DM811 – Fall 2009

Heuristics for Combinatorial Optimization

Lecture 5

Construction Heuristics and Metaheuristics

Marco Chiarandini

Deptartment of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Complete Search Methods
   - General Search Methods
   - A* search
2. Incomplete Search Methods
3. Metaheuristics
   - Rollout/Pilot Method
   - Beam Search
   - Iterated Greedy
   - GRASP
   - Adaptive Iterated Construction Search
   - Multilevel Refinement
4. Heuristics for TSP

Search Methods

- initial state: the empty assignment $\{\}$ in which all variables are unassigned
- successor function: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables
- goal test: the current assignment is complete
- path cost: a constant cost

Types of problems:

- Assignment
- Sequencing
- Subset
- Routing
- ...

Tree Search

Search Space

tree with branching factor at the top level nd
at the next level $(n-1)d$.
The tree has $n! \cdot d^n$ even if only d^n possible complete assignments.

- CSP is commutative in the order of application of any given set of action. (the order of the assignment does not influence)
- Hence we can consider search algs that generate successors by considering possible assignments for only a single variable at each node in the search tree.

Backtrack Search

Backtracking search

depth first search that chooses one variable at a time and backtracks when a variable has no legal values left to assign.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment according to CONSTRAINTS[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```


Backtrack Search

- No need to copy solutions all the times but rather extensions and undo extensions
- Since CSP is standard then the alg is also standard and can use general purpose algorithms for initial state, successor function and goal test.
- Backtracking is uninformed and complete. Other search algorithms may use information in form of heuristics.

Complete Tree Search

Uninformed

- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional Search

Informed

- Informed search algorithm: exploit problem-specific knowledge
- Best-first search: node that "appears" to be the best selected for expansion based on an evaluation function $f(x)$
- Implemented through a priority queue of nodes in ascending order of f (See later discussion on A* search)

General Purpose Methods

Decisions in general purpose methods:

1) Which variable should we assign next, and in what order should its values be tried?
2) What are the implications of the current variable assignments for the other unassigned variables?
3) When a path fails – that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?

Search (1) + Inference (2) + Backtracking (3) = Constraint Programming

In the general case, at point 1) we use heuristic rules.

If we do not backtrack (point 3) then we have a construction heuristic.

1) Which variable should we assign next, and in what order should its values be tried?

- Select-Initial-Unassigned-Variable
- Select-Unassigned-Variable
  - most constrained first = fail-first heuristic
    = Minimum remaining values (MRV) heuristic
    (tend to reduce the branching factor and to speed up pruning)
  - least constrained last

  Eg.: max degree, farthest, earliest due date, etc.
- Order-Domain-Values
  - greedy
  - least constraining value heuristic
    (leaves maximum flexibility for subsequent variable assignments)
  - maximal regret
    implements a kind of look ahead

2) What are the implications of the current variable assignments for the other unassigned variables?

Propagating information through constraints:

- Implicit in Select-Unassigned-Variable
- Forward checking (coupled with Minimum Remaining Values)
- Constraint propagation in CSP
  - arc consistency: force all (directed) arcs uv to be consistent:
    $\exists$ a value in $D(v)$: $\forall$ values in $D(u)$, otherwise detects inconsistency

    can be applied as preprocessing or as propagation step after each assignment (Maintaining Arc Consistency)

    Applied repeatedly

Propagation: An Example

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA*=*red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q*=*green*	(R)	B	(G)	R B	R G B	B	R G B
After *V*=*blue*	(R)	B	(G)	R	(B)		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

3) When a path fails – that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?

Backtracking-Search

- chronological backtracking, the most recent decision point is revisited
- backjumping, backtracks to the most recent variable in the conflict set (set of previously assigned variables connected to X by constraints).

An Empirical Comparison

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV
USA	(> 1,000K)	(> 1,000K)	2K	60
n-Queens	(> 40,000K)	13,500K	(> 40,000K)	817K
Zebra	3,859K	1K	35K	0.5K
Random 1	415K	3K	26K	2K
Random 2	942K	27K	77K	15K

Median number of consistency checks

A*, a best-first search

A* search

- The priority assigned to a node x is determined by the function

$$f(x) = g(x) + h(x)$$

$g(x)$: cost of the path so far
$h(x)$: heuristic estimate of the minimal cost to reach the goal from x.

