

# Sudoku Problem Solving and Problem Generation

# Implementation of a Sudoku puzzle solver

- Start from a (mostly) formal specification
- Constraint resolution
- Depth-first search
  - Search trees
  - Abstract representation of search
  - DFS versus BFS
- Random generator for Sudoku problems
  - Random deleting values from valid Sudoku grids.

# Specifying Sudoku solving

- A Sudoku is a 9x9 matrix containing numbers in  $\{1, \dots, 9\}$ , *possibly* containing blanks, satisfying certain constraints:
  - Each row contains each number in  $\{1, \dots, 9\}$
  - Each column contains each number in  $\{1, \dots, 9\}$
  - Each subgrid[i][j], with i,j ranging over 1..3, 4..6 and 7..9 contains each number in  $\{1 \dots 9\}$
- A Sudoku problem is a Sudoku containing blanks, otherwise satisfying the Sudoku constraints → is a partial Sudoku matrix
  - The values in each row/column/subgrid should be in  $\{1, \dots, 9\}$  and should all be different
- A Sudoku solver transforms the problem into a solution.

# Sudoku constraints and injectivity

- A function  $f$  is injective if it preserves distinction: if  $x \neq y$  then  $f(x) \neq f(y)$ 
  - $f$  is injective if  $f(x) == f(y) \rightarrow x == y$
  - injective  $xs = \text{nub } xs == xs$
- Let's rewrite the Sudoku constraints for a Sudoku matrix  $f[i][j]$ :
  - Row members different  $\rightarrow [f[i,j] \mid j \leftarrow [1\dots 9]]$  should not have duplicates
  - Column members different  $\rightarrow [f[i,j] \mid i \leftarrow [1\dots 9]]$  should not have duplicates
  - Subgrid members different  $\rightarrow [f[i,j] \mid i \leftarrow [1\dots 3], j \leftarrow [1\dots 3]]$  should not have duplicates, similar for the other subgrids
- datatypes?

# Sudoku constraints and injectivity

- A function  $f$  is injective if it preserves distinction: if  $x \neq y$  then  $f(x) \neq f(y)$ 
  - $f$  is injective if  $f(x) == f(y) \rightarrow x == y$
- Let's rewrite the Sudoku constraints for a Sudoku matrix  $f[i][j]$ :
  - Row members different  $\rightarrow [f[i,j] \mid j \leftarrow [1..9]]$  should not have duplicates
  - Column members different  $\rightarrow [f[i,j] \mid i \leftarrow [1..9]]$  should not have duplicates
  - Subgrid members different  $\rightarrow [f[i,j] \mid i \leftarrow [1..3], j \leftarrow [1..3]]$  should not have duplicates, similar for the other subgrids

```
type Row    = Int
type Column = Int
type Value  = Int
type Grid   = [[Value]]
type Constraint = (Row,Column,[Value])
```

```
positions, values :: [Int]
positions = [1..9]
values    = [1..9]
```

```
blocks :: [[Int]]
blocks = [[1..3],[4..6],[7..9]]
```

# Displaying a Sudoku (problem)

```
showVal :: Value -> String
```

```
showVal 0 = " "
```

```
showVal d = show d
```

```
showRow :: [Value] -> IO()
```

```
showRow [a1,a2,a3,a4,a5,a6,a7,a8,a9] =
```

```
do  putChar '|'      ; putChar ' '
    putStr (showVal a1) ; putChar ' '
    putStr (showVal a2) ; putChar ' '
    putStr (showVal a3) ; putChar ' '
    putChar '|'      ; putChar ' '
    putStr (showVal a4) ; putChar ' '
    putStr (showVal a5) ; putChar ' '
    putStr (showVal a6) ; putChar ' '
    putChar '|'      ; putChar ' '
    putStr (showVal a7) ; putChar ' '
    putStr (showVal a8) ; putChar ' '
    putStr (showVal a9) ; putChar ' '
    putChar '|'      ; putChar '\n'
```

```
showGrid :: Grid -> IO()
```

```
showGrid [as,bs,cs,ds,es,fs,gs,hs,is] =
```

```
do  putStrLn ("+-----+-----+-----+")
    showRow as; showRow bs; showRow cs
    putStrLn ("+-----+-----+-----+")
    showRow ds; showRow es; showRow fs
    putStrLn ("+-----+-----+-----+")
    showRow gs; showRow hs; showRow is
    putStrLn ("+-----+-----+-----+")
```

