Functional Programming: Exploiting Laziness

Henrik Nilsson

Dynamic Programming (1)

Common problem solving strategy:

- Divide problem recursively into subproblems until easy to solve.

- Combine subproblem solutions into solutions for larger sub-problems until overall solution obtained.

Problem: one might end up solving the *same* subproblem over and over!

Dynamic Programming (2)

Dynamic programming:

- Create table of solutions to all possible subproblems.
- Look up solution to subproblems in table when needed to solve larger subproblem.

Problem: The order in which to fill in table may not be obvious!

Dynamic Programming (3)

Observations:

- The purpose of the table is to ensure that any subproblem is solved *at most once*.
- A suitable tabulating order is given by the data dependences; i.e., *demand-driven* tabulation.

But this is exactly how a lazy functional language works!

Lazy languages allow dynamic programming solutions to be formulated declaratively without worrying about evaluation order.

Digression: Arrays in Haskell

Arrays in imperative languages are filled in and manipulated incrementally.

This would be too costly in a pure (functional) language. Haskell's arrays are therefore *monolithic* – created in one go.

Array creation is strict in the index set, but elements are evaluated lazily. This allows arrays to be defined by a *recurrence relation*, i.e. in terms of themselves.

Exactly what we need for dynamic programming!

Index Types (1)

The index type of an array must belong to the class `Ix`:

```
class (Show a, Ord a) => Ix a where
  range   :: (a,a) -> [a]
  index   :: (a,a) -> a -> Int
  inRange :: (a,a) -> a -> Bool
```

`Int`, `Integer`, `Char`, any enumeration type, tuples of types in `Ix` are in `Ix`.

Index Types (2)

Examples:

```
range (0,4) ⇒ [0,1,2,3,4]

range ((0,0), (1,2)) ⇒
[(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)]

index ((0,0), (1,2)) (1,1) ⇒ 4
```

Array Creation and Indexing

```
array :: (Ix i) => (i,i) -> [(i,e)] -> Array i e
bounds :: (Ix i) => Array i e -> (i,i)
(!) :: (Ix i) => Array i e -> i -> e

squares = array (1,100) [ (i, i*i) | i <- [1..100] ]
squares1 = array (bounds squares)
                 [ (i, squares ! i + 1)
                 | i <- range (bounds squares) ]


fibs :: Int -> Array Int Int
fibs n = a where
    a = array (0,n) ([ (0,1), (1,1) ]
                     ++ [ (i, a ! (i - 2) + a ! (i - 1))
                        | i <- [2..n] ])
```

Accumulated Arrays

Accumulated arrays allow the index to appear in the list more than once.

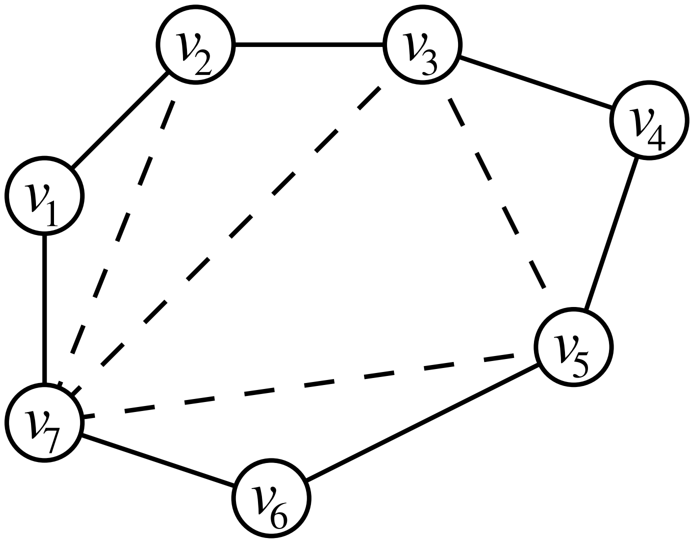
A function is used for combining values at the same index.

```
accumArray :: (Ix i) =>
    (e->a->e) -> e -> (i,i) -> [(i,a)] -> Array i e

hist :: (Ix a, Integral b) => (a,a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds
                          [ (i,1) | i <- is, inRange bnds i ]
```

The Triangulation Problem

Select a set of *chords* that divides a convex polygon into triangles so that no two chords cross each other. The sum of their lengths should be minimal.



