzle p, Eq (PState p) )
4
5  -- | Loese ein Puzzle Problem durch Wahl der Suchmethode
6  -- und des Startzustandes.
7  solve :: SolveTy m p => SearchTy -> PState p
8        -> m (Maybe [Move p])
9  solve sty q = search sty [] [([],q)]
10
11 -- | Standard-Runner, extrahiert die Groesse der Grenze
12 runSolve sty q = runState (solve sty q) (FSz 0)
```

Breiten- und Tiefensuche zur Lösung

```
search :: (Monad m, MonadState FrontierSz m
           , Puzzle p, Eq (PState p))
  => SearchTy      -- ^ DFS oder BFS
  -> [PState p]    -- ^ alle bisher besuchten States
  -> Frontier p    -- ^ momentan aktive Pfade
  -> m (Maybe [Move p])

-- Keine Frontier mehr zu testen, keine Lsg
search sty qs [] = pure Nothing

search sty qs (p@(ms,q):ps)
-- wir haben die erste Lsg gefunden
  | solved q     = pure $ Just ms
-- "q" ist schon Teil elaborierter State's
  | q `elem` qs = search sty qs ps

(... unten weiter)
```

Breiten- und Tiefensuche zur Lösung

```
search :: (Monad m, MonadState FrontierSz m
          , Puzzle p, Eq (PState p))
  => SearchTy      -- ^ DFS oder BFS
  -> [PState p]    -- ^ alle bisher besuchten States
  -> Frontier p    -- ^ die zum Ziel fuehren koennten
  -> m (Maybe [Move p])

(von oben ...)

-- teste weitere Pfade
search sty qs (p@(ms,q):ps) = do
  let xs = succs p
  modify (+1)
  search sty (q:qs) $ case sty of
    -- erst schon aktive Pfade, dann tiefer gehen
    BFS -> ps ++ xs
    -- erst tiefer gehen, dann schon aktive Pfade
    DFS -> xs ++ ps
```

Spielkoordinaten

```
 1     1   2   3   4   5   6
 2
 3     8   9  10  11  12  13
 4
 5    15  16  17  18  19  20 ==> -- Ausgang
 6
 7    22  23  24  25  26  27
 8
 9    29  30  31  32  33  34
10
11    36  37  38  39  40  41
```

Definiere und löse Rush-hour

```
1  -- | Noetig fuer die Typ-klasse
2  data Rushhour = Rushhour
3
4  -- | Zellen sind einfach 'Int's
5  type Cell = Int
6
7  -- | Grid ist eine Liste von Zell-paaren
8  -- fuer erste, letzte Zelle
9  type Grid = [(Cell,Cell)]
10
11 -- | Auto-Index
12 newtype VehicleIx = VIx Int
13   deriving (Eq,Ord,Show,Enum,Num)
```

```haskell
-- | Gegeben ein Gridd, welche Zellen sind belegt?
occupied :: Grid -> [Cell]
occupied = concatMap fillcells

-- | Autos entweder horizontal @(3,4)@ oder
-- vertical @(3,10)@; 1 oder mehr Zellen
fillcells (r,f) = if r > f-7 then [r..f] else [r,r+7..f]

-- | Ist ein Auto horizontal?
isHoriz (r,f) = r>f-7

-- | Transformation von Grid nach x,y Koordinaten
toXY :: (Int,Int) -> [(Int,Int)]
toXY = map ((\(l,r) -> (l+1,r)) . (`divMod` 7)) . fillcells

-- | Welche Zellen sind nicht belegt?
freecells :: Grid -> [Cell]
freecells g = allcells \\ occupied g

-- | Alle Zellen des Spielfeldes
allcells :: [Cell]
allcells = [ c | c <- [1..41], c `mod` 7 /= 0 ]
```

```
1  -- | Gegeben ein Auto, was sind die benachbarten Zellen
2  -- potentiell ausserhald des Spielfeldes!
3  adjs :: (Cell,Cell) -> [Cell]
4  adjs (r,f) = if r > f-7 then [f+1,r-1] else [f+7,r-7]
5
6  -- | Bewege ein Auto einen Schritt.
7  adjust :: (Cell,Cell) -> Cell -> (Cell,Cell)
8  adjust (r,f) c
9    -- horizontale Autos nach rechts oder links
10   | r > f-7 = if c > f then (r+1,c) else (c,f-1)
11   -- vertikale Autos nach oben oder unten
12   | otherwise = if c < r then (c,f-7) else (r+7,c)
```

Assoziierte Datentypen

```
1  -- | Loese Rushhour, indem eine Instanz von Puzzle
2  -- geschrieben wird
3  instance Puzzle Rushhour where
4    -- | Der State ist durch das jeweils aktive Grid gegeben
5    newtype PState Rushhour = Rstate { rstate :: Grid }
6      deriving (Eq)
7    -- | Ein Move schiebt ein Auto, wir brauchen den Index
8    -- des Autos und die Ziel-Zelle
9    newtype Move Rushhour = Rmove {rmove :: (VehicleIx,Cell)}
10     deriving (Show)
11
12 (...)
```

Moves

```
 1  instance Puzzle Rushhour where
 2    newtype PState Rushhour = Rstate Grid
 3      deriving (Eq)
 4    newtype Move Rushhour = Rmove (VehicleIx,Cell)
 5      deriving (Show)
 6
 7    -- | Legale Moves, gegeben einen State
 8    moves :: PState Rushhour -> [Move Rushhour]
 9    moves (Rstate g) =
10      let fs = freecells g
11      in  [ Rmove (v,c)
12          | (v,i) <- zip [0..] g
13          , c <- adjs i
14          , c `elem` fs ]
15
16  adjs :: (Cell,Cell) -> [Cell]
17  adjs (r,f) = if r > f-7 then [f+1,r-1] else [f+7,r-7]
```

Move

