BRANCH-AND-BOUND

The book presents branch-and-bound as a variation of backtracking in which the choice set is tried in decreasing order of bounds.

However, branch-and-bound is usually a more general scheme. It often involves keeping all active nodes in a priority queue, and processing nodes with higher priority first (priority is given by upper bound).

Here is the book's version of branch-and-bound:

Algorithm $\textsc{BranchAndBound}(l)$

```
external B(), PROFIT();
global C_l (l = 0, 1, ...)
if ([x_0, x_1, ..., x_{l-1}] is a feasible solution) then
   P <- PROFIT([x_0, x_1, ..., x_{l-1}])
   if (P > OptP) then
      OptP <- P;
      OptX <- [x_0, x_1, ..., x_{l-1}];
Compute C_l;
count <- 0;
for each (x ∈ C_l) do
     x_l <- x;
     nextchoice[count] <- x;
     nextbound[count] <- B([x_0, x_1, ..., x_{l-1}, x]);
     count <- count + 1;
Sort nextchoice and nextbound by decreasing order of nextbound;
for i <- 0 to count - 1 do
     if (nextbound[i] ≤ OptP) then return;
     x_l <- nextchoice[i];
     BranchAndBound(l + 1);
```

HEURISTIC SEARCH

Heuristic Search vs Exhaustive Search

Exhaustive Search

Types of methods and their uses:

- Backtracking (backtracking with bounding):
  - Find all feasible solutions.
  - Find one optimal solution.
  - Find all optimal solutions.
- Branch-and-Bound:
  - Find one optimal solution.

Heuristic Search

Types of problem it can be applied to:

- Find 1 optimal solution.
- Find a "close to" optimal solution (the best solution we manage).

Heuristics methods we will study:

- Hill-climbing
- Simulated annealing
- Tabu search
- Genetic algorithm

Characteristics of heuristic search:

- The state space is not fully explored.
- Randomization is often employed.
- There is a concept of neighbourhood search.
- **Heuristics** are applied to explore the solutions.
  The word "heuristics" means "serving or helping to find or discover" or "proceeding by trial and error".

A general framework for heuristic search

Generic Optimization Problem (maximization):

Instance: A finite set $\mathcal{X}$.
an objective function $P : \mathcal{X} \to Z$.
m feasibility functions $g_j : \mathcal{X} \to Z$, $1 \leq j \leq m$.
Find: the maximum value of $P(X)$
subject to $X \in \mathcal{X}$ and $g_j(X) \geq 0$, for $1 \leq j \leq m$.

Exercise: pick your favorite combinatorial optimization problem and write it in this framework.

Designing a heuristic search:

1. Define a **neighbourhood function** $N : \mathcal{X} \to 2^{\mathcal{X}}$.

   E.g. $N(X) = \{X_1, X_2, X_3, X_4, X_5\}$.

2. Design a **neighbourhood search**:
   Algorithm that finds a fasible solution on the neighbourhood of a feasible solution X.

   There are two types of neghbourhood searches:

   - Exhaustive (chooses best profit among neighbour points)
   - Randomized (picks a random point among the neighbour points)

Defining a neighbourhood function

$N : \mathcal{X} \rightarrow 2^{\mathcal{X}}$, $N(X) \subseteq \mathcal{X}$.
$N(X)$ should be elements that are similar or "close to" X.
$N(X)$ may contain infesible elements of $\mathcal{X}$.

Examples of neighbourhood functions:
Let d_0 be a constant positive integer.

$$N_{d_0}(X) = \{Y \in \mathcal{X} : dist(X, Y) \leq d_0\},$$

- $\mathcal{X} = \{0, 1\}^n$, all binary n-tuples.

  Here $dist$ is the Hamming distance.

  $N_1([010]) = \{[000], [110], [011], [010]\}$.

$$|N_{d_0}(X)| = \sum_{i=0}^{d_0} \binom{n}{i}.$$

- $\mathcal{X} =$ all permutations of $\{1, 2, \ldots, n\}$.

  Let $\alpha = [\alpha_1, \ldots, \alpha_n]$ and $\beta = [\beta_1, \ldots, \beta_n]$ be two permutations.
  Define distance as follows: $dist(\alpha, \beta) = |\{i : \alpha_i \neq \beta_i\}|$.