We will only compute their minimal length.

Division into Subproblems

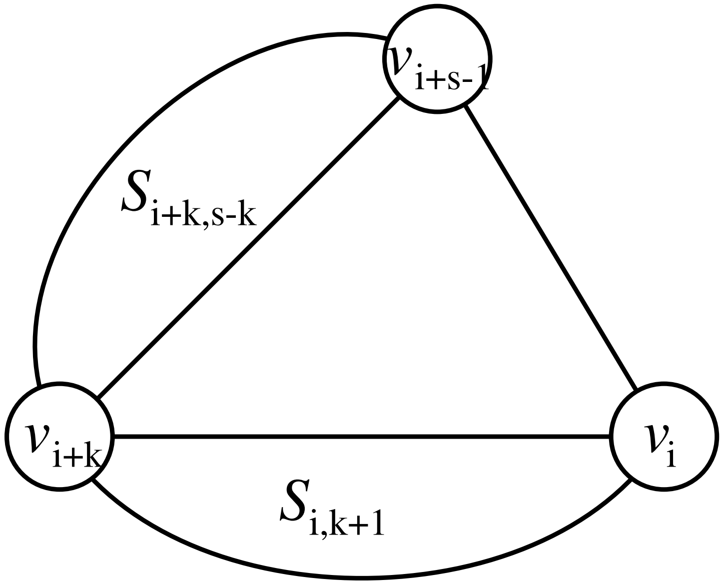
Define the subproblem of size s beginning at vertex v_i, denoted S_{is}, to be the minimum triangulation of the polygon v_i, v_{i+1}, ..., v_{i+s-1} (counting modulo the number of vertices).

Subproblem of size $s < 4$ are trivial.

Solving S_{is} is done by picking a k, $1 \leq k \leq s - 2$, and solving $S_{i,k+1}$ and $S_{i+k,s-k}$.

Division into Subproblems (2)

Subproblem S_{is}:



Division into Subproblems (3)

We obviously have to try all values for k. Writing the obvious recursive algorithm results in 3^{s-4} recursive calls.

Exponential complexity!

However! For n vertices there are only $n(n-4)$ non-trivial sub-problems (n vertices, $4 \leq s \leq n$).

Many subproblems are obviously being considered more than once by the obvious recursive formulation!

Formulation as a Recurrence

Denote the minimal triangulation cost of S_{is} by C_{is} and tabulate.

Denote the length of a *chord* between v_p and v_q by $D(v_p, v_q)$.

$D(v_P, v_q)$ is 0 for *non-chords*, i.e. adjacent v_p and v_q.

For n vertices, $1 \leq i \leq n$, $4 \leq s \leq n$ we have

$$C_{is} = \min[C_{i,k+1} + C_{i+k,s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1})]$$

where $1 \leq k \leq s - 2$.

For $s < 4$ we have $C_{is} = 0$

Haskell Solution: Auxiliary Definitions (1)

```
type Point = (Double, Double)
type Polygon = Array Int Point

polygon :: [Point] -> Polygon
polygon ps = array (1, length ps) (zip [1..] ps)

dist :: Polygon -> Int -> Int -> Double
dist p i j | adjacent p i j = 0.0
dist p i j | otherwise      = dist' (p!i) (p!j)

dist' :: Point -> Point -> Double
dist' (x,y) (x',y') = sqrt ((x - x')^2 + (y - y')^2)
```

Haskell Solution: Auxiliary Definitions (2)

```
adjacent :: Polygon -> Int -> Int -> Bool
adjacent p i j = d == 1 || d == n - 1
    where
        d = abs (i - j)
        n = snd (bounds p)
```

Haskell Solution: Rendering of the Recurrence

```
triCosts :: Polygon -> Array (Int, Int) Double
triCosts p = cost where
    cost = array ((1,2), (n,n))
                 ([ ((i, s),
                     minimum [ cost ! (i, k+1)
                              + cost ! (i +% k, s-k)
                              + dist p i (i +% k)
                              + dist p (i +% k) (i +% (s-1))
                            | k <- [1..s-2] ])
                  | i <- [1..n], s <- [4..n] ]
                  ++ [((i,s),0.0) | i <- [1..n], s <- [2..3]])
    n = snd (bounds p)
    i +% k = (i + k - 1) `mod` n + 1
```