# Sudoku type and some conversions

```
type Sudoku = (Row,Column) -> Value
```

```
sud2grid :: Sudoku -> Grid
```

```
sud2grid s =
```

```
  [ [ s (r,c) | c <- [1..9] ] | r <- [1..9] ]
```

```
grid2sud :: Grid -> Sudoku
```

```
grid2sud gr = \ (r,c) -> pos gr (r,c)
```

```
  where
```

```
    pos :: [[a]] -> (Row,Column) -> a
```

```
    pos gr (r,c) = (gr !! (r-1)) !! (c-1)
```

```
bl :: Int -> [Int]
```

```
bl x = concat $ filter (elem x) blocks
```

```
subGrid :: Sudoku -> (Row,Column) -> [Value]
```

```
subGrid s (r,c) =
```

```
  [ s (r',c') | r' <- bl r, c' <- bl c ]
```

```
showSudoku :: Sudoku -> IO()
```

```
showSudoku = showGrid . sud2grid
```

# Free Values

```
freeInSeq :: [Value] -> [Value]
freeInSeq seq = values \\ seq
```

```
freeInRow :: Sudoku -> Row -> [Value]
freeInRow s r =
  freeInSeq [ s (r,i) | i <- positions ]
```

```
freeInColumn :: Sudoku -> Column -> [Value]
freeInColumn s c =
  freeInSeq [ s (i,c) | i <- positions ]
```

```
freeInSubgrid :: Sudoku -> (Row,Column) -> [Value]
freeInSubgrid s (r,c) = freeInSeq (subGrid s (r,c))
```

```
freeAtPos :: Sudoku -> (Row,Column) -> [Value]
freeAtPos s (r,c) =
  (freeInRow s r)
  `intersect` (freeInColumn s c)
  `intersect` (freeInSubgrid s (r,c))
```



# Injectivity by row, column, subgrid

```
injective :: Eq a => [a] -> Bool
injective xs = nub xs == xs
```

```
rowInjective :: Sudoku -> Row -> Bool
rowInjective s r = injective vs where
  vs = filter (/= 0) [ s (r,i) | i <- positions ]
```

```
colInjective :: Sudoku -> Column -> Bool
colInjective s c = injective vs where
  vs = filter (/= 0) [ s (i,c) | i <- positions ]
```

```
subgridInjective :: Sudoku -> (Row,Column) -> Bool
subgridInjective s (r,c) = injective vs where
  vs = filter (/= 0) (subGrid s (r,c))
```

```
consistent :: Sudoku -> Bool
consistent s = and $
  [ rowInjective s r | r <- positions ] ++ [ colInjective s c | c <- positions ]
  ++
  [ subgridInjective s (r,c) | r <- [1,4,7], c <- [1,4,7]]
```

# Extending a Sudoku to search for a solution

- Fill in a value in a blank (new) position → remember update?

```
extend :: Sudoku -> ((Row,Column),Value) -> Sudoku  
extend = update
```

```
update :: Eq a => (a -> b) -> (a,b) -> a -> b  
update f (y,z) x = if x == y then z else f x
```

- Build a search tree, a node is a pair consisting of a Sudoku and its Constraints

```
type Sudoku = (Row,Column) -> Value  
type Constraint = (Row,Column,[Value])  
type Node = (Sudoku,[Constraint])
```

```
showNode :: Node -> IO()  
showNode = showSudoku . fst
```

# A Sudoku Solution

- A Sudoku is solved when there are no more empty cells.

```
solved :: Node -> Bool  
solved = null . snd
```

- Successors in the search tree: Sudokus extended with values listed in the constraint for the next empty cell.
  - Prune values that are no longer possible from the constraint list
  - Constraints are ordered by length of possible values (shortest-first)

```
type Node = (Sudoku,[Constraint])  
extend :: Sudoku -> ((Row,Column),Value) -> Sudoku
```

```
extendNode :: Node -> Constraint -> [Node]  
extendNode (s,constraints) (r,c,vs) =  
  [(extend s ((r,c),v),  
    sortBy length3rd $  
      prune (r,c,v) constraints) | v <- vs ]
```

# Search Trees

```
data Tree a = T a [Tree a] deriving (Eq,Ord,Show)
```

- Growing a Tree

```
grow :: (node -> [node]) -> node -> Tree node  
grow step seed = T seed (map (grow step) (step seed))
```

Q: what is the difference between `grow (\x -> if x `mod` 2 == 0 then [x+1, x+1] else [x]) 0` and `grow (\x -> if x `mod` 2 == 0 then [x+1, x+1] else []) 0`?