  Note that $N_1(X) = \{X\}$ is not very useful; we need $d_0 > 1$.

  $N_2([1, 2, 3, 4]) = \{[1, 2, 3, 4], [2, 1, 3, 4], [3, 2, 1, 4],$
  $[4, 2, 3, 1], [1, 3, 2, 4], [1, 4, 3, 2], [1, 2, 4, 3]\}$

  $|N_2(X)| = 1 + \binom{n}{2}$.

Designing a neighbourhood search algorithm

Neighbourhood Search Algorithm

Input: X
Output: $Y \in N(X) \setminus X$ such that Y is feasible, or "fail".

Possible Neighbourhood Search Strategies:

1. Find a feasible solution $Y \in N(X) \setminus \{X\}$ such that $P(Y)$ is maximized.
   Return "fail" if there is no feasible solution in $N(X) \setminus \{X\}$.

2. Find a feasible solution $Y \in N(X) \setminus \{X\}$ such that $P(Y)$ is maximized.
   if $P(Y) > P(X)$ then return Y; else return "fail".
   (steepest ascent method)

3. Find any feasible solution $Y \in N(X) \setminus \{X\}$.
   Return "fail" if there is no feasible solution in $N(X) \setminus \{X\}$.

4. Find any feasible solution $Y \in N(X) \setminus \{X\}$.
   if $P(Y) > P(X)$ then return Y; else return "fail".

Strategies 1 and 2 may be exhaustive.
Strateges 3 and 4 are usually randomized.

A generic heuristic search algorithm

Given N, a neighbourhood function, the heuristic algorithm h_N does either of the following:

- Perform one neighbourhood search (using one of the strategies)
- Perform a sequence of j neighbourhood searches $[X = X_0, X_1, \ldots, X_j = Y]$, where you get from X_i to X_{i+1} through a neighbourhood search.

Let c_{max} be the maximum number of iterations allowed for the search.

Algorithm GenericHeuristicSearch(c_{max})

    $c \leftarrow 0$;
    Select a feasible solution $X \in \mathcal{X}$;
    $X_{best} \leftarrow X$; (stores best so far)
    while ($c \leq c_{max}$) do
        $Y \leftarrow h_N(X)$;
        if ($Y \neq$ "fail") then
            $X \leftarrow Y$;
            if ($P(X) > P(X_{best})$) then $X_{best} \leftarrow X$;
        [else $c \leftarrow c_{max} + 1$; (add this if h_N is not randomized)]
        $c \leftarrow c + 1$;
    return X_{best};

Design Strategies for Heuristic Algorithms

1. Hill-Climbing

Idea: Go up the hill continuously, stop when stuck.
Problem: it can get stuck in a local optimum.
Improvement: run the algorithm many times from random start X.

For Hill-Climbing, $h_N(X)$ returns:

- $Y \in N(X)$ such that Y is feasible and $P(Y) > P(X)$,
- or, otherwise, "fail".

Algorithm GenericHillClimbing()

```
Select a feasible solution X ∈ 𝒳.
X_best ← X; searching ← true;
while (searching) do
    Y ← h_N(X);
    if (Y ≠ "fail") then
      X ← Y;
      if (P(X) > P(X_best)) then X_best ← X;
    else searching ← false;
return X_best;
```

Hill-climbing will get trapped in a local optimum.
Other search strategies, such as simulated annealing and tabu search, try to escape from local optima.

2. Simulated annealing

- Analogy with a method of cooling metal: annealing. Temperature T decreases at each iteration, according to a **cooling schedule**: Initally $T \leftarrow T_0$; later $T \leftarrow \alpha T$ for a fixed $0 < \alpha < 1$.
- Going uphill is always accepted.
- Going downhill is sometimes accepted with a probability based on how much downhill we go and on the current temperature. Given $Y = h_N(X)$ with $P(Y) \leq P(X)$, accept Y with probability $e^{\frac{P(Y)-P(X)}{T}}$. (We get pickier as we progress.)