- It is optimal if $h(x)$ is an
  - admissible heuristic: *never overestimates* the cost to reach the goal
  - consistent: $h(n) \leq c(n, a, n') + h(n')$

A* best-first search

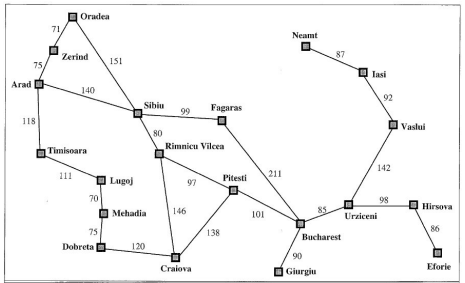


Figure 3.2 A simplified road map of part of Romania.

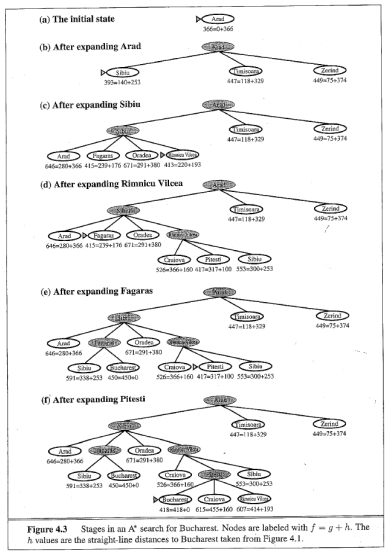


Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

A* search

Possible choices for admissible heuristic functions

- optimal solution to an easily solvable **relaxed problem**
- optimal solution to an easily solvable **subproblem**
- learning from experience by gathering statistics on state features
- preferred heuristics functions with higher values (provided they do not overestimate)
- if several heuristics available $h_1, h_2, \ldots, h_m$ and not clear which is the best then:

$$h(x) = \max\{h_1(x), \ldots, h_m(x)\}$$

A* search

Drawbacks

- Time complexity: In the worst case, the number of nodes expanded is exponential,
  (but it is polynomial when the heuristic function h meets the following condition:
  $$|h(x) - h^*(x)| \leq O(\log h^*(x))$$
  h^* is the optimal heuristic, the exact cost of getting from x to the goal.)
- Memory usage: In the worst case, it must remember an exponential number of nodes.
  Several variants: including iterative deepening A* (IDA*), memory-bounded A* (MA*) and simplified memory bounded A* (SMA*) and recursive best-first search (RBFS)

Outline

1. Complete Search Methods
   - General Search Methods
   - A* search
2. Incomplete Search Methods
3. Metaheuristics
   - Rollout/Pilot Method
   - Beam Search
   - Iterated Greedy
   - GRASP
   - Adaptive Iterated Construction Search
   - Multilevel Refinement
4. Heuristics for TSP

Incomplete Search

Complete search is often better suited when ...

- proofs of insolubility or optimality are required;
- time constraints are not critical;
- problem-specific knowledge can be exploited.

Heuristics are often better suited when ...

- non linear constraints and non linear objective function;
- reasonably good solutions are required within a short time;
- problem-specific knowledge is rather limited.

Greedy algorithms

Greedy algorithms (aka best-first)

- Strategy: always make the choice that is best at the moment.
- They are not generally guaranteed to find globally optimal solutions (but sometimes they do: Minimum Spanning Tree, Single Source Shortest Path, etc.)

We will see problem sepcific examples

Metaheuristics

Beyond best-first search

Based on backtracking

- Bounded backtrack
- Credit-based search
- Limited Discrepancy Search
- Barrier Search
- Randomization in Tree Search

Based on other ideas
(and simpler to implement)

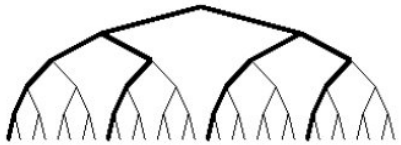
- Rollout/Pilot Method
- Beam Search
- Iterated Greedy
- GRASP
- Adaptive Iterated Construction Search
- Multilevel Refinement

Bounded backtrack

Bounded-backtrack search:

bbs(10)

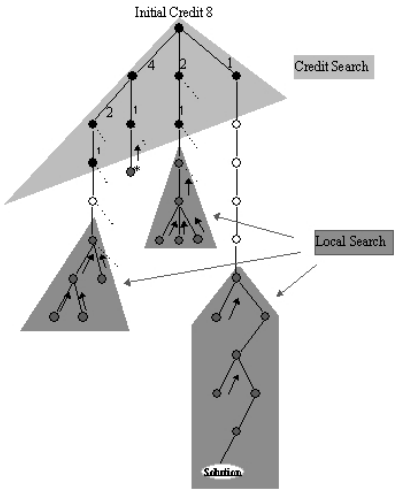
Depth-bounded, then bounded-backtrack search:



dbs(2, bbs(0))

