# Introduction to Programming: Lecture 20

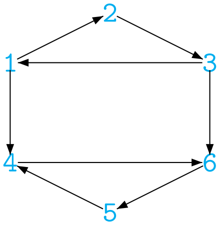
K Narayan Kumar

Chennai Mathematical Institute

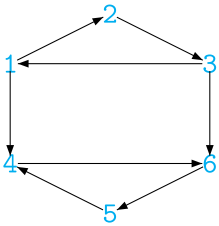
http://www.cmi.ac.in/~kumar

24 October 2013

# Graphs



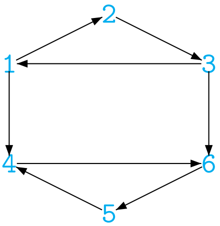
# Graphs



- Represent edges in the graph as a function

```
type Vert = Int
maxvert = 6
edge :: Vert -> Vert -> Bool
edge 1 2  = True
edge 1 4  = True
...
edge 5 6  = True
edge  _   _   = False
```

# Graphs



- Represent edges in the graph as a function

```
type Vert = Int
maxvert = 6
edge :: Vert -> Vert -> Bool
edge 1 2  = True
edge 1 4  = True
...
edge 5 6  = True
edge  _   _   = False
```

- Goal: define `reachable ::  Vert -> [Vert]`

# Graphs . . .

- Inductive definition of `reachable v`

# Graphs . . .

- Inductive definition of `reachable v`
  - `v` is reachable from `v`
  - `x` is reachable from `v` and `edge x y` then `y` is also reachable from `v`.

# Graphs . . .

- Inductive definition of `reachable v`
  - `v` is reachable from `v`
  - `x` is reachable from `v` and `edge x y` then `y` is also reachable from `v`.
- Cannot directly translate this definition into Haskell
- `extend` picks up the neighbours of a given vertex.

```
extend :: Vert -> [Vert]
extend v = [w | w <- [1..maxvert], edge v w]
```

# Reachability ...

- Now, vertices reachable from `v` can be identified as

```
extendall :: [Vert] -> [Vert]
extendall = concatMap extend
```

# Reachability ...

- Now, vertices reachable from `v` can be identified as

```
extendall :: [Vert] -> [Vert]
extendall = concatMap extend
```

- Computes all vertices reachable in one step.

# Reachability ...

- Now, vertices reachable from `v` can be identified as

```
extendall :: [Vert] -> [Vert]
extendall = concatMap extend
```

- Computes all vertices reachable in one step.
- Find all vertices reachable in $\leq$ `maxvert` steps.

# Reachability ...

- Now, vertices reachable from `v` can be identified as

```
extendall :: [Vert] -> [Vert]
extendall = concatMap extend
```

- Computes all vertices reachable in one step.
- Find all vertices reachable in $\leq$ `maxvert` steps.

```
reachable v = concat (take maxvert
                       (iterate extendall [v]))
```

# Reachability ...

- Now, vertices reachable from `v` can be identified as

```
extendall :: [Vert] -> [Vert]
extendall = concatMap extend
```

- Computes all vertices reachable in one step.
- Find all vertices reachable in $\leq$ `maxvert` steps.

```
reachable v = concat (take maxvert
                      (iterate extendall [v]))
```

- There are repetitions at each level.

# Reachability …

- Now, vertices reachable from `v` can be identified as

```
extendall :: [Vert] -> [Vert]
extendall = concatMap extend
```

- Computes all vertices reachable in one step.
- Find all vertices reachable in $\leq$ `maxvert` steps.

```
reachable v = concat (take maxvert
                       (iterate extendall [v]))
```

- There are repetitions at each level.
- Use `Set` instead of lists.

# Reachability ...

- Now, vertices reachable from `v` can be identified as

```
extendall :: [Vert] -> [Vert]
extendall = concatMap extend
```

- Computes all vertices reachable in one step.
- Find all vertices reachable in $\leq$ `maxvert` steps.

```
reachable v = concat (take maxvert
                       (iterate extendall [v]))
```

- There are repetitions at each level.
- Use `Set` instead of lists.

  or remove duplicates after each level is generated.

# Without Duplicates

```
reachable v = concat (take maxvert
                       (iterate (remDup.extendall) [v])
```

# Without Duplicates

```
reachable v = concat (take maxvert
                       (iterate (remDup.extendall) [v])
```

where `remDup` removes duplicates in a list.

