# Genetic Algorithms

- Join trees seen as population
- Successor generations generated by crossover and mutation
- Only the fittest survive

Problem: Encoding

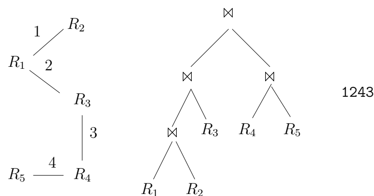
- Chromosome $\longleftrightarrow$ string
- Gene $\longleftrightarrow$ character

# Encoding

We distinguish *ordered list* and *ordinal number* encodings.
Both encodings are used for left-deep and bushy trees.
In all cases we assume that the relations $R_1, \ldots, R_n$ are to be joined and use the index $i$ to denote $R_i$.

# Ordered List Encoding

1. left-deep trees
   A left-deep join tree is encoded by a permutation of $1, \ldots, n$. For instance, $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$ is encoded as "1423".
2. bushy trees
   A bushy join-tree without cartesian products is encoded as an ordered list of the edges in the join graph. Therefore, we number the edges in the join graph. Then, the join tree is encoded in a bottom-up, left-to-right manner.



# Ordinal Number Encoding

In both cases, we start with the list $L =< R_1, \ldots, R_n >$.

- left-deep trees
  Within $L$ we find the index of first relation to be joined. If this relation be $R_i$ then the first character in the chromosome string is $i$. We eliminate $R_i$ from $L$. For every subsequent relation joined, we again determine its index in $L$, remove it from $L$ and append the index to the chromosome string.
  For instance, starting with $< R_1, R_2, R_3, R_4 >$, the left-deep join tree $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$ is encoded as "1311".

# Ordinal Number Encoding (2)

- bushy trees
  We encode a bushy join tree in a bottom-up, left-to-right manner. Let $R_i \bowtie R_j$ be the first join in the join tree under this ordering. Then we look up their positions in $L$ and add them to the encoding. Then we eliminate $R_i$ and $R_j$ from $L$ and push $R_{i,j}$ to the front of it. We then proceed for the other joins by again selecting the next join which now can be between relations and or subtrees. We determine their position within $L$, add these positions to the encoding, remove them from $L$, and insert a composite relation into $L$ such that the new composite relation directly follows those already present.
  For instance, starting with the list $< R_1, R_2, R_3, R_4 >$, the bushy join tree $((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4))$ is encoded as “12 23 12”.

# Crossover

1. Subsequence exchange
2. Subset exchange

# Crossover: Subsequence exchange

The subsequence exchange for the ordered list encoding:

- Assume two individuals with chromosomes $u_1 v_1 w_1$ and $u_2 v_2 w_2$.
- From these we generate $u_1 v'_1 w_1$ and $u_2 v'_2 w_2$ where $v'_i$ is a permutation of the relations in $v_i$ such that the order of their appearence is the same as in $u_{3-i} v_{3-i} w_{3-i}$.

Subsequence exchange for ordinal number encoding:

- We require that the $v_i$ are of equal length ($|v_1| = |v_2|$) and occur at the same offset ($|u_1| = |u_2|$).
- We then simply swap the $v_i$.
- That is, we generate $u_1 v_2 w_1$ and $u_2 v_1 w_2$.

# Crossover: Subset exchange

The subset exchange is defined only for the ordered list encoding. Within the two chromosomes, we find two subsequences of equal length comprising the same set of relations. These sequences are then simply exchanged.

# Mutation

A mutation randomly alters a character in the encoding.
If duplicates may not occur— as in the ordered list encoding—swapping two characters is a perfect mutation.

# Selection

- The probability of survival is determined by its rank in the population.
- We calculate the costs of the join trees encoded for each member in the population.
- Then, we sort the population according to their associated costs and assign probabilities to each individual such that the best solution in the population has the highest probability to survive and so on.
- After probabilities have been assigned, we randomly select members of the population taking into account these probabilities.
- That is, the higher the probability of a member the higher its chance to survive.

# The Algorithm

1. Create a random population of a given size (say 128).
2. Apply crossover and mutation with a given rate.
   For example such that 65% of all members of a population participate in crossover, and 5% of all members of a population are subject to random mutation.
3. Apply selection until we again have a population of the given size.
4. Stop after no improvement within the population was seen for a fixed number of iterations (say 30).