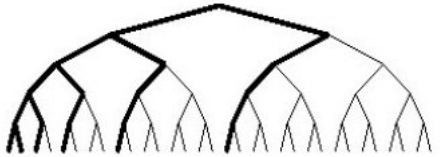
Credit-based search

- Key idea: important decisions are at the top of the tree
- Credit = backtracking steps
- Credit distribution: one half at the best child the other divided among the other children.
- When credits run out follow deterministic best-search
- In addition: allow limited backtracking steps (eg, 5) at the bottom
- Control parameters: initial credit, the distribution of credit among the children, and the amount of local backtracking at the bottom.



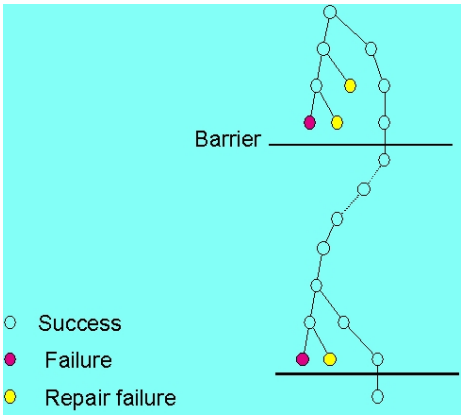
Limited Discrepancy Search

- Key observation that often the heuristic used in the search is nearly always correct with just a few exceptions.
- Explore the tree in increasing number of discrepancies, modifications from the heuristic choice.
- Eg: count one discrepancy if second best is chosen
  count two discrepancies either if third best is chosen or twice the second best is chosen
- Control parameter: the number of discrepancies



Barrier Search

- Extension of LDS
- Key idea: we may encounter several, independent problems in our heuristic choice. Each of these problems can be overcome locally with a limited amount of backtracking.
- At each barrier start LDS-based backtracking



Randomization in Tree Search

The idea comes from complete search

- Dynamical selection of solution components in construction or choice points in backtracking.
- Randomization of construction method or selection of choice points in backtracking while still maintaining the method complete
  $\rightsquigarrow$ *randomized systematic search*.

Randomization can also be used in incomplete search

Outline

1. Complete Search Methods
   - General Search Methods
   - A* search
2. Incomplete Search Methods
3. Metaheuristics
   - Rollout/Pilot Method
   - Beam Search
   - Iterated Greedy
   - GRASP
   - Adaptive Iterated Construction Search
   - Multilevel Refinement
4. Heuristics for TSP

Rollout/Pilot Method

Derived from A*

- Each candidate solution is a collection of m components $S = (s_1, s_2, \ldots, s_m)$.
- Master process adds components sequentially to a partial solution $S_k = (s_1, s_2, \ldots s_k)$
- At the k-th iteration the master process evaluates feasible components to add based on an heuristic look-ahead strategy.
- The evaluation function $H(S_{k+1})$ is determined by sub-heuristics that complete the solution starting from S_k
- Sub-heuristics are combined in $H(S_{k+1})$ by
  - weighted sum
  - minimal value

Speed-ups:

- halt whenever cost of current partial solution exceeds current upper bound
- evaluate only a fraction of possible components

Beam Search

Possible extension of tree based construction heuristics:

- maintains a set B of bw (beam width) partial candidate solutions
- at each iteration extend each solution from B in fw (filter width) possible ways
- rank each $bw \times fw$ candidate solutions and take the best bw partial solutions
- complete candidate solutions obtained by B are maintained in B_f
- Stop when no partial solution in B is to be extended

Iterated Greedy

Key idea: use greedy construction

- alternation of Construction and Deconstruction phases
- an acceptance criterion decides whether the search continues from the new or from the old solution.

Iterated Greedy (IG):
determine initial candidate solution s
while termination criterion is not satisfied **do**
  $r := s$
  greedily destruct part of s
  greedily reconstruct the missing part of s
  based on acceptance criterion,
    keep s or revert to $s := r$

Extension: Squeaky Wheel

Key idea: solutions can reveal problem structure which maybe worth to exploit.

