Generating Permutations

The algorithms so far have some drawbacks:

- greedy heuristics only heuristics
- will probably not find the optimal solution
- DP algorithms optimal, but very heavy weight
- especially memory consumption is high
- find a solution only after the complete search

Sometimes we want a more light-weight algorithm:

- low memory consumption
- stop if time runs out
- still find the optimal solution if possible

Generating Permutations (2)

We can achieve this when only considering left-deep trees:

- left-deep trees are permutations of the relations to be joined
- permutations can be generated directly
- generating all permutations is too expensive
- but some permutations can be ignored:
  Consider the join sequence $R_1 R_2 R_3 R_4$. If we know that $R_1 R_3 R_2$ is cheaper than $R_1 R_2 R_3$, we do not have to consider $R_1 R_2 R_3 R_4$.

Idea: successively add a relation. An extended sequence is only explored if exchanging the last two relations does not result in a cheaper sequence.

Recursive Search

ConstructPermutations(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal left-deep join tree
$B = \epsilon$
$P = \epsilon$
for each $R_i \in R$ {
  ConstructPermutationsRec($P \circ < R_i >$,$R \setminus \{R_i\}$,B)
} **return** B

- algorithm considers a prefix P and the rest R
- keeps track of the best tree found so far B
- increases the prefix recursively

Recursive Search (2)

ConstructPermutationsRec(P, R, B)
Input: a prefix P, remaining relations R, best plan B
Output: side effects on B

if $|R| = 0$ {
  if $B = \epsilon \vee C(B) > C(P)$ {
    $B = P$
  }
} **else** {
  for each $R_i \in R$ {
    if $C(P\circ < R_i >) \leq C(P[1 : |P| - 1]\circ < R_i, P[|P|] >)$ {
      ConstructPermutationsRec($P\circ < R_i >, R \setminus \{R_i\}, B$)
    }
  }
}

Remarks

Good:

- linear memory
- immediately produces plan alternatives
- anytime algorithm
- finds the optimal plan eventually

Bad:

- worst-case runtime if ties occur
- worst-case runtime if no ties occur is an open problem

Often fast, can be stopped anytime, but may perform poorly.

Transformative Approaches

Main idea: [6]

- use equivalences directly (associativity, commutativity)
- would make integrating new equivalences easy

Problems:

- how to navigate the search space
- equivalences have no order
- how to guarantee finding the optimal solution
- how to avoid exhaustive search

Rule Set

$R_1 \bowtie R_2$	$\rightsquigarrow$	$R_2 \bowtie R_1$	Commutativity
$(R_1 \bowtie R_2) \bowtie R_3$	$\rightsquigarrow$	$R_1 \bowtie (R_2 \bowtie R_3)$	Right Associativity
$R_1 \bowtie (R_2 \bowtie R_3)$	$\rightsquigarrow$	$(R_1 \bowtie R_2) \bowtie R_3$	Left Associativity
$(R_1 \bowtie R_2) \bowtie R_3$	$\rightsquigarrow$	$(R_1 \bowtie R_3) \bowtie R_2$	Left Join Exchange
$R_1 \bowtie (R_2 \bowtie R_3)$	$\rightsquigarrow$	$R_2 \bowtie (R_1 \bowtie R_3)$	Right Join Exchange

Two more rules are often used to transform left-deep trees:

- *swap* exchanges two arbitrary relations in a left-deep tree
- *3Cycle* performs a cyclic rotation of three arbitrary relations in a left-deep tree.

To try another join method, another rule called *join method exchange* is introduced.