# Avoiding duplicate `expand`s

- Vertices repeat at different levels. Can we do better?

# Avoiding duplicate `expand`s

- Vertices repeat at different levels. Can we do better?
- Why not `expand` only vertices that have not been expanded already?

# Avoiding duplicate `expand`s

- Vertices repeat at different levels. Can we do better?
- Why not `expand` only vertices that have not been expanded already?
- Keep two lists,
  - one with elements that are reachable and which have been expanded
  - one with elements that are reachable and which have not been expanded yet.

# Avoiding duplicate `expand`s

- Vertices repeat at different levels. Can we do better?
- Why not `expand` only vertices that have not been expanded already?
- Keep two lists,
  - one with elements that are reachable and which have been expanded
  - one with elements that are reachable and which have not been expanded yet.
- In each iteration pick an element from the second list.

# Avoiding duplicate `expand`s

- Vertices repeat at different levels. Can we do better?
- Why not `expand` only vertices that have not been expanded already?
- Keep two lists,
  - one with elements that are reachable and which have been expanded
  - one with elements that are reachable and which have not been expanded yet.
- In each iteration pick an element from the second list.
  - Expand the element and move it to the first list.
  - Add all new elements in the expansion to the second list.

# Avoiding duplicate `expand`s

- Vertices repeat at different levels. Can we do better?
- Why not `expand` only vertices that have not been expanded already?
- Keep two lists,
  - one with elements that are reachable and which have been expanded
  - one with elements that are reachable and which have not been expanded yet.
- In each iteration pick an element from the second list.
  - Expand the element and move it to the first list.
  - Add all new elements in the expansion to the second list.
- Continue till the second list is empty

# Avoiding duplicate `expand`s

- Vertices repeat at different levels. Can we do better?
- Why not `expand` only vertices that have not been expanded already?
- Keep two lists,
  - one with elements that are reachable and which have been expanded
  - one with elements that are reachable and which have not been expanded yet.
- In each iteration pick an element from the second list.
  - Expand the element and move it to the first list.
  - Add all new elements in the expansion to the second list.
- Continue till the second list is empty

  This is the function `step`.

# Avoiding repetition

```
step:: ([Vert], [Vert]) -> ([Vert], [Vert])
step  (s,[])  =  s
step  (s,x:t) = (x:s,nt)
 where
  tmp = extend x
  nt = t ++ filter g tmp
  g y = not (elems y s) && not (elems y t)
```

# Avoiding repetition

```
step:: ([Vert], [Vert]) -> ([Vert], [Vert])
step  (s,[])  =  s
step  (s,x:t) = (x:s,nt)
 where
  tmp = extend x
  nt = t ++ filter g tmp
  g y = not (elems y s) && not (elems y t)
```

- Iterate `step` to get the answer.

```
iterstep (s,t)
  | (t == []) = (s,[])
  | otherwise = iterstep  (step (s,t))

reachable v = iterstep ([],[v])
```

# Search problems

- Place $n$ queens on $n \times n$ chessboard to not attack each other

# Search problems

- Place $n$ queens on $n \times n$ chessboard to not attack each other
    - Queens attack along rows, columns and diagonals
    - Should have exactly one queen on each row and column

# Search problems

- Place $n$ queens on $n \times n$ chessboard to not attack each other
  - Queens attack along rows, columns and diagonals
  - Should have exactly one queen on each row and column
- Naive strategy
  - Place first queen on leftmost square of first row
  - In each new row, place queen at leftmost safe square

# Search problems

- Place $n$ queens on $n \times n$ chessboard to not attack each other
  - Queens attack along rows, columns and diagonals
  - Should have exactly one queen on each row and column
- Naive strategy
  - Place first queen on leftmost square of first row
  - In each new row, place queen at leftmost safe square
- After 7 moves we find no safe squares on bottom row

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Q | | | | | | | |
| | | Q | | | | | |
| | | | | Q | | | |
| | | | | | | Q | |
| | Q | | | | | | |
| | | | Q | | | | |
| | | | | | Q | | |
| | | | | | | | |