Use a greedy heuristic repeatedly by prioritizing the elements that create troubles.

Squeaky Wheel

- Constructor: greedy algorithm on a sequence of problem elements.
- Analyzer: assign a penalty to problem elements that contribute to flaws in the current solution.
- Prioritizer: uses the penalties to modify the previous sequence of problem elements. Elements with high penalty are moved toward the front.

Hybridize with subsidiary perturbative search

GRASP

Greedy Randomized Adaptive Search Procedure

Key Idea: Combine randomized constructive search with subsequent local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by local search.
- Local search methods often require substantially fewer steps to reach high-quality solutions when initialized using greedy constructive search rather than random picking.
- By iterating cycles of constructive + local search, further performance improvements can be achieved.

Greedy Randomized "Adaptive" Search Procedure (GRASP):
while *termination criterion* is not satisfied **do**
  generate candidate solution s using
    subsidiary greedy randomized constructive search
  perform subsidiary local search on s

- Randomization in *constructive search* ensures that a large number of good starting points for *subsidiary local search* is obtained.
- Constructive search in GRASP is 'adaptive' (or dynamic): Heuristic value of solution component to be added to a given partial candidate solution may depend on solution components present in it.
- Variants of GRASP without local search phase (aka *semi-greedy heuristics*) typically do not reach the performance of GRASP with local search.

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a restricted candidate list (RCL).
- RCLs are constructed in each step using a *heuristic function* h.
  - RCLs based on cardinality restriction comprise the k best-ranked solution components. (k is a parameter of the algorithm.)
  - RCLs based on value restriction comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$, where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)

Example: GRASP for SAT [Resende and Feo, 1996]

- **Given:** CNF formula F over variables $x_1, \ldots, x_n$
- **Subsidiary constructive search:**
  - start from empty variable assignment
  - in each step, add one atomic assignment (*i.e.*, assignment of a truth value to a currently unassigned variable)
  - heuristic function $h(i, v) :=$ number of clauses that become satisfied as a consequence of assigning $x_i := v$
  - RCLs based on cardinality restriction (contain fixed number k of atomic assignments with largest heuristic values)
- **Subsidiary local search:**
  - iterative best improvement using 1-flip neighborhood

GRASP has been applied to many combinatorial problems, including:

- SAT, MAX-SAT
- various scheduling problems

Extensions and improvements of GRASP:

- reactive GRASP (*e.g.*, dynamic adaptation of α during search)

Adaptive Iterated Construction Search (AICS):
initialise weights
while *termination criterion* is not satisfied: **do**
  generate candidate solution s using
    subsidiary randomized constructive search
  perform subsidiary local search on s
  adapt weights based on s

Adaptive Iterated Construction Search

Key Idea: Alternate construction and local local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate *weights* with possible decisions made during constructive search.
- Initialize all weights to some small value τ_0 at beginning of search process.
- After every cycle (= constructive + local local search phase), update weights based on solution quality and solution components of current candidate solution.

Subsidiary constructive search:

- The solution component to be added in each step of *constructive search* is based on *weights* and heuristic function $\mathfrak{h}$.
- $\mathfrak{h}$ can be standard heuristic function as, *e.g.*, used by greedy heuristics
- It is often useful to design solution component selection in constructive search such that any solution component may be chosen (at least with some small probability) irrespective of its weight and heuristic value.

Subsidiary local local search:

- As in GRASP, local local search phase is typically important for achieving good performance.
- Can be based on Iterative Improvement or more advanced LS method (the latter often results in better performance).
- Tradeoff between computation time used in construction phase *vs* local search phase (typically optimized empirically, depends on problem domain).

Weight updating mechanism:

- Typical mechanism: increase weights of all solution components contained in candidate solution obtained from local search.
- Can also use aspects of search history; *e.g.*, current *incumbent candidate solution* can be used as basis for weight update for additional intensification.

Example: A simple AICS algorithm for the TSP (1)

(Based on Ant System for the TSP [Dorigo et al. 1991].)

- Search space and solution set as usual (all Hamiltonian cycles in given graph G). However represented in a construction tree T.
- Associate weight τ_{ij} with each edge (i, j) in G and T
- Use heuristic values $\eta_{ij} := 1/w((i, j))$.
- Initialize all weights to a small value τ_0 (parameter).
- *Constructive search* starts with randomly chosen vertex and iteratively extends partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{l \in N'(i)} [\tau_{il}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}$$

Example: A simple AICS algorithm for the TSP (2)

- *Subsidiary local search* = typical iterative improvement
- *Weight update* according to

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(i, j, s')$$

where $\Delta(i, j, s') := 1/f(s')$, if edge (i, j) is contained in the cycle represented by s', and 0 otherwise.