Rule Set RS-0

- commutativity
- left-associativity
- right-associativity

Basic Algorithm

ExhaustiveTransformation($\{R_1, \ldots, R_n\}$)
Input: a set of relations
Output: an optimal join tree
Let T be an arbitrary join tree for all relations
Done $= \emptyset$ // contains all trees processed
ToDo $= \{T\}$ // contains all trees to be processed
while $|\text{ToDo}| > 0$ {
    T = an arbitrary tree in ToDo
    ToDo = ToDo $\setminus T$;
    Done = Done $\cup \{T\}$;
    Trees = ApplyTransformations(T);
    for each $T \in$ Trees {
        if $T \notin$ ToDo $\cup$ Done
            ToDo = ToDo $\cup \{T\}$
    }
}
return $\arg\min_{T \in \text{Done}} C(T)$

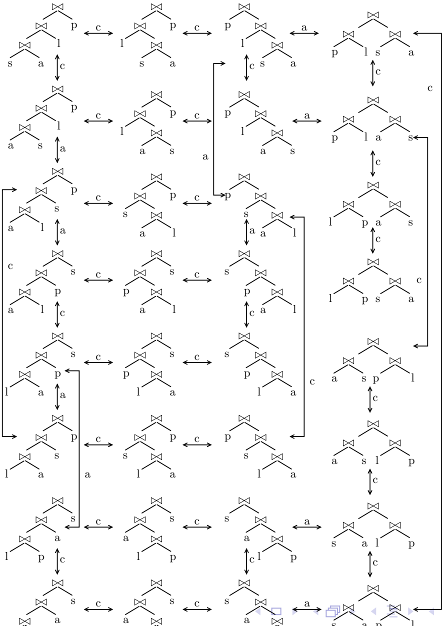
Basic Algorithm (2)

ApplyTransformations(T)
Input: join tree
Output: all trees derivable by associativity and commutativity
Trees $= \emptyset$
Subtrees $=$ all subtrees of T rooted at inner nodes
for each $S \in$ Subtrees {
    if S is of the form $S_1 \bowtie S_2$
        Trees $=$ Trees $\cup \{S_2 \bowtie S_1\}$
    if S is of the form $(S_1 \bowtie S_2) \bowtie S_3$
        Trees $=$ Trees $\cup \{S_1 \bowtie (S_2 \bowtie S_3)\}$
    if S is of the form $S_1 \bowtie (S_2 \bowtie S_3)$
        Trees $=$ Trees $\cup \{(S_1 \bowtie S_2) \bowtie S_3\}$
}
return Trees;

Remarks

- if no cross products are to be considered, extend **if** conditions for associativity rules.
- problem 1: explores the whole search space
- problem 2: generates join trees more than once
- problem 3: sharing of subtrees is non-trivial

Search Space



Introducing the Memo Structure

A memoization strategy is used to keep the runtime reasonable:

- for any subset of relations, dynamic programming remembers the best join tree.
- this does not quite suffice for the transformation-based approach.
- instead, we have to keep all join trees generated so far including those differing in the order of the arguments of a join operator.
- however, subtrees can be shared.
- this is done by keeping pointers into the data structure (see next slide).

Memo Structure Example

$\{R_1, R_2, R_3\}$	$\{R_1, R_2\} \bowtie R_3, R_3 \bowtie \{R_1, R_2\},$ $\{R_1, R_3\} \bowtie R_2, R_2 \bowtie \{R_1, R_3\},$ $\{R_2, R_3\} \bowtie R_1, R_1 \bowtie \{R_2, R_3\}$
$\{R_2, R_3\}$	$\{R_2\} \bowtie \{R_3\}, \{R_3\} \bowtie \{R_2\}$
$\{R_1, R_3\}$	$\{R_1\} \bowtie \{R_3\}, \{R_3\} \bowtie \{R_1\}$
$\{R_1, R_2\}$	$\{R_1\} \bowtie \{R_2\}, \{R_2\} \bowtie \{R_1\}$
$\{R_3\}$	R_3
$\{R_2\}$	R_2
$\{R_1\}$	R_1

- in Memo Structure: arguments are pointers to classes
- Algorithm: `ExploreClass` expands a class
- Algorithm: `ApplyTransformation2` expands a member of a class

Memoizing Algorithm

ExhaustiveTransformation2(Query Graph G)
Input: a query specification for relations $\{R_1, \ldots, R_n\}$.
Output: an optimal join tree
initialize MEMO structure
ExploreClass($\{R_1, \ldots, R_n\}$)
return $\arg\min_{T \in \text{class } \{R_1, \ldots, R_n\}} C(T)$

- stored an arbitrary join tree in the memo structure
- explores alternatives recursively

Memoizing Algorithm (2)