# Combinations

- metaheuristics are often not used in isolation
- they can be used to improve existing heurstics
- or heuristics can be used to speed up metaheuristics

# Two Phase Optimization

1. For a number of randomly generated initial trees, Iterative Improvement is used to find a local minima.
2. Then Simulated Annealing is started to find a better plan in the neighborhood of the local minima.
   The initial temperature of Simulated Annealing can be lower as is its original variants.

# AB Algorithm

1. If the query graph is cyclic, a spanning tree is selected.
2. Assign join methods randomly
3. Apply IKKBZ
4. Apply iterative improvement

# Toured Simulated Annealing

The basic idea is that simulated annealing is called $n$ times with different initial join trees, if $n$ is the number of relations to be joined.

- Each join sequence in the set $S$ produced by `GreedyJoinOrdering-3` is used to start an independent run of simulated annealing.

As a result, the starting temperature can be descreased to 0.1 times the cost of the initial plan.

# GOO-II

Append an iterative improvement step to GOO

# Iterative Dynamic Programming

- Two variants: IDP-1, IDP-2 [8]
- Here: Only IDP-1 base version

Idea:

- create join trees with up to $k$ relations
- replace cheapest one by a compound relation
- start all over again

# Iterative Dynamic Programming (2)

IDP-1($\{R_1, \ldots, R_n\}$, $k$)
**Input:** a set of relations to be joined, maximum block size $k$
**Output:** a join tree
**for** $\forall 1 \leq i \leq n$ {
  BestTree($\{R_i\}$) $= R_i$;
}
ToDo $= \{R_1, \ldots, R_n\}$

# Iterative Dynamic Programming (3)

**while** $|\text{ToDo}| > 1$ {
  $k = \min(k, |ToDo|)$
  **for** $\forall 2 \leq i < k$ **ascending**
    **for all** $S \subseteq ToDo, |S| = i$ **do**
      **for all** $O \subset S$ **do**
        $\text{BestTree}(S) = \text{CreateJoinTree}(\text{BestTree}(S), \text{BestTree}(O))$;
  **find** $V \subset ToDo, |V| = k$ **with**
    $\text{cost}(\text{BestTree}(V)) = \min\{\text{cost}(\text{BestTree}(W)) \mid W \subset ToDo, |W| = k\}$
  **generate** new symbol $T$
  $\text{BestTree}(\{T\}) = \text{BestTree}(V)$
  $\text{ToDo} = (\text{ToDo} \setminus V) \cup \{T\}$
  **for** $\forall O \subset V$ **do** $\text{delete}(\text{BestTree}(O))$
}
**return** $\text{BestTree}(\{R_1, \ldots, R_n\})$

# Iterative Dynamic Programming (4)

- compromise between runtime and optimality
- combines greedy heuristics with dynamic programming
- scales well to large problems
- finds the optimal solution for smaller problems
- approach can be used for different DP strategies

# Order Preserving Joins

- some query languages operatore on lists instead of sets/bags
- order of tuples matters
- examples: XPath/XQuery
- alternatives: either add sort operators or use order preserving operators

Here, we define order preserving operators, $list \rightarrow list$

- let $L$ be a list
- $L[1]$ is the first entry in $L$
- $L[2 : |L|]$ are the remaining entries

# Order Preserving Selection

We define the order preserving selection $\sigma^L$ as follows:

$$\sigma^L_p(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ < e[1] > \circ \sigma^L_p(e[2:|e|]) & \text{if } p(e[1]) \\ \sigma^L_p(e[2:|e|]) & \text{otherwise} \end{cases}$$

- filters like a normal selection
- preserves the relative ordering (guaranteed)

# Order Preserving Cross Product

We define the order preserving cross product $\times^L$ as follows:

$$e_1 \times^L e_2 := \begin{cases} \epsilon & \text{if } e_1 = \epsilon \\ (e[1] \hat{\times}^L e_2) \circ (e_1[2 : |e_1] \times^L e_2) & \text{otherwise} \end{cases}$$

using the tuple/list product defined as:

$$t \hat{\times}^L e := \begin{cases} \epsilon & \text{if } e = \epsilon \\ < t \circ e[1] > \circ (t \hat{\times}^L e[2 : |e|]) & \text{otherwise} \end{cases}$$

- preserves the order of $e_1$
- order of $e_2$ is preserved for each $e_1$ group

