Unit 3: Computational Complexity

- Course contents:
  - Computational complexity
  - NP-completeness
  - General Purpose Combinational Optimizations
- Readings
  - Chapters 3, 4, and 5

<table>
<thead>
<tr>
<th>Time</th>
<th>Big-Oh</th>
<th>n = 10</th>
<th>n = 100</th>
<th>n = 10^3</th>
<th>n = 10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>O(1)</td>
<td>5 x 10^{-7} sec</td>
<td>5 x 10^{-7} sec</td>
<td>5 x 10^{-7} sec</td>
<td>5 x 10^{-7} sec</td>
</tr>
<tr>
<td>3n</td>
<td>O(n)</td>
<td>3 x 10^{-8} sec</td>
<td>3 x 10^{-7} sec</td>
<td>3 x 10^{-6} sec</td>
<td>0.003 sec</td>
</tr>
<tr>
<td>n log n</td>
<td>O(n log n)</td>
<td>3 x 10^{-8} sec</td>
<td>2 x 10^{-7} sec</td>
<td>3 x 10^{-6} sec</td>
<td>0.006 sec</td>
</tr>
<tr>
<td>n^2</td>
<td>O(n^2)</td>
<td>1 x 10^{-7} sec</td>
<td>1 x 10^{-5} sec</td>
<td>0.001 sec</td>
<td>16.7 min</td>
</tr>
<tr>
<td>n^3</td>
<td>O(n^3)</td>
<td>1 x 10^{-6} sec</td>
<td>0.001 sec</td>
<td>1 sec</td>
<td>3 x 10^5 cent.</td>
</tr>
<tr>
<td>2^n</td>
<td>O(2^n)</td>
<td>1 x 10^{-6} sec</td>
<td>3 x 10^17 cent.</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>n!</td>
<td>O(n!)</td>
<td>0.003 sec</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

O: Upper Bounding Function

- **Def:** \( f(n) = O(g(n)) \) if \( \exists c > 0 \) and \( n_0 > 0 \) such that \( 0 \leq f(n) \leq cg(n) \) for all \( n \geq n_0 \).
- Examples: \( 2n^2 + 3n = O(n^2) \), \( 2n^2 = O(n^3) \), \( 3n \log n = O(n^2) \)
- Intuition: \( f(n) \) “\( \leq \)” \( g(n) \) when we ignore constant multiples and small values of \( n \).
Big-O Notation

- How to show $O$ (Big-Oh) relationships?
  - $f(n) = O(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ for some $c \geq 0$.
- “An algorithm has worst-case running time $O(f(n))$”: there is a constant $c$ s.t. for every $n$ big enough, every execution on an input of size $n$ takes at most $cf(n)$ time.

Big-Theta Notation

- **Def**: $f(n) = \Theta(g(n))$ if $\exists c_1 > 0, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$.
  - Examples: $2n^2 + 3n = \Theta(n^2)$, $2n^2 = O(n^2)$, $3n \log n = O(n \log n)$
  - $g(n)$ is asymptotically tight bound of $f(n)$
### Computational Complexity

- **Computational complexity**: an abstract measure of the time and space necessary to execute an algorithm as functions of its “input size”.

- **Input size examples**:
  - sort $n$ words of bounded length $\Rightarrow n$
  - the input is the integer $n \Rightarrow \lg n$
  - the input is the graph $G(V, E) \Rightarrow |V|$ and $|E|$

- **Time complexity** is expressed in *elementary computational steps* (e.g., an addition, multiplication, pointer indirection).

- **Space Complexity** is expressed in *memory locations* (e.g. bits, bytes, words).

### Asymptotic Functions

- Polynomial-time complexity: $O(n^k)$, where $n$ is the **input size** and $k$ is a constant ($k = O(1)$).

- **Example polynomial functions**:
  - 999: constant
  - $\lg n$: logarithmic
  - $\sqrt{n}$: sublinear
  - $n$: linear
  - $n \lg n$: loglinear
  - $n^2$: quadratic
  - $n^3$: cubic

- **Example non-polynomial functions**
  - $2^n$, $3^n$: exponential
  - $n!$: factorial
Running-time Comparison

- Assumes 1000 MIPS (Yr: 200x), 1 instruction/operation

<table>
<thead>
<tr>
<th>Time</th>
<th>Big-Oh</th>
<th>$n = 10$</th>
<th>$n = 100$</th>
<th>$n = 10^3$</th>
<th>$n = 10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3n$</td>
<td>$O(1)$</td>
<td>$5 \times 10^{-7}$ sec</td>
<td>$5 \times 10^{-7}$ sec</td>
<td>$5 \times 10^{-7}$ sec</td>
<td>$5 \times 10^{-7}$ sec</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>$O(n \log n)$</td>
<td>$3 \times 10^{-8}$ sec</td>
<td>$2 \times 10^{-7}$ sec</td>
<td>$3 \times 10^{-6}$ sec</td>
<td>$0.006$ sec</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$O(n^2)$</td>
<td>$1 \times 10^{-7}$ sec</td>
<td>$1 \times 10^{-5}$ sec</td>
<td>$0.001$ sec</td>
<td>$16.7$ min</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$O(n^3)$</td>
<td>$1 \times 10^{-6}$ sec</td>
<td>$0.001$ sec</td>
<td>$1$ sec</td>
<td>$3 \times 10^5$ cent.</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$O(2^n)$</td>
<td>$1 \times 10^{-6}$ sec</td>
<td>$10^{17}$ cent.</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$n!$</td>
<td>$O(n!)$</td>
<td>$0.003$ sec</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Ch4. Tractable and Intractable Problems

- Tractable problems
  - Can be solved within polynomial time
- Intractable problems
  - Cannot be solved within polynomial time

- NP-complete problems
  - Likely to be intractable
  - Still under research...
Optimization Problems

- **Problem**: a general class, e.g., “the shortest-path problem for directed acyclic graphs.”
- **Instance**: a specific case of a problem, e.g., “the shortest-path problem in a specific graph, between two given vertices.”
- **Optimization problems**: those finding a legal configuration such that its cost is minimum (or maximum).
  - MST: Given a graph \(G=(V, E)\), find the cost of a minimum spanning tree of \(G\).
- An optimization problem \(\Pi\) has instance \(I=(F, c)\) where
  - \(F\) is the set of feasible solutions, and
  - \(c