ExploreClass(C)
Input: a class $\mathcal{C} \subseteq \{R_1, \ldots, R_n\}$
Output: none, but has side-effect on MEMO-structure
while not all join trees in C have been explored {
    choose an unexplored join tree T in $\mathcal{C}$
    ApplyTransformation2(T)
    mark T as explored
}

- considers all alternatives within one class
- transformations themselves are done in ApplyTransformation2

Memoizing Algorithm (3)

ApplyTransformations2(T)
Input: a join tree of a class $\mathcal{C}$
Output: none, but has side-effect on MEMO-structure
ExploreClass(left-child(T))
ExploreClass(right-child(T));
for each transformation $\mathcal{T}$ and class member of child classes {
    for each T' resulting from applying $\mathcal{T}$ to T {
        if T' not in MEMO structure {
            add T' to class $\mathcal{C}$ of MEMO structure
        }
    }
}

- first explores subtrees
- then applies all known transformations to the tree
- stores new trees in the memo structure

Remarks

- Applying `ExhaustiveTransformation2` with a rule set consisting of Commutativity and Left and Right Associativity generates $4^n - 3^{n+1} + 2^{n+2} - n - 2$ duplicates
- Contrast this with the number of join trees contained in a completely filled MEMO structure: $3^n - 2^{n+1} + n + 1$
- Solve the problem of duplicate generation by disabling applied rules.

Rule Set RS-1

T_1: Commutativity $C_1 \bowtie_0 C_2 \rightsquigarrow C_2 \bowtie_1 C_1$
Disable all transformations T_1, T_2, and T_3 for $\bowtie_1$.

T_2: Right Associativity $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow C_1 \bowtie_2 (C_2 \bowtie_3 C_3)$
Disable transformations T_2 and T_3 for $\bowtie_2$ and enable all rules for $\bowtie_3$.

T_3: Left associativity $C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \rightsquigarrow (C_1 \bowtie_2 C_2) \bowtie_3 C_3$
Disable transformations T_2 and T_3 for $\bowtie_3$ and enable all rules for $\bowtie_2$.

Example for chain $R_1 - R_2 - R_3 - R_4$

Class	Initialization	Transformation	Step
$\{R_1, R_2, R_3, R_4\}$	$\{R_1, R_2\} \bowtie_{111} \{R_3, R_4\}$	$\{R_3, R_4\} \bowtie_{000} \{R_1, R_2\}$	3
		$R_1 \bowtie_{100} \{R_2, R_3, R_4\}$	4
		$\{R_1, R_2, R_3\} \bowtie_{100} R_4$	5
		$\{R_2, R_3, R_4\} \bowtie_{000} R_1$	8
		$R_4 \bowtie_{000} \{R_1, R_2, R_3\}$	10
$\{R_2, R_3, R_4\}$		$R_2 \bowtie_{111} \{R_3, R_4\}$	4
		$\{R_3, R_4\} \bowtie_{000} R_2$	6
		$\{R_2, R_3\} \bowtie_{100} R_4$	6
		$R_4 \bowtie_{000} \{R_2, R_3\}$	7
$\{R_1, R_3, R_4\}$			
$\{R_1, R_2, R_4\}$			
$\{R_1, R_2, R_3\}$		$\{R_1, R_2\} \bowtie_{111} R_3$	5
		$R_3 \bowtie_{000} \{R_1, R_2\}$	9
		$R_1 \bowtie_{100} \{R_2, R_3\}$	9
		$\{R_2, R_3\} \bowtie_{000} R_1$	9
$\{R_3, R_4\}$	$R_3 \bowtie_{111} R_4$	$R_4 \bowtie_{000} R_3$	2
$\{R_2, R_4\}$			
$\{R_2, R_3\}$			
$\{R_1, R_4\}$			
$\{R_1, R_3\}$			
$\{R_1, R_2\}$	$R_1 \bowtie_{111} R_2$	$R_2 \bowtie_{000} R_1$	1

Rule Set RS-2

Bushy Trees: Rule set for clique queries and if cross products are allowed:

T_1: Commutativity $C_1 \bowtie_0 C_2 \leadsto C_2 \bowtie_1 C_1$
Disable all transformations T_1, T_2, T_3, and T_4 for $\bowtie_1$.

