

Quick Pick

- problem: build (pseudo-)random join trees fast
- unranking without cross products is quite involved
- idea: randomly select an edge in the query graph
- extend join tree by selected edge

No longer uniformly distributed, but very fast

Quick Pick (2)

QuickPick(Query Graph G)

Input: a query graph $G = (\{R_1, \dots, R_n\}, E)$

Output: a bushy join tree

$E' = E;$

Trees = $\{R_1, \dots, R_n\};$

while |Trees| > 1 {

 choose a random $e \in E'$

$E' = E' \setminus \{e\}$

if e connects two relations in different subtrees $T_1, T_2 \in \text{Trees}$

 Trees = Trees $\setminus \{T_1, T_2\} \cup \text{CreateJoinTree}(T_1, T_2)$

}

return $T \in \text{Trees}$

- repeated multiple times to find a good tree

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 $G = (\{R_1, \dots, R_n\}, E)$

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 JoinTree  
    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 $G = (\{R_1, \dots, R_n\}, E)$

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  $e^{-(\text{cost}(\text{Tree}') - \text{cost}(\text{Tree})) / \text{temperature}}$   
        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 $G = (\{R_1, \dots, R_n\}, E)$

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 \ TabuSet  
    if cost(Tree) < cost(BestTreeSoFar)  
        BestTreeSoFar = Tree  
    if (|TabuSet| > limit) remove oldest tree from TabuSet  
    TabuSet = TabuSet  $\cup$  {Tree}  
}  
return BestTreeSoFar
```