- Criterion for weight increase is based on intuition that edges contained in short round trips should be preferably used in subsequent constructions.

Multilevel Refinement

Key idea: make the problem recursively less refined creating a hierarchy of approximations of the original problem.

- an initial solution is found on the original problem or at a refined level
- solutions are iteratively refined at each level
- use of projection operators to transfer the solution from one level to another

Multilevel Refinement
while Termination criterion is not satisfied **do**
  coarse the problem π_0 into π_i, $i = 0, \ldots, k$ successive non degenerate problems
  $i = k$
  determine an initial candidate solution for π_k
  repeat
    $i = i - 1$
    extend the solution found in π_{i+1} to π_i
    use subsidiary local search to refine the solution on π_i
  until $i \geq 0$;

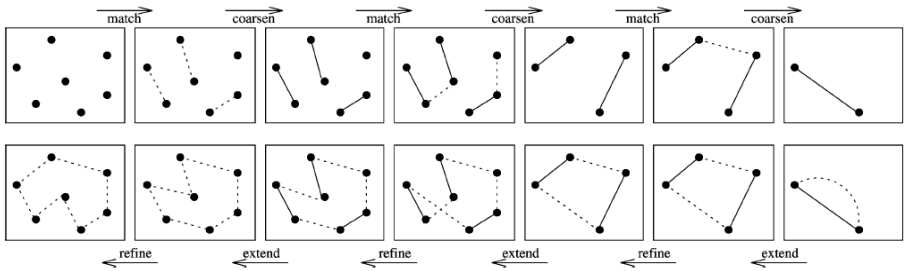
Example: Multilevel Refinement for TSP

Coarsen: fix some edges and contract vertices

Solve: matching
(always match vertices with the nearest unmatched neighbors)

Extend: uncontract vertices

Refine: LK heuristic



Note

- crucial point: the solution to each refined problem must contain a solution of the original problem (even if it is a poor solution)

Applications to

- Graph Partitioning
- Traveling Salesman
- Graph Coloring

Outline

1. Complete Search Methods
   General Search Methods
   A* search
2. Incomplete Search Methods
3. Metaheuristics
   Rollout/Pilot Method
   Beam Search
   Iterated Greedy
   GRASP
   Adaptive Iterated Construction Search
   Multilevel Refinement
4. Heuristics for TSP

Construction Heuristics

Construction heuristics specific for TSP

- Heuristics that Grow Fragments
  - Nearest neighborhood heuristics
  - Double-Ended Nearest Neighbor heuristic
  - Multiple Fragment heuristic (aka, greedy heuristic)
- Heuristics that Grow Tours
  - Nearest Addition
  - Farthest Addition
  - Random Addition
  - Clarke-Wright savings heuristic
  - Nearest Insertion
  - Farthest Insertion
  - Random Insertion
- Heuristics based on Trees
  - Minimum span tree heuristic
  - Christofides' heuristics
  - Fast recursive partitioning heuristic

Construction Heuristics for TSP

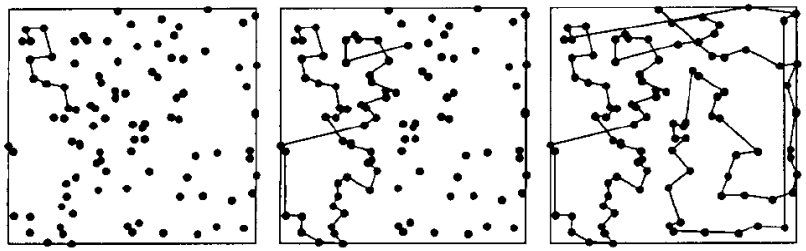


Figure 1. The Nearest Neighbor heuristic.

Construction Heuristics for TSP

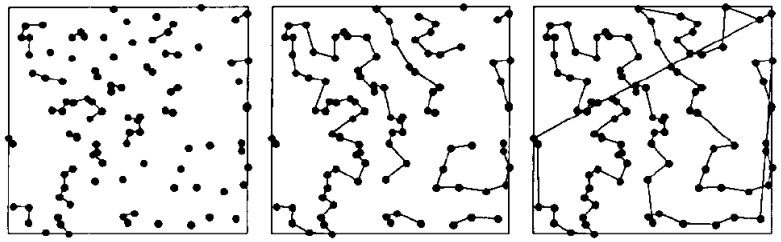


Figure 5. The Multiple Fragment heuristic.