T_2: Right Associativity $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \leadsto C_1 \bowtie_2 (C_2 \bowtie_3 C_3)$
Disable transformations T_2, T_3, and T_4 for $\bowtie_2$.

T_3: Left Associativity $C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \leadsto (C_1 \bowtie_2 C_2) \bowtie_3 C_3$
Disable transformations T_2, T_3 and T_4 for $\bowtie_3$.

T_4: Exchange $(C_1 \bowtie_0 C_2) \bowtie_1 (C_3 \bowtie_2 C_4) \leadsto (C_1 \bowtie_3 C_3) \bowtie_4 (C_2 \bowtie_5 C_4)$
Disable all transformations T_1, T_2, T_3, and T_4 for $\bowtie_4$.

If we initialize the MEMO structure with left-deep trees, we can strip down the above rule set to Commutativity and Left Associativity. Reason: from a left-deep join tree we can generate all bushy trees with only these two rules

Rule Set RS-3

Left-deep trees:

T_1 Commutativity $R_1 \bowtie_0 R_2 \leadsto R_2 \bowtie_1 R_1$
Here, the R_i are restricted to classes with exactly one relation. T_1 is disabled for $\bowtie_1$.

T_2 Right Join Exchange $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \leadsto (C_1 \bowtie_2 C_3) \bowtie_3 C_2$
Disable T_2 for $\bowtie_3$.

Generating Random Join Trees

Generating a random join tree is quite useful:

- allows for cost sampling
- randomized optimization procedures
- basis for Simulated Annealing, Iterative Improvement etc.
- easy with cross products, difficult without
- we consider with cross products first

Main problems:

- generating all join trees (potentially)
- creating all with the same probability

Ranking/Unranking

Let S be a set with n elements.

- a bijective mapping $f : S \rightarrow [0, n[$ is called *ranking*
- a bijective mapping $f : [0, n[\rightarrow S$ is called *unranking*

Given an unranking function, we can generate random elements in S by generating a random number in $[0, n[$ and unranking this number.
Challenge: making unranking fast.

Random Permutations

Every permutation corresponds to a left-deep join tree possibly with cross products.
Standard algorithm to generate random permutations is the starting point for the algorithm:

for each $k \in [0, n[$ **descending**
  $\text{swap}(\pi[k], \pi[\text{random}(k)])$

Array π initialized with elements $[0, n[$.
`random`(k) generates a random number in $[0, k]$.

Random Permutations

- Assume the random elements produced by the algorithm are $r_{n-1}, \ldots, r_0$ where $0 \leq r_i \leq i$.
- Thus, there are exactly $n(n-1)(n-2)\ldots 1 = n!$ such sequences and there is a one to one correspondance between these sequences and the set of all permutations.
- Unrank $r \in [0, n![$ by turning it into a unique sequence of values $r_{n-1}, \ldots, r_0$.
  Note that after executing the swap with r_{n-1} every value in $[0, n[$ is possible at position $\pi[n-1]$.
  Further, $\pi[n-1]$ is never touched again.
- Hence, we can unrank r as follows. We first set $r_{n-1} = r \mod n$ and perform the swap. Then, we define $r' = \lfloor r/n \rfloor$ and iteratively unrank r' to construct a permutation of $n-1$ elements.

Generating Random Permutations

Unrank(n, r)
Input: the number n of elements to be permuted
and the rank r of the permutation to be constructed
Output: a permutation π
for each $0 \leq i < n$
  $\pi[i] = i$
for each $n \geq i > 0$ **descending** {
  swap($\pi[i-1]$, $\pi[r \mod i]$)
  $r = \lfloor r/i \rfloor$
}
return π;

Generating Random Bushy Trees with Cross Products

Steps of the algorithm:

1. Generate a random number b in $[0, C(n)[$.
2. Unrank b to obtain a bushy tree with $n-1$ inner nodes.
3. Generate a random number p in $[0, n![$.
4. Unrank p to obtain a permutation.
5. Attach the relations in order p from left to right as leaf nodes to the binary tree obtained in Step 2.

The only step that we have still to discuss is Step 2.

