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 `BRANCHANDBOUND(l)`

1. `external B(), PROFIT();`
2. `global C_i (l = 0, 1, ...)`
3. `if ([x_0, x_1, ..., x_{l-1}] is a feasible solution) then`
4. `P ← PROFIT([x_0, x_1, ..., x_{l-1}])`
5. `if (P > OptP) then`
6. `OptP ← P;`
7. `OptX ← [x_0, x_1, ..., x_{l-1}];`
8. `Compute C_i;`
9. `count ← 0;`
10. `for each (x ∈ C_i) do`
11. `x_i ← x;`
12. `nextchoice[count] ← x;`
13. `nextbound[count] ← B([x_0, x_1, ..., x_{l-1}, x]);`
14. `count ← count + 1;`
15. `Sort nextchoice and nextbound by decreasing order of nextbound;`
16. `for i ← 0 to count − 1 do`
17. `if (nextbound[i] ≤ OptP) then return;`
18. `x_i ← nextchoice[i];`
19. `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 feasible solution on the neighbourhood of a feasible solution $X$.

   There are two types of neighbourhhood searches:
   - Exhaustive (chooses best profit among neighbour points)
   - Randomized (picks a random point among the neighbour points)
Defining a neighbourhood function

\[ N : \mathcal{X} \rightarrow 2^\mathcal{X}, \ N(X) \subseteq \mathcal{X}. \]
\[ N(X) \text{ should be elements that are similar or } \text{“close to” } X. \]
\[ N(X) \text{ may contain infeasible elements of } \mathcal{X}. \]

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

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

- \( \mathcal{X} = \{0, 1\}^n \), all binary \( n \)-tuples.
  Here \( \text{dist} \) is the Hamming distance.
  \[ N_1([010]) = \{ [000], [110], [011], [010] \}. \]

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

- \( \mathcal{X} = \text{all permutations of } \{1, 2, \ldots, n\}. \)
  Let \( \alpha = [\alpha_1, \ldots, \alpha_n] \) and \( \beta = [\beta_1, \ldots, \beta_n] \) be two permutations.
  Define distance as follows: \( \text{dist}(\alpha, \beta) = |\{ i : \alpha_i \neq \beta_i \}|. \)
  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.

Strategies 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 \( c_{\text{max}} \) be the maximum number of iterations allowed for the search.

Algorithm **GENERICHEURISTICSEARCH**(\( c_{\text{max}} \))

\[ c \leftarrow 0; \]
Select a feasible solution \( X \in \mathcal{X} \);

\[ X_{\text{best}} \leftarrow X; \text{ (stores best so far)} \]
while (\( c \leq c_{\text{max}} \)) do

\[ Y \leftarrow h_N(X); \]
if (\( Y \neq \text{ “fail”} \)) then

\[ X \leftarrow Y; \]
if \( (P(X) > P(X_{\text{best}})) \) then \( X_{\text{best}} \leftarrow X; \)
[else \( c \leftarrow c_{\text{max}} + 1; \) (add this if \( h_N \) is not randomized)]
\[ c \leftarrow c + 1; \]
return \( X_{\text{best}} \);
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 $X \in \mathcal{X}$.

$X_{best} \leftarrow X$; $searching \leftarrow true$

while ($searching$) do

$Y \leftarrow h_N(X)$;

if ($Y \neq \text{“fail”}$) then

$X \leftarrow Y$;

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

else $searching \leftarrow false$

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: Initially $T \leftarrow T_0$; 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 $\text{GENERICSIMULATEDANNEALING}(c_{\text{max}}, T_0, \alpha)$

\begin{verbatim}
c \leftarrow 0; T \leftarrow T_0;
Select a feasible solution $X \in \mathcal{X}$; $X_{\text{best}} \leftarrow X$;
while ($c \leq c_{\text{max}}$) do
    $Y \leftarrow h_N(X)$; // this is usually a randomized choice
    if ($Y \neq \text{"fail"}$) then
        if ($P(Y) > P(X)$) then
            $X \leftarrow Y$;
            if ($P(X) > P(X_{\text{best}})$) then $X_{\text{best}} \leftarrow X$;
        else
            $r \leftarrow \text{random}(0, 1)$;
            if ($r < e^{\frac{P(Y) - P(X)}{T}}$) then $X \leftarrow Y$;
    
c \leftarrow c + 1;
T \leftarrow \alpha T;
return $X_{\text{best}}$;
\end{verbatim}
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 forbid the application of $\text{CHANGE}(Y, X)$ for $L$ iterations ($L$ is the lifetime of the tabu list).

Example:

$\mathcal{X} = \{0, 1\}^n$, using $N_1(X) = \{Y \in \mathcal{X} : \text{dist}(X, Y) = 1\}$.