Construction Heuristics for TSP

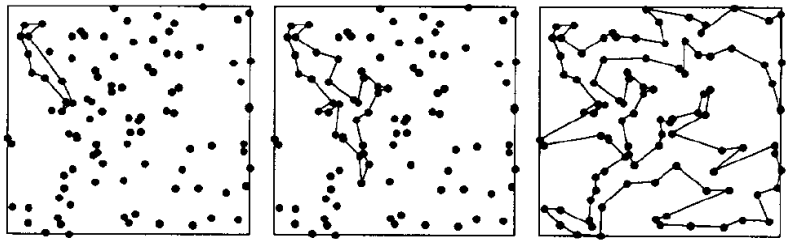


Figure 8. The Nearest Addition heuristic.

Construction Heuristics for TSP

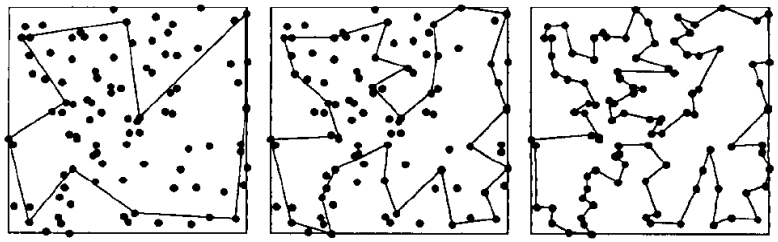


Figure 11. The Farthest Addition heuristic.

Construction Heuristics for TSP

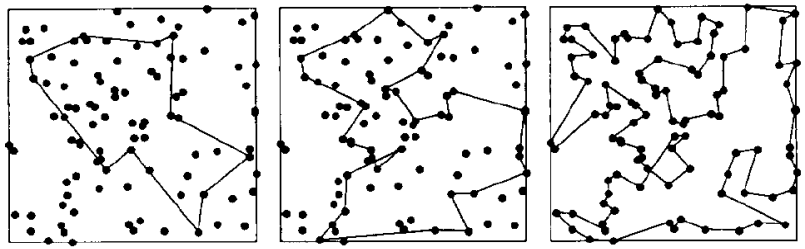


Figure 14. The Random Addition heuristic.

Construction Heuristics for TSP

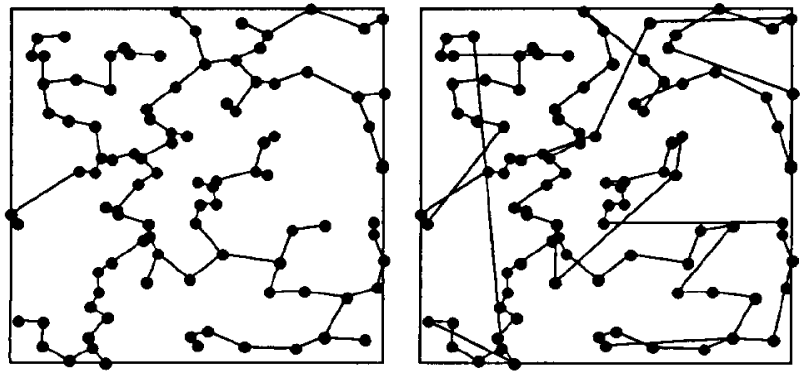


Figure 18. The Minimum Spanning Tree heuristic.

Construction Heuristics for TSP

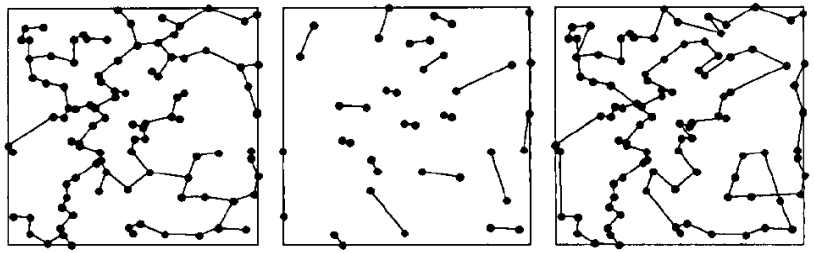


Figure 19. Christofides' heuristic.