Algorithm GenericSimulatedAnnealing(c_{max}, T_0, α)

```
    c ← 0; T ← T_0;
    Select a feasible solution X ∈ 𝒳; X_best ← X;
    while (c ≤ c_max) do
        Y ← h_N(X); // this is usually a randomized choice
        if (Y ≠ "fail") then
          if (P(Y) > P(X)) then
            X ← Y;
            if (P(X) > P(X_best)) then X_best ← X;
          else
            r ← random(0,1);
            if (r < e^((P(Y)-P(X))/T)) then X ← Y;
        c ← c + 1;
        T ← αT;
    return X_best;
```

3. Tabu Search

Choose $Y \in N(X) \setminus \{X\}$ such that Y is feasible and $P(X)$ is maximum among all such elements (exhaustive neighbourhood search).
It may happen that $P(Y) < P(X)$ (we escape from a local optimum).

What may be the risk? Cycling.
When going downhill from X to Y we may go back from X to Y.
Indeed, cycling may take several steps, such as
$X \rightarrow Y \rightarrow Z \rightarrow X$.

Tabu-search uses a strategy for avoiding cycling: a **tabu list**.
After a move $X \rightarrow Y$,
we forbit the application of $\textsc{change}(Y, X)$ for L iterations (L is the lifetime of the tabu list).

Example:
$\mathcal{X} = \{0,1\}^n$, using $N_1(X) = \{Y \in \mathcal{X} : dist(X,Y) = 1\}$.
$X = [0100]$ and $Y = [0101]$, we have that $\textsc{change}(Y, X) = 4 =$ index of coordinate that was swapped.

Suppose $L = 2$.

sequence of points:	[0100]	[0101]	[1101]	[1001]	[1011]
tabu list:		4	4,1	1,2	2,3

So any sequence that cycles $X \rightarrow \ldots \rightarrow X$ has length at least $2L$. Choosing $L = 1$ is typical.

TABULIST is implemented as a list where TABULIST[c] $= \delta$, where δ is the designated forbidden (tabu) change at iteration c.

In absolute no circumstance implement TABULIST as an array indexed by the number of iterations! Instead, implement TABULIST as a queue of length L. Note that the algorithm may mislead you to think you are using such an array; careful!

For tabu search, $h_N(X) = Y$, where

- $Y \in N(X)$, Y is feasible;
- CHANGE$(X, Y) \notin \{$ TABULIST$[d] : c - L \leq d \leq c - 1\}$;
- $P(Y)$ is maximum among all such feasible elements.

Algorithm GENERICTABUSEARCH(c_{max}, L)

$c \leftarrow 1$;
Select a feasible solution $X \in \mathcal{X}$.
$X_{best} \leftarrow X$;
while ($c \le c_{max}$) do
  $N \leftarrow N(X) \setminus \{F$: CHANGE$(X, F) \in$ TABULIST$[d]$,
    $c - L \le d \le c - 1\}$; (typo corrected)
  for each ($Y \in N$) do
    if (Y is infeasible) then $N \leftarrow N \setminus \{Y\}$;
  if ($N = \emptyset$) then return X_{best};
  Find $Y \in N$ such that $P(Y)$ is maximum;
  TABULIST$[c]$ $\leftarrow$CHANGE(Y, X);
  $X \leftarrow Y$;
  if ($P(X) > P(X_{best})$) then $X_{best} \leftarrow X$;
  $c \leftarrow c + 1$;
return X_{best};