$X = [0100]$ and $Y = [0101]$, we have that $\text{CHANGE}(Y, X) = 4 =$ index of coordinate that was swapped.

Suppose $L = 2$.

<table>
<thead>
<tr>
<th>sequence of points:</th>
<th>[0100]</th>
<th>[0101]</th>
<th>[1101]</th>
<th>[1001]</th>
<th>[1011]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabu list:</td>
<td>4</td>
<td>4,1</td>
<td>1,2</td>
<td>2,3</td>
<td></td>
</tr>
</tbody>
</table>
So any sequence that cycles $X \rightarrow \ldots \rightarrow X$ has length at least $2L$. Choosing $L = 1$ is typical.

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

In absolute no circumstance implement \textbf{TabuList} as an array indexed by the number of iterations! Instead, implement \textbf{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;
- $\text{ CHANGE}(X, Y) \not\in \{ \text{ TabuList}[d] : c - L \leq d \leq c - 1 \}$;
- $P(Y)$ is maximum among all such feasible elements.
Algorithm **GenericTabuSearch**($c_{max}, L$)

1. $c \leftarrow 1$;
2. Select a feasible solution $X \in \mathcal{X}$.
3. $X_{best} \leftarrow X$;
4. while ($c \leq c_{max}$) do
   1. $N \leftarrow N(X) \setminus \{F : \text{change}(X, F) \in \text{Tabulist}[d],\]$\[ c - L \leq d \leq c - 1\}$; (typo corrected)
   2. for each ($Y \in N$) do
      1. if ($Y$ is infeasible) then $N \leftarrow N \setminus \{Y\}$;
      2. if ($N = \emptyset$) then return $X_{best}$;
   3. Find $Y \in N$ such that $P(Y)$ is maximum;
   4. $\text{Tabulist}[c] \leftarrow \text{change}(Y, X)$;
   5. $X \leftarrow Y$;
   6. if ($P(X) > P(X_{best})$) then $X_{best} \leftarrow X$;
   7. $c \leftarrow c + 1$;
   8. return $X_{best}$;

In the real algorithm, $\text{Tabulist}$ must be a queue of length $L$!!!

So, the operation

$\text{Tabulist}[c] \leftarrow \text{change}(Y, X)$;

must be implemented as:

$\text{Tabulist}.insert(\text{change}(Y, X))$; (only keeps last $L$ elements)

and the line: $N \leftarrow N(X) \setminus \{F : \text{change}(X, F) \in \text{Tabulist}[d], c - L \leq d \leq c - 1\}$

should be understood as:

$N \leftarrow N(X) \setminus \{F : \text{change}(X, F) \in \text{Tabulist}\}$;
4. Genetic Algorithms

More complex than neighbourhood search.
Fix a number $\text{PopSize}$ (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.

   Parents:
   \[
   \begin{array}{c}
   \text{Parents:} \\
   \text{j} \\
   \text{2 Children:}
   \end{array}
   \]

   Example: \( j = 3 \)
   Parents: \([110|1101001] \quad [100|1000101]\)
   Children: \([110|1000101] \quad [100|1101001]\)

2. Partially matched crossover (for permutations)

   Two crossover points: \( 1 \leq j < k \leq n \)

   Example: \( j = 3 \) and \( k = 6 \)
   \[\alpha = [3, 1, 4, 7, 6, 5, 2, 8] \quad \beta = [8, 6, 4, 3, 7, 1, 2, 5]\]

<table>
<thead>
<tr>
<th>swap</th>
<th>( \alpha )</th>
<th>( \beta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 \leftrightarrow 4</td>
<td>([3, 1, 4, 7, 6, 5, 2, 8])</td>
<td>([8, 6, 4, 3, 7, 1, 2, 5])</td>
</tr>
<tr>
<td>7 \leftrightarrow 3</td>
<td>([7, 1, 4, 3, 6, 5, 2, 8])</td>
<td>([8, 6, 4, 3, 7, 1, 2, 5])</td>
</tr>
<tr>
<td>6 \leftrightarrow 7</td>
<td>([6, 1, 4, 3, 7, 5, 2, 8])</td>
<td>([8, 7, 4, 6, 3, 1, 2, 5])</td>
</tr>
<tr>
<td>5 \leftrightarrow 1</td>
<td>([6, 5, 4, 3, 7, 1, 2, 8])</td>
<td>([8, 7, 4, 6, 3, 5, 2, 1])</td>
</tr>
</tbody>
</table>
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 \texttt{GENERICGENETICALGORITHM}(PopSize, c_{\text{max}})

\begin{verbatim}
c ← 1;
Select an initial population \( \mathcal{P} \) with PopSize feasible solutions;
for each \( X \in \mathcal{P} \) do \( X \leftarrow h_N(X) \); [mutation]
\( X_{\text{best}} \leftarrow \text{element in } \mathcal{P} \text{ with maximum profit}; \)
while \( (c \leq c_{\text{max}}) \) do
  Construct a pairing of the elements in \( \mathcal{P} \);
  \( \mathcal{Q} \leftarrow \mathcal{P} \)
  for each pair \( (W, X) \) in the pairing do
    \( (Y, Z) \leftarrow \text{rec}(W, X); \) [recombination/mating]
    \( Y \leftarrow h_N(Y); \) [mutation]
    \( Z \leftarrow h_N(Z); \) [mutation]
    \( \mathcal{Q} \leftarrow \mathcal{Q} \cup \{Y, Z\}; \)
  \end{verbatim}