# Order Preserving Join

The definition of the order preserving join is analogous to the non-order preserving case:

$$e_1 \Join_p^L e_2 := \sigma_p^L(e_1 \times^L e_2)$$

- preserves order of $e_1$, order of $e_2$ relative to $e_1$

# Equivalences

$$
\begin{array}{rcll}
\sigma^L_{p_1}(\sigma^L_{p_2}(e)) & \equiv & \sigma^L_{p_2}(\sigma^L_{p_1}(e)) & \\
\sigma^L_{p_1}(e_1 \bowtie^L_{p_2} e_2) & \equiv & \sigma^L_{p_1}(e_1) \bowtie^L_{p_2} e_2) & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_1) \\
\sigma^L_{p_2}(e_1 \bowtie^L_{p_2} e_2) & \equiv & e_1 \bowtie^L_{p_2} \sigma^L_{p_1}(e_2) & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_2) \\
e_1 \bowtie^L_{p_1} (e_2 \bowtie^L_{p_2} e_3) & \equiv & (e_1 \bowtie^L_{p_1} e_2) \bowtie^L_{p_2} e_3) & \text{if } \mathcal{F}(p_i) \subseteq \mathcal{A}(e_i) \cup \mathcal{A}(e_{i+1})
\end{array}
$$

- swap selections
- push selections down
- associativity

# Commutativity

Consider the relations $R_1 =< [a:1],[a:2] >$ and $R_2 =< [b:1],[b:2] >$. Then

$$\begin{array}{rcl} R_1 \bowtie^L_{true} R_2 & = & < [a:1,b:1],[a:1,b:2],[a:2,b:1],[a:2,b:2] > \\ R_2 \bowtie^L_{true} R_1 & = & < [a:1,b:1],[a:2,b:1],[a:1,b:2],[a:2,b:2] > \end{array}$$

- the order preserving join is not commutative

# Algorithm

- similar to matrix multiplication
- in addition: selection push down
- DP table is a $n \times n$ array (or rather 4 arrays)
- algorithm fills arrays $p, s, c, t$:
  - $p$: applicable predicates
  - $s$: statistics (cardinality, perhaps more)
  - $c$: costs
  - $t$: split position for larger plans
- plan is extracted from the arrays afterwards

# Algorithm (2)

OrderPreservingJoins($R = \{R_1, \ldots, R_n\}, P$)
**Input:** a set of relations to be joined and a set of predicates
**Output:** fills $p, s, c, t$
**for** $\forall 1 \leq i \leq n$ {
  $p[i,i]$ =predicates from $P$ applicable to $R_i$
  $P = P \setminus p[i,i]$
  $s[i,i]$ =statistics for $\sigma_{p[i,i]}(R_i)$
  $c[i,i]$ =costs for $\sigma_{p[i,i]}(R_i)$
}

# Algorithm (3)

**for** $\forall 2 \leq l \leq n$ **ascending** $\{$
  **for** $\forall 1 \leq i \leq n - l + 1$ $\{$
    $j = i + l - 1$
    $p[i,j]$=predicates from $P$ applicable to $R_i, \ldots, R_j$
    $P = P \setminus p[i,j]$
    $s[i,j]$=statistics derived from $s[i,j-1]$ and $s[j,j]$ including $p[i,j]$
    $c[i,j]$=$\infty$
    **for** $\forall i \leq k < j$ $\{$
      $q = c[i,k] + c[k+1,j]$+costs for $s[i,k]$ and $s[k+1,j]$ and $p[i,j]$
      **if** $q < c[i,j]$ $\{$
        c[i,j]=q
        t[i,j]=k
      $\}$
    $\}$
  $\}$
$\}$

# Algorithm (4)

ExtractPlan($R = \{R_1, \ldots, R_n\}$,$t$,$p$)
**Input:** a set of relations, arrays $t$ and $p$
**Output:** a bushy join tree
**return** ExtractPlanRec($R$,$t$,$p$,1,$n$)

ExtractPlanRec($R = \{R_1, \ldots, R_n\}$,$t$,$p$,$i$,$j$)
**if** $i < j$ {
  $T_1$ =ExtractPlanRec($R$,$t$,$p$,$i$,$t[i,j]$)
  $T_2$ =ExtractPlanRec($R$,$t$,$p$,$t[i,j]+1$,$j$)
  **return** $T_1 \bowtie^L_{p[i,j]} T_2$
} **else** {
  **return** $\sigma_{p[i,j]} R_i$
}