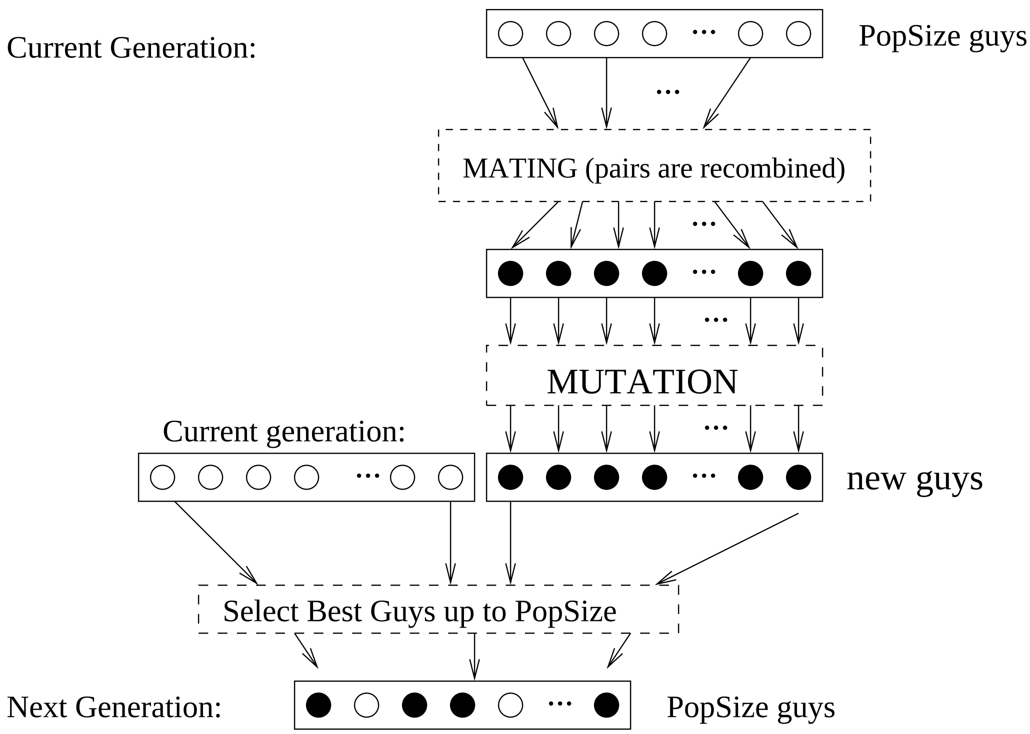
In the real algorithm, TABULIST must be a queue of length L!!!

So, the operation
TABULIST$[c]$ $\leftarrow$CHANGE(Y, X);
must be implemented as:
TABULIST.insert(CHANGE(Y, X)); (only keeps last L elements)
and the line: $N \leftarrow N(X) \setminus \{F$:
CHANGE$(X, F) \in$TABULIST$[d], c - L \le d \le c - 1\}$
should be understood as:
$N \leftarrow N(X) \setminus \{F$: CHANGE$(X, F) \in$ TABULIST$\}$;

4. Genetic Algorithms

More complex than neighbourhood search.
Fix a number PopSize (population size).

One iteration works as follows:

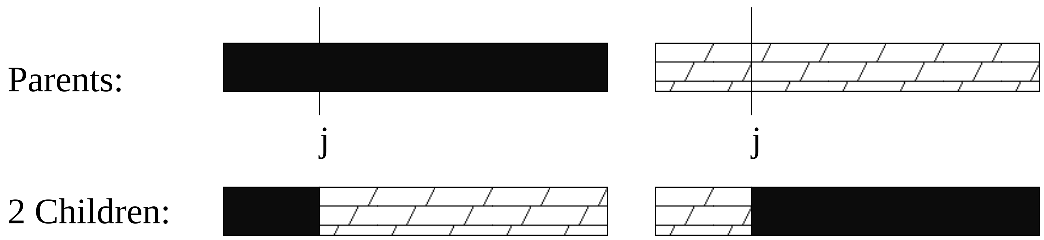


Iterate as many generations as you like.

Mating Strategies (Recombination)

Producing children from parents.

1. Crossover.
   Let j be a crossover point.



   Example: $j = 3$

Parents:	[110|1101001]	[100|1000101]
Children:	[110|1000101]	[100|1101001]

2. Partially matched crossover (for permutations)

   Two crossover points: $1 \leq j < k \leq n$

   Example: $j = 3$ and $k = 6$
   $\alpha = [3, 1, \underline{4, 7, 6, 5}, 2, 8]$ $\quad \beta = [8, 6, \underline{4, 3, 7, 1}, 2, 5]$

swap	α	β
$4 \leftrightarrow 4$	$[3, 1, 4, 7, 6, 5, 2, 8]$	$[8, 6, 4, 3, 7, 1, 2, 5]$
$7 \leftrightarrow 3$	$[7, 1, 4, 3, 6, 5, 2, 8]$	$[8, 6, 4, 7, 3, 1, 2, 5]$
$6 \leftrightarrow 7$	$[6, 1, 4, 3, 7, 5, 2, 8]$	$[8, 7, 4, 6, 3, 1, 2, 5]$
$5 \leftrightarrow 1$	$[6, 5, 4, 3, 7, 1, 2, 8]$	$[8, 7, 4, 6, 3, 5, 2, 1]$

Mating Schemes

Kids may be infeasible: incorporate constraints as penalties.

