Metaheuristics

- provide a very general optimization strategy
- applicable for many different problems
- work well even for very large problems
- but are often considered a "brute-force" method

We consider the metaheuristics formulated for the join ordering problem.

Iterative Improvement

- Start with random join tree
- Select rule that improves join tree
- Stop when no further improvement possible

Iterative Improvement (2)

IterativeImprovementBase(Query Graph G)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a join tree
do {
  JoinTree = random tree
  JoinTree = IterativeImprovement(JoinTree)
  if cost(JoinTree) < cost(BestTree) {
    BestTree = JoinTree
  }
} **while** (time limit not exceeded)
return BestTree

Iterative Improvement (3)

```
IterativeImprovement(JoinTree)
Input:  a join tree
Output: improved join tree
do {
  JoinTree' = randomly apply a transformation from the rule set to the JoinT
  if (cost(JoinTree') < cost(JoinTree)) {
    JoinTree = JoinTree'
  }
} while local minimum not reached
return JoinTree
```

- problem: local minimum detection

Simulated Annealing

- II: stuck in local minimum
- SA: allow moves that result in more expensive join trees
- lower the threshold for worsening

Simulated Annealing (2)

SimulatedAnnealing(Query Graph G)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a join tree
BestTreeSoFar = random tree
Tree = BestTreeSoFar

Simulated Annealing (3)

```
do {
  do {
    Tree' = apply random transformation to Tree
    if (cost(Tree') < cost(Tree)) {
      Tree = Tree'
    } else {
      with probability e^{-(cost(Tree')-cost(Tree))/temperature}
        Tree = Tree'
    }
    if (cost(Tree) < cost(BestTreeSoFar)) {
      BestTreeSoFar = Tree'
    }
  } while equilibrium not reached
  reduce temperature
} while not frozen
return BestTreeSoFar
```

Simulated Annealing (4)

Advantages:

- can escape from local minimum
- produces better results than II

Problems:

- parameter tuning
- initial temperature
- when and how to decrease the temperature

Tabu Search

- Select cheapest reachable neighbor (even if it is more expensive)
- Maintain tabu set to avoid running into circles

Tabu Search (2)

TabuSearch(Query Graph)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a join tree
Tree = random join tree
BestTreeSoFar = Tree
TabuSet = $\emptyset$
do {
  Neighbors = all trees generated by applying a transformation to Tree
  Tree = cheapest in Neighbors $\setminus$ TabuSet
  if cost(Tree) $<$ cost(BestTreeSoFar)
    BestTreeSoFar = Tree
  if ($|$TabuSet$| >$ limit) remove oldest tree from TabuSet
  TabuSet = TabuSet$\cup\{$Tree$\}$
}
return BestTreeSoFar

Genetic Algorithms

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

Problem: Encoding

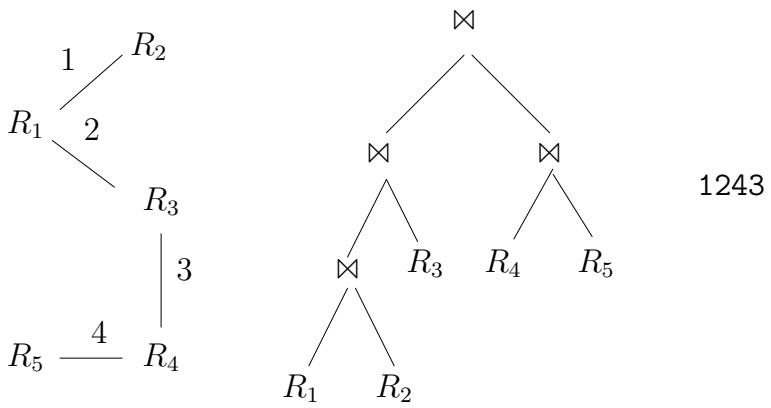
- Chromosome $\longleftrightarrow$ string
- Gene $\longleftrightarrow$ character

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 each $1 \leq i \leq n$ {
  BestTree($\{R_i\}$) $= R_i$;
}
ToDo $= \{R_1, \ldots, R_n\}$

Iterative Dynamic Programming (3)

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