Haskell Solution: Minimal cost

The minimal for the entire problem is just C_{1n}:

```
triCost :: Polygon -> Double
triCost p = triCosts p ! (1, snd (bounds p))

poly3 = polygon [(2.0, 0.0), (0.0, 5.0),
                 (4.0, 10.0), (6.0, 5.0)]
```

`> triCost poly3` $\Rightarrow$ 6

List of Successes

Combinatorial search:

- Look for object(s) that satisfy a given property.
- Can be solved using backtracking (cf. logical programming).

In a lazy language, we can use the technique “list of successes”:

- Enumerate all possible solutions *explicitly* in a list.
- Laziness ensures that only needed solutions are computed.
- These lists are akin to logic variables!

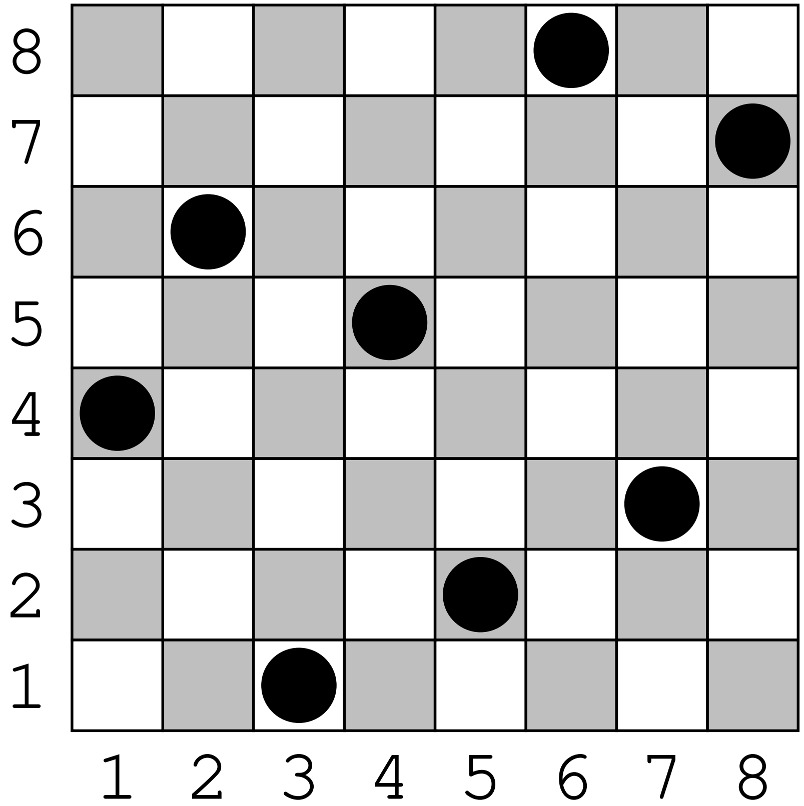
Eight Queens (1)

Given a chess board and eight queens, place the queens on the board so that no two queens threaten each other.

Observation: precisely one queen per row.

Board with queens in the first m columns represented by a list of m row numbers.

Eight Queens (2)



Representation: `[4,6,1,5,2,8,3,7]`

Eight Queens (3)

Extending a placement p by adding a queen in row n: p ++ $[n]$.

Checking whether an extension is safe:

```
safe p n = and [ not (check (i,j) (m,n))
               | (i, j) <- zip [1..lp] p
               ]
    where
        m = lp + 1
        lp = length p

check (i, j) (m, n) = j == n
                    || i + j == m + n
                    || i - j == m - n
```

Eight Queens (4)

Computing a valid placement for m columns:

```
queens 0     = [[]]
queens (m+1) = [ p ++ [n]
               | p <- queens m,
                 n <- [1..8],
                 safe p n
               ]
```

`> head (queens 8)` $\Rightarrow$ [1,5,8,6,3,7,2,4]