- Random monogamy with 2 kids per couple: randomly partition population into pairs, with two kids produced by each pair.
- Make better parents having more kids:
  measure parent fitness by objective function; parents with higher fitness produce more kids.

Algorithm GENERICGENETICALGORITHM($PopSize, c_{max}$)

```
    c ← 1;
    Select an initial population 𝒫 with PopSize feasible solutions;
    for each X ∈ 𝒫 do X ← h_N(X);    [mutation]
    X_best ← element in 𝒫 with maximum profit;
    while (c ≤ c_max) do
        Construct a pairing of the elements in 𝒫;
        𝒬 ← 𝒫
        for each pair (W, X) in the pairing do
            (Y, Z) ← rec(W, X);    [recombination/mating]
            Y ← h_N(Y);    [mutation]
            Z ← h_N(Z);    [mutation]
            𝒬 ← 𝒬 ∪ {Y, Z};
        Let 𝒫 be the best PopSize members of 𝒬;
        Let Y be the element in 𝒫 with maximum profit;
        if (P(Y) > P(X_best)) then X_best ← Y;
        c ← c + 1;
    return X_best;
```

HEURISTIC SEARCHES APPLIED TO VARIOUS PROBLEMS

Lecture 8: Hill-climbing Algorithms

In Lecture 8, we have seen two Hillclimbing algorithms:

- A steepest ascent algorithm for uniform graph partition (Section 5.3)
- A hill-climbing algorithm for Steiner triple systems (Section 5.4)

Slides are missing for this lecture.

Two heuristics for the Knapsack Problem

Knapsack (Optimization) Problem

Instance: Profits $p_0, p_1, \ldots, p_{n-1}$
Weights $w_0, w_1, \ldots, w_{n-1}$
Knapsack capacity M

Universe: $\mathcal{X} = \{0,1\}^n$ (set of all n-tuples)
an n-tuple $[x_0, x_1, \ldots, x_{n-1}]$ is feasible if
$\Sigma_{i=0}^{n-1} w_i x_i \leq M$.

Objective: maximize $P(X) = \Sigma_{i=0}^{n-1} p_i x_i$.

A Simulated Annealing Algorithm for Knapsack

Neighbourhood function:

$$N(X) = N_1(X) = \{Y \in \{0,1\}^n : dist(X,Y) = 1\}$$

Algorithm $\textsc{KnapsackSimulatedAnnealing}(c_{max}, T_0, \alpha)$

```
c ← 0; T ← T_0;
X ← [x_0, x_1, ..., x_{n-1}] = [0, 0, ..., 0];
CurW ← 0; X_best ← X;
while (c ≤ c_max) do
        j ← randomInt(0, n − 1);
        Y ← X;
        y_j ← 1 − x_j;    (swap j coordinate of X)
        if (y_j = 1) and (curW + w_j > M) then Y ← fail;
        if (Y ≠ fail) then
          if (y_j = 1) then
            X ← Y;
            curW ← curW + w_j;
            if P(X) > P(X_best) then X_best ← X;
          else
              r ← random(0, 1);
              if (r < e^{−p_j/T}) then
                X ← Y;
                curW ← curW − w_j;
        c ← c + 1;
        T ← αT;
return (X_best);
```

Experimental results from Table 5.3 (page 178):

Tabu Search for Knapsack

We will use the same neighbourhood $N_1(.)$.

Do exhaustive search on the neighbourhood in order to find the best way to update the current solution.