Tree Encoding

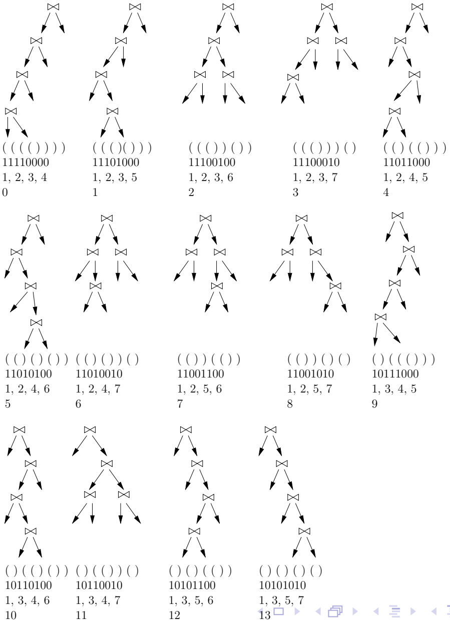
- Preordertraversal:
  - Inner node: '('
  - Leaf Node: ')'

  Skip last leaf node.
- Replace '(' by 1 and ')' by 0
- Just take positions of 1s.

Example: all trees with four inner nodes:

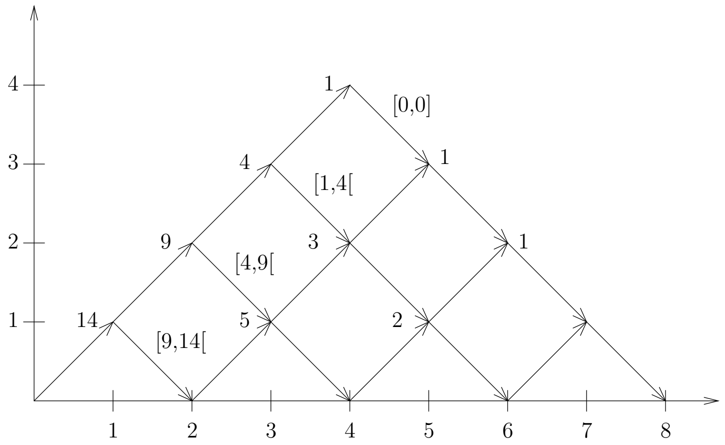
- The ranks are in $[0, 14[$

Tree Ranking Example



Unranking Binary Trees

We establish a bijection between Dyck words and paths in a grid:



Every path from $(0, 0)$ to $(2n, 0)$ uniquely corresponds to a Dyck word.

Counting Paths

The number of different paths from $(0,0)$ to (i,j) can be computed by

$$p(i,j) = \frac{j+1}{i+1}\binom{i+1}{\frac{1}{2}(i+j)+1}$$

These numbers are the *Ballot numbers*.
The number of paths from (i,j) to $(2n,0)$ can thus be computed as:

$$q(i,j) = p(2n-i,j)$$

Note the special case $q(0,0) = p(2n,0) = C(n)$.

Unranking Outline

- We open a parenthesis (go from (i, j) to $(i + 1, j + 1)$) as long as the number of paths from that point does no longer exceed our rank r.
- If it does, we close a parenthesis (go from (i, j) to $(i + 1, j - 1)$).
- Assume, that we went upwards to (i, j) and then had to go down to $(i + 1, j - 1)$.
  We subtract the number of paths from $(i + 1, j + 1)$ from our rank r and proceed iteratively from $(i + 1, j - 1)$ by going up as long as possible and going down again.
- Remembering the number of parenthesis opened and closed along our way results in the required encoding.

Generating Bushy Trees

UnrankTree(n, r)
Input: a number of inner nodes n and a rank $r \in [0, C(n)[$
Output: encoding of the inner leafes of a tree
open $= 1$, close $= 0$
pos $= 1$, encoding $= < 1 >$
while $|$encoding$| < n$ {
  $k = q$(open+close,open-close)
  if $k \leq r$ {
    $r = r - k$, close=close+1
  } **else** {
    encoding=encoding$\circ <$ pos $>$, open=open+1
  }
  pos=pos+1
}
return encoding