# Backtracking

# Backtracking

- Go back and try new position for 7th queen

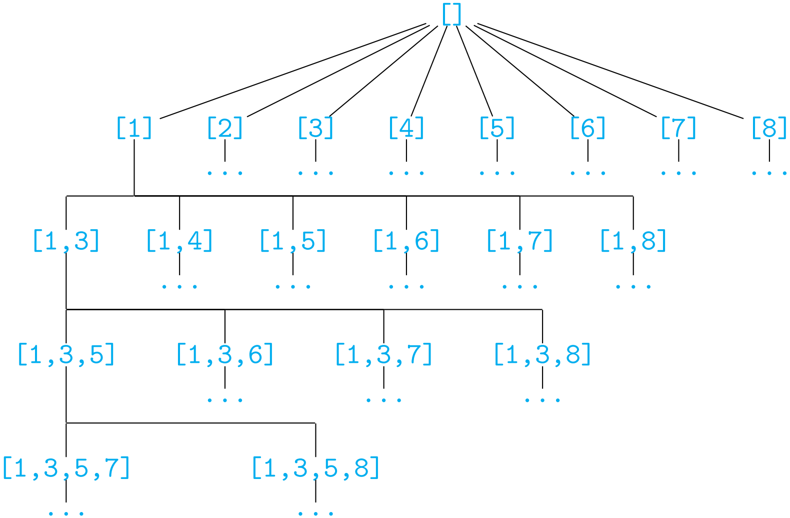
# Backtracking

- Go back and try new position for 7th queen
- After all possibilities for 7th queen exhausted, go back and try new position for 6th queen

# Backtracking

- Go back and try new position for 7th queen
- After all possibilities for 7th queen exhausted, go back and try new position for 6th queen
- Similarly go back to 5th queen, 4th queen, . . . , 1st queen

# Backtracking . . .



[]

[1] [2] [3] [4] [5] [6] [7] [8]

. . . . . . . . . . . . . . . . . . . . .

[1,3] [1,4] [1,5] [1,6] [1,7] [1,8]

. . . . . . . . . . . . . . .

[1,3,5] [1,3,6] [1,3,7] [1,3,8]

. . . . . . . . .

[1,3,5,7] [1,3,5,8]

. . . . . .

# Backtracking via Generate and Test

- In Haskell this can be implemented as follows . . .
  - Represent (partial) placement of queens as a list
  - Position `i` is column number of queen in row `i+1`
    Earlier board is represented `[1,3,5,7,2,4,6]`

# Backtracking via Generate and Test

- In Haskell this can be implemented as follows . . .
  - Represent (partial) placement of queens as a list
  - Position `i` is column number of queen in row `i+1`
    Earlier board is represented `[1,3,5,7,2,4,6]`
- `extend` computes all ways to extend a placement by one row

# Backtracking via Generate and Test

- In Haskell this can be implemented as follows . . .
  - Represent (partial) placement of queens as a list
  - Position `i` is column number of queen in row `i+1`
    Earlier board is represented `[1,3,5,7,2,4,6]`
- `extend` computes all ways to extend a placement by one row
- `extendall` maps `extend` over all placements on $k$ rows to get all placements on $k+1$ rows

# Backtracking via Generate and Test

- In Haskell this can be implemented as follows . . .
  - Represent (partial) placement of queens as a list
  - Position `i` is column number of queen in row `i+1`
    Earlier board is represented `[1,3,5,7,2,4,6]`
- `extend` computes all ways to extend a placement by one row
- `extendall` maps `extend` over all placements on $k$ rows to get all placements on $k+1$ rows
- All possible placements of $n$ queens
  - `allqueens n = head (drop n (iterate extendall [[]]))`

# Backtracking via Generate and Test

- In Haskell this can be implemented as follows ...
  - Represent (partial) placement of queens as a list
  - Position `i` is column number of queen in row `i+1`
    Earlier board is represented `[1,3,5,7,2,4,6]`
- `extend` computes all ways to extend a placement by one row
- `extendall` maps `extend` over all placements on $k$ rows to get all placements on $k+1$ rows
- All possible placements of $n$ queens
  - `allqueens n = head (drop n (iterate extendall [[]]))`
- One placement of $n$ queens
  - `queens n = head (allqueens n)`

# Queens ...

```
n :: Int
n = ..

extend l  = [(x:l) | x <- [1..n], compat x l]

compat x []  = True
compat x l  = not (elem x l) && (notdiag x l)

extendall l = concatMap extend l

allqueens = head (drop n  (iterate extendall [[]]))
queens = head allqueens
```