Q: if `count $ grow (\x -> if x < 2 then [x+1, x+1] else []) 0` has value 7, what is the value of `count $ grow (\x -> if x < 6 then [x+1, x+1] else []) 0`? How would you generalize it? Why?

# Search Trees

```
data Tree a = T a [Tree a] deriving (Eq,Ord,Show)
```

- Growing a Tree

```
grow :: (node -> [node]) -> node -> Tree node
grow step seed = T seed (map (grow step) (step seed))
```

Q: what is the difference between `grow (\x -> if x `mod` 2 == 0 then [x+1, x+1] else [x]) 0` and `grow (\x -> if x `mod` 2 == 0 then [x+1, x+1] else []) 0`?

A: the first Tree is infinite

Q: if `count $ grow (\x -> if x < 2 then [x+1, x+1] else []) 0` has value 7, what is the value of `count $ grow (\x -> if x < 6 then [x+1, x+1] else []) 0`? How would you generalize it? Why?

A: The structure is a balanced binary branching tree of depth 6. A balanced binary tree of depth  $n$  has  $2^n - 1$  internal nodes and  $2^n$  leaf nodes, which makes  $2^{n+1} - 1$  nodes in total. For a balanced binary tree of depth 6 this gives  $2^7 - 1 = 127$  nodes.

# Depth-first search

## Generic algorithm for depth-first search

```
search :: (node -> [node])  
      -> (node -> Bool) -> [node] -> [node]  
search children goal [] = []  
search children goal (x:xs)  
  | goal x    = x : search children goal xs  
  | otherwise = search children goal ((children x) ++ xs)
```

Q: What does the first argument represent? `children :: node -> [node]`

A: The `step` function which, given a seed, grows a tree

# Depth-first search

## Generic algorithm for depth-first search

```
search :: (node -> [node])  
      -> (node -> Bool) -> [node] -> [node]  
search children goal [] = []  
search children goal (x:xs)  
  | goal x    = x : search children goal xs  
  | otherwise = search children goal ((children x) ++ xs)
```

Q: What does the second argument represent? `goal :: node -> Bool`

# Depth-first search

## Generic algorithm for depth-first search

```
search :: (node -> [node])  
      -> (node -> Bool) -> [node] -> [node]  
search children goal [] = []  
search children goal (x:xs)  
  | goal x    = x : search children goal xs  
  | otherwise = search children goal ((children x) ++ xs)
```

Q: What does the second argument represent? `goal :: node -> Bool`

A: The desired property of a node



# Depth-first search

## Generic algorithm for depth-first search

```
search :: (node -> [node])  
       -> (node -> Bool) -> [node] -> [node]  
search children goal [] = []  
search children goal (x:xs)  
  | goal x    = x : search children goal xs  
  | otherwise = search children goal ((children x) ++ xs)
```

Q: how would you change the above definition to obtain a breadth-first search?

# Depth-first search

## Generic algorithm for depth-first search

```
search :: (node -> [node])  
       -> (node -> Bool) -> [node] -> [node]  
search children goal [] = []  
search children goal (x:xs)  
  | goal x    = x : search children goal xs  
  | otherwise = search children goal ((children x) ++ xs)
```

Q: how would you change the above definition to obtain a breadth-first search?

A: Tip: `xs ++ (children x)`

Try:

```
*Lecture5>search (\x -> if x<6 then [2*x, 2*x+1] else []) (\x -> (odd x) && (x>1)) [1]
```

```
*Lecture5>searchBFS (\x -> if x<6 then [2*x, 2*x+1] else []) (\x -> (odd x) && (x>1)) [1]
```

# Depth-first search

## Generic algorithm for depth-first search

```
search :: (node -> [node])
      -> (node -> Bool) -> [node] -> [node]
search children goal [] = []
search children goal (x:xs)
  | goal x    = x : search children goal xs
  | otherwise = search children goal ((children x) ++ xs)
```

Try:

```
*Lecture5>search (\x -> if x<6 then [2*x, 2*x+1] else []) (\x -> (odd x) && (x>1)) [1]
```