Let \( \mathcal{P} \) be the best PopSize members of \( \mathcal{Q} \);
Let \( Y \) be the element in \( \mathcal{P} \) with maximum profit;
if \( (P(Y) > P(X_{\text{best}})) \) then \( X_{\text{best}} \leftarrow Y; \)
\( c \leftarrow c + 1; \)
return \( X_{\text{best}}; \)
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
$\sum_{i=0}^{n-1} w_i x_i \leq M$.

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

\begin{align*}
c & \leftarrow 0; T \leftarrow T_0; \\
X & \leftarrow [x_0, x_1, \ldots, x_{n-1}] = [0, 0, \ldots, 0]; \\
CurW & \leftarrow 0; X_{best} \leftarrow X; \\
\text{while } (c \leq c_{max}) \text{ do} & \\
& \quad j \leftarrow \text{randomInt}(0, n - 1); \\
& \quad Y \leftarrow X; \\
& \quad y_j \leftarrow 1 - x_j; \quad \text{(swap } j \text{ coordinate of } X) \\
& \quad \text{if } (y_j = 1) \text{ and } (curW + w_j > M) \text{ then } Y \leftarrow \text{fail}; \\
& \quad \text{if } (Y \neq \text{fail}) \text{ then} \\
& \quad \quad \text{if } (y_j = 1) \text{ then} \\
& \quad \quad \quad X \leftarrow Y; \\
& \quad \quad \quad curW \leftarrow curW + w_j; \\
& \quad \quad \quad \text{if } P(X) > P(X_{best}) \text{ then } X_{best} \leftarrow X; \\
& \quad \quad \text{else} \\
& \quad \quad \quad r \leftarrow \text{random}(0, 1); \\
& \quad \quad \quad \text{if } (r < e^{-p_j/T}) \text{ then} \\
& \quad \quad \quad \quad X \leftarrow Y; \\
& \quad \quad \quad \quad curW \leftarrow curW - w_j; \\
& \quad \quad \quad c \leftarrow c + 1; \\
& \quad \quad \quad T \leftarrow \alpha T; \\
& \quad \text{return } (X_{best});
\end{align*}
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 improvement 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 $\textsc{KnapsackTabuSearch}(c_{\text{max}}, L)$
\begin{algorithmic}
  \State $c \leftarrow 1$;
  \State Select a random feasible $X = [x_0, x_1, \ldots, x_{n-1}] \in \{0, 1\}^n$;
  \State $curW \leftarrow \sum_{i=0}^{n-1} x_i w_i$;
  \State $X_{\text{best}} \leftarrow X$;
  \While{$(c \leq c_{\text{max}})$}
    \State $N \leftarrow \{0, 1, \ldots, n - 1\}$;
    \State $\text{start} \leftarrow \max\{0, c - L\}$;
    \For{$j \leftarrow \text{start}$ \text{to} $c - 1$}
      \State $N \leftarrow N \setminus \{\text{Tabulist}[j]\}$;
      \For{each $(i \in N)$}
        \If{$(x_i = 0)$ and $(curW + w_i > M)$} \Then
          \State $N \leftarrow N \setminus \{i\}$;
        \EndIf
      \EndFor
      \If{$(N = \emptyset)$} \Then
        \State Exit;
      \EndIf
      \State Find $i \in N$ such that $(-1)^{x_i} p_i / w_i$ is maximum;
      \State $\text{Tabulist}[c] \leftarrow i$;
      \State $x_i \leftarrow 1 - x_i$; \hspace{1em} (swap $i$ coordinate)
      \If{$(x_i = 1)$} \Then
        \State $curW \leftarrow curW + w_i$;
      \Else
        \State $curW \leftarrow curW - w_i$;
      \EndIf
      \If{$P(X) > P(X_{\text{best}})$} \Then
        \State $X_{\text{best}} \leftarrow X$;
        \State $c \leftarrow c + 1$;
      \EndIf
    \EndFor
  \EndWhile
  \State return $X_{\text{best}}$;
\end{algorithmic}
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 $c : V \times V \rightarrow R$