Instead of Profit imporvement only, we look for improvements based on the ratio p_i/w_i:

1. Chose i with maximum p_i/w_i among the indexes j where $x_j = 0$, j is not on TABULIST, and changing x_j to 1 des not exceed M.
2. If there is no j as above, then choose i with minimum p_i/w_i among the indexes j where $x_j = 1$ and j is not on TABULIST.

This can be expressed by saying that we want to maximize

$$(-1)^{x_j} \frac{p_j}{w_j}.$$

Algorithm $\text{KNAPSACKTABUSEARCH}(c_{max}, L)$

$c \leftarrow 1;$

Select a random feasible $X = [x_0, x_1, \ldots, x_{n-1}] \in \{0,1\}^n$;

$curW \leftarrow \Sigma_{i=0}^{n-1} x_i w_i;$

$X_{best} \leftarrow X;$

while $(c \leq c_{max})$ do

  $N \leftarrow \{0, 1, \ldots, n-1\};$

  $start \leftarrow max\{0, c - L\};$

  for $j \leftarrow start$ to $c - 1$ do

    $N \leftarrow N \setminus \{\text{TABULIST}[j]\};$

  for each $(i \in N)$ do

    if $(x_i = 0)$ and $(curW + w_i > M)$ then $N \leftarrow N \setminus \{i\};$

  if $(N = \emptyset)$ then exit;

  Find $i \in N$ such that $(-1)^{x_i} p_i / w_i$ is maximum;

  $\text{TABULIST}[c] \leftarrow i;$

  $x_i \leftarrow 1 - x_i;$ (swap i coordinate)

  if $(x_i = 1)$ then $curW \leftarrow curW + w_i;$

    else $curW \leftarrow curW - w_i;$

  if $P(X) > P(X_{best})$ then $X_{best} \leftarrow X;$

  $c \leftarrow c + 1;$

return $X_{best};$

Experimental results from Tables 5.4 and 5.6 (pages 180-181):

A Genetic Algorithm for the TSP

Traveling Salesman Problem (TSP)

Instance: a complete graph K_n
a cost function $c : V \times V \to R$
Find: a Hamiltonian circuit $[x_0, x_1, \ldots, x_{n-1}]$ that minimizes
$C(X) = c(x_0, x_1) + c(x_1, x_2) + \ldots + c(x_{n-1}, x_0)$

Note that $2n$ permutations represent the same cycle.

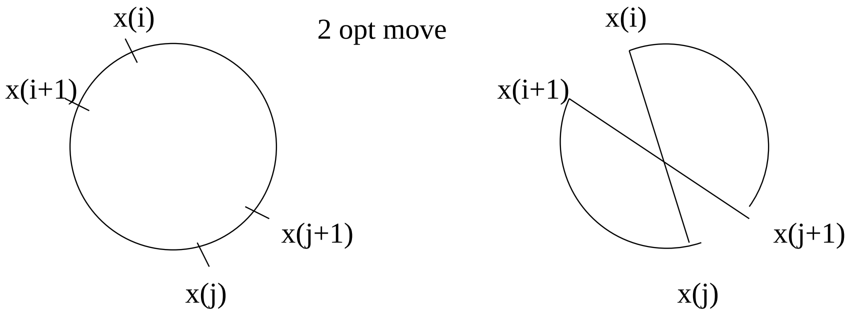
Universe: $\mathcal{X}$ = set of all $n!$ permutations.

Steps:

- Selection of initial population.
- Mutation: steepest ascent 2-opt.
- Recombination using two methods: partially matched crossover and another method.

Mutation

Steepest ascent algorithm based on the 2-opt heuristic:



Gain in applying a 2-opt move:

$$\begin{aligned} G(X,i,j) &= C(X) - C(X_{ij}) \\ &= c(x_i, x_{i+1}) + c(x_j, x_{j+1}) - c(x_{i+1}, x_{j+1}) - c(x_i, x_j) \end{aligned}$$

$N(X) =$ all $Y \in \mathcal{X}$ that can be obtained from X by a 2-opt move.