**[9,5,3]**

```
*Lecture5>searchBFS (\x -> if x<6 then [2*x, 2*x+1] else []) (\x -> (odd x) && (x>1)) [1]
```

**[3,5,9]**

Try on your own and find an explanation:

```
*Lecture5>searchBFS (\x -> if x<6 then [2*x-1, 2*x+2] else []) (\x -> even x) [1]
```

```
*Lecture5>search (\x -> if x<6 then [2*x-1, 2*x+2] else []) (\x -> even x) [1]
```

# Search for a (Sudoku) solution

```
solveNs :: [Node] -> [Node]
solveNs = search succNode solved
```

```
succNode :: Node -> [Node]
succNode (s,[]) = []
succNode (s,p:ps) = extendNode (s,ps) p
```

- Showing (off) your solution

```
solveAndShow :: Grid -> IO[()]
solveAndShow gr = solveShowNs (initNode gr)
```

```
solveShowNs :: [Node] -> IO[()]
solveShowNs = showNs . solveNs
```

```
showNs :: showNs = sequence . fmap showNode
```

Q: what is the type of `showNs`?

# Search for a (Sudoku) solution

```
solveNs :: [Node] -> [Node]
solveNs = search succNode solved
```

```
succNode :: Node -> [Node]
succNode (s,[]) = []
succNode (s,p:ps) = extendNode (s,ps) p
```

- Showing (off) your solution

```
solveAndShow :: Grid -> IO[()]
solveAndShow gr = solveShowNs (initNode gr)
```

```
solveShowNs :: [Node] -> IO[()]
solveShowNs = showNs . solveNs
```

```
showNs :: showNs = sequence . fmap showNode
```

Q: what is the type of `showNs`? What will `showNs (initNode gr)` print?

A: `showNs :: Traversable t => t Node -> IO (t ())`

# Generating a Sudoku problem

- Generate random constraints

```
getRandomCnstr :: [Constraint] -> IO [Constraint]
getRandomCnstr cs = getRandomItem (f cs)
  where f [] = []
        f (x:xs) = takeWhile (sameLen x) (x:xs)
```

- Generate a random successor node

```
rsuccNode :: Node -> IO [Node]
rsuccNode (s,cs) = do xs <- getRandomCnstr cs
  if null xs
    then return []
  else return
    (extendNode (s,cs\\xs) (head xs))
```

- Find a random solution

```
rsolveNs :: [Node] -> IO [Node]
rsolveNs ns = rsearch rsuccNode solved (return ns)
```

```
genRandomSudoku :: IO Node
genRandomSudoku = do [r] <- rsolveNs [emptyN]
  return r
randomS = genRandomSudoku >>= showNode
```

# Generating a Sudoku problem

- Erase a position from a Sudoku

```
eraseS :: Sudoku -> (Row,Column) -> Sudoku
eraseS s (r,c) (x,y) | (r,c) == (x,y) = 0
                    | otherwise      = s (x,y)
```

- Erase a position from a Node

```
eraseN :: Node -> (Row,Column) -> Node
eraseN n (r,c) = (s, constraints s)
  where s = eraseS (fst n) (r,c)
```

- Erase positions until the result becomes ambiguous  $\rightarrow$  a minimal node with a unique solution.

```
minimalize :: Node -> [(Row,Column)] -> Node
minimalize n [] = n
minimalize n ((r,c):rcs) | uniqueSol n' = minimalize n' rcs
                        | otherwise     = minimalize n rcs
  where n' = eraseN n (r,c)
```

# Generating a Sudoku problem

Use a minimized randomly generated Sudoku to generate a Sudoku problem

```
genProblem :: Node -> IO Node
genProblem n = do ys <- randomize xs
               return (minimize n ys)
  where xs = filledPositions (fst n)
```

```
main :: IO ()
main = do [r] <- rsolveNs [emptyN]
         showNode r
         s <- genProblem r
         showNode s
```



# Further reading

Felgenhauer, Bertram, and Frazer Jarvis. 2006. "Mathematics of Sudoku I."

Rosenhouse, Jason, and Laura Taalman. 2011. *Taking Sudoku Seriously: The Math Behind the World's Most Popular Pencil Puzzle*. Oxford University Press.

Russell, Ed, and Frazer Jarvis. 2006. "Mathematics of Sudoku II."