```
1  instance Puzzle Rushhour where
2    newtype PState Rushhour = Rstate Grid
3      deriving (Eq)
4    newtype Move Rushhour = Rmove (VehicleIx,Cell)
5      deriving (Show)
6
7   -- Ein move ist ein Update von genau einem Auto via Index
8    move :: PState Rushhour->Move Rushhour->PState Rushhour
9    move (Rstate g) (Rmove (VIx v,c)) =
10     let (g1,i:g2) = splitAt v g
11     in  Rstate $ g1 ++ adjust i c : g2
```

```
 1  instance Puzzle Rushhour where
 2    newtype PState Rushhour = Rstate Grid
 3      deriving (Eq)
 4    newtype Move Rushhour = Rmove (VehicleIx,Cell)
 5      deriving (Show)
 6
 7    -- Teste ob das Rechte Ende vom Auto in der
 8    -- End-Zelle steht
 9    solved :: PState Rushhour -> Bool
10    solved (Rstate g) = snd (head g) == 20
```

DFS vs. BFS

BFS: Breitensuche

- erweitert in jedem "moves" alle noch aktiven Pfade
- deren Anzahl exponentiell wächst
- insbesondere wächst die Anzahl der Intermediär-Zustände exponentiell, da *alle* Pfade der Länge $n-1$ besucht werden, bevor der Lösungspfad der Länge n gegangen
- dafür ist aber n minimal

DFS: Tiefensuche

- erweitert in jedem "moves" alle noch aktiven Pfade
- deren Anzahl auch exponentiell wächst
- es wächst auch die Anzahl der Intermediärzustände exponentiell, aber wahrscheinlich langsamer
- insbesondere wird der "lexikographisch" (was immer das hier bedeutet) erste Pfad gefunden, der löst
- die Pfadlänge n kann aber sehr lang werden

Zahlen, bitte!

- Es sind drei Probleme g_0, g_1, g_2 gegeben
- |Pfad| ist die Länge der Lösung
- |States| wie viele Zwischen-States besucht wurden
- sec. die Zeit um die Lsg. zu finden

	BFS			DFS		
	|Pfad|	|States|	sec.	|Pfad|	|States|	sec.
g_0	8	646	0.10	67	66	0.00
g_1	35	2757	0.55	1229	2094	0.31
g_2	82	3051	0.61	1306	1675	0.17

```
plotMoves :: SearchTy -> PState Rushhour -> IO ()
plotMoves sty grid = do
  setSGR [SetColor Foreground Dull Blue]
  print (rstate grid)
  before <- getCurrentTime
  let solu = runSolve sty grid
  after <- snd solu `seq` getCurrentTime
  printf "%s␣seconds\n" (show $ diffUTCTime after before)
  case runSolve sty grid of
    (Nothing, FSz depth) -> do
      setSGR [SetColor Foreground Vivid Red]
      printf "no␣solution␣with␣final␣frontier␣size:␣%d\n" depth
    (Just sol, FSz depth) -> do
      setSGR [SetColor Foreground Vivid Blue]
      printf "found␣solution␣with␣frontier␣size:␣%d\n" depth
      let is = scanl move grid sol
      hSetEcho stdin False
      whileGrid is 0
      hSetEcho stdin True
  setSGR [Reset]  -- reset to default colour scheme
```

```
 1  whileGrid :: [PState Rushhour] -> Int -> IO ()
 2  whileGrid is k = do
 3    printf "%5d␣of␣%5d\n" (k+1) (length is)
 4    plotGrid $ is !! k
 5    cursorUp 13
 6    c <- getChar
 7    if | c == 'q' -> cursorDown 13 >> pure ()
 8       | c == 'p' -> whileGrid is (max 0 $ k-1)
 9       | c == 'P' -> whileGrid is (max 0 $ k-25)
10       | c == 'n' -> whileGrid is (min (k+1) $ length is -1)
11       | c == 'N' -> whileGrid is (min (k+25) $ length is -1)
12       | otherwise -> whileGrid is k
```

```
plotGrid :: PState Rushhour -> IO ()
plotGrid g = do
  let occ = occupied (rstate g)
      (x,y) = head (rstate g)
      colors :: M.Map Int [SGR]
      colors = M.fromList . concatMap (\(c,l)->map (,l) c)
        $ zip (map fillcells $ rstate g) (cycle others)
      pix sgr i = setSGR [Reset] >> putStr "␣" >>
        setSGR sgr >> printf "%2d" i
  forM_ [1..6] $ \r -> do
    forM_ [1..6] $ \c -> do
      let ix = (r-1)*7 + (c::Int)
      if | ix `elem` [x,y] -> do
             pix [SetColor Foreground Vivid Red
                 ,SetColor Background Vivid White] ix
         | Just clr <- colors M.!? ix -> do
             pix clr ix
         | otherwise ->
             pix [SetColor Foreground Dull White] ix
      setSGR [Reset]
      putStrLn "\n"
  setSGR [Reset]
```

Hausaufgabe

`https://www.michaelfogleman.com/rush/`

- Download der Datenbank
- Parsen der Datenbank
- Konvertierung in das Format hier *oder* eigene Rush-hour Definition
- Ausprobieren
- Diese Version kennt auch "Mauern" die nicht beweglich sind!

Wer sich wirklich austoben möchte: generiert eigene Probleme für "beliebige" $N \times N$ große Instanzen.
Oder einen Sudoko Solver schreiben