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:

\[
G(X, i, j) = C(X) - C(X_{ij})
\]
\[
= c(x_i, x_{i+1}) + c(x_j, x_{j+1}) - c(x_{i+1}, x_{j+1}) - c(x_i, x_j)
\]

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

Algorithm \textsc{SteepestAscentTwoOpt}(X)

\[
done \leftarrow \text{false};
\]
\[
\text{while (not done) do}
\]
\[
done \leftarrow \text{true}; g_0 \leftarrow 0;
\]
\[
\text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do}
\]
\[
\quad \text{for } j \leftarrow i + 2 \text{ to } n - 1 \text{ do}
\]
\[
\quad g \leftarrow G(X, i, j);
\]
\[
\quad \text{if } (g > g_0) \text{ then}
\]
\[
\quad \quad g_0 \leftarrow G; i_0 \leftarrow i; j_0 \leftarrow j;
\]
\[
\quad \text{if } (g_0 > 0) \text{ then}
\]
\[
\quad X \leftarrow X_{i_0, j_0};
\]
\[
\quad done \leftarrow \text{false};
\]
Selecting the initial population

Randomly pick one and then mutate it:

Algorithm $\text{SELECT}(\text{popsize})$

for $i \leftarrow 0$ to $\text{popsize} - 1$ do
    $r \leftarrow \text{RANDOMINTEGER}(0, n! - 1)$;
    $P_i \leftarrow \text{PERMLEXUNRANK}(n, r)$;
    $\text{STEEPESTASCENTTWOOPT}(P_i)$;
return $[P_0, P_1, \ldots, P_{\text{popsize}-1}]$;

Two recombination algorithms

1. Partially Matched Crossover

Algorithm $\text{PMRec}(A, B)$

$h \leftarrow \text{RANDOMINTEGER}(10, n/2)$; (length of the substring)
$j \leftarrow \text{RANDOMINTEGER}(0, n - 1)$; (start of the substring)
$(C, D) \leftarrow \text{PARTIALLYMATCHEDCROSSOVER}(A, B, j, (h + j)mod n)$
$\text{STEEPESTASCENTTWOOPT}(C)$;
$\text{STEEPESTASCENTTWOOPT}(D)$;
return $(C, D)$;
2. Another Recombination Algorithm

Algorithm MGKRec(A, B)

\[ h \leftarrow \text{RANDOMINTEGER}(10, n/2); \] (length of the substring)
\[ j \leftarrow \text{RANDOMINTEGER}(0, n - 1); \] (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
\[ \text{if } A[j] \notin T \text{ then } D[i] \leftarrow A[j]; \]
\[ i \leftarrow i + 1; \]
\[ \text{STEEPESTASCENTTWOOPT}(D); \]
(Similarly build \( C \) swapping \( A \) and \( B \) roles:)
\[ j \leftarrow \text{RANDOMINTEGER}(0, n - 1); \] (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
\[ \text{if } B[j] \notin T \text{ then } C[i] \leftarrow B[j]; \]
\[ i \leftarrow i + 1; \]
\[ \text{STEEPESTASCENTTWOOPT}(C); \]
return \((C, D)\);
Genetic Algorithm for TSP

Algorithm \textsc{GeneticTSP}(\textit{popsize}, \textit{c}_{max})
\begin{align*}
c & \leftarrow 1; \\
[P_0, P_1, \ldots, P_{\text{popsize}-1}] & \leftarrow \textsc{Select}(\text{popsize}); \\
\text{Sort } P_0, P_1, \ldots, P_{\text{popsize}-1} \text{ in increasing order of cost.} \\
X_{\text{best}} & \leftarrow P_0; \\
\text{BestCost} & \leftarrow C(P_0); \\
\text{while } (c \leq c_{\text{max}}) \text{ do} \\
& \hspace{1em} \text{for } i \leftarrow 0 \text{ to } \text{popsize}/2 - 1 \text{ do} \\
& \hspace{2em} (P_{\text{popsize}+2i}, P_{\text{popsize}+2i+1}) \leftarrow \textsc{Rec} \ (P_{2i}, P_{2i+1}); \\
& \hspace{1em} \text{Sort } P_0, P_1, \ldots, P_{2\text{popsize}-1} \text{ in increasing order of cost.} \\
& \hspace{1em} \text{curCost} \leftarrow C(P_0); \\
& \hspace{1em} \text{if } (\text{curCost} < \text{BestCost}) \text{ then} \\
& \hspace{2em} X_{\text{best}} \leftarrow P_0; \\
& \hspace{2em} \text{BestCost} \leftarrow \text{curCost}; \\
& \hspace{1em} c \leftarrow c + 1; \\
& \hspace{1em} \text{return } X_{\text{best}};
\end{align*}

Note: \textsc{Rec} represents either of the two recombination algorithms.
Experimental results from Tables 5.7 and 5.8 (pages 186-187):