Algorithm $\textsc{SteepestAscentTwoOpt}(X)$

$done \leftarrow false$;
while ($not\ done$) do
  $done \leftarrow true$; $g_0 \leftarrow 0$;
  for $i \leftarrow 0$ to $n-1$ do
    for $j \leftarrow i+2$ to $n-1$ do
      $g \leftarrow G(X,i,j)$;
      if ($g > g_0$) then
        $g_0 \leftarrow G$; $i_0 \leftarrow i$; $j_0 \leftarrow j$;
  if ($g_0 > 0$) then
    $X \leftarrow X_{i_0,j_0}$;
    $done \leftarrow false$;

Selecting the initial population

Randomly pick one and then mutate it:

Algorithm $\textsc{Select}(popsize)$
for $i \leftarrow 0$ to $popsize - 1$ do
$r \leftarrow \textsc{RandomInteger}(0, n! - 1)$;
$P_i \leftarrow \textsc{PermLexUnrank}(n, r)$;
$\textsc{SteepestAscentTwoOpt}(P_i)$;
return $[P_0, P_1, \ldots, P_{popsize-1}]$;

Two recombination algorithms

1. Partially Matched Crossover

Algorithm $\textsc{PMRec}(A, B)$
$h \leftarrow \textsc{RandomInteger}(10, n/2)$; (length of the substring)
$j \leftarrow \textsc{RandomInteger}(0, n - 1)$; (start of the substring)
$(C, D) \leftarrow \textsc{PartiallyMatchedCrossover}(A, B, j, (h + j) mod\ n)$
$\textsc{SteepestAscentTwoOpt}(C)$;
$\textsc{SteepestAscentTwoOpt}(D)$;
return (C, D);

2. Another Recombination Algorithm

Algorithm $\textsc{MGKRec}(A, B)$

$h \leftarrow \textsc{RandomInteger}(10, n/2)$; (length of the substring)
$j \leftarrow \textsc{RandomInteger}(0, n-1)$; (start of the substring)
$T \leftarrow \emptyset$;
(pick subcycle of length h starting from pos j:)
for $i \leftarrow 0$ to $h-1$ do
    $D[i] \leftarrow B[(i+j) mod n]$;
    $T \leftarrow T \cup \{D[i]\}$;
Complete cycle with permutation in A using guys not already in D in the order prescribed by A:
for $j \leftarrow 0$ to $n-1$ do
    if $A[j] \notin T$ then $D[i] \leftarrow A[j]$;
        $i \leftarrow i+1$;
$\textsc{SteepestAscentTwoOpt}(D)$;
(Similarly build C swapping A and B roles:)
$j \leftarrow \textsc{RandomInteger}(0, n-1)$; (start of the substring)
$T \leftarrow \emptyset$;
for $i \leftarrow 0$ to $h-1$ do
    $C[i] \leftarrow A[(i+j) mod n]$;
    $T \leftarrow T \cup \{C[i]\}$;
for $j \leftarrow 0$ to $n-1$ do
    if $B[j] \notin T$ then $C[i] \leftarrow B[j]$;
        $i \leftarrow i+1$;
$\textsc{SteepestAscentTwoOpt}(C)$;
return (C, D);

Genetic Algorthm for TSP

Algorithm $\textsc{GeneticTSP}(popsize, c_{max})$

$c \leftarrow 1$;
$[P_0, P_1, \ldots, P_{popsize-1}] \leftarrow \textsc{Select}(popsize)$;
Sort $P_0, P_1, \ldots, P_{popsize-1}$ in increasing order of cost.
$X_{best} \leftarrow P_0$;
$BestCost \leftarrow C(P_0)$;
while $(c \leq c_{max})$ do
  for $i \leftarrow 0$ to $popsize/2 - 1$ do
    $(P_{popsize+2i}, P_{popsize+2i+1}) \leftarrow \textsc{Rec}\ (P_{2i}, P_{2i+1})$;
  Sort $P_0, P_1, \ldots, P_{2popsize-1}$ in increasing order of cost.
  $curCost \leftarrow C(P_0)$;
  if $(curCost < BestCost)$ then
    $X_{best} \leftarrow P_0$;
    $BestCost \leftarrow curCost$;
  $c \leftarrow c + 1$;
return X_{best};

Note: REC represents either of the two recombination algorithms.

Experimental results from Tables 5.7 and 5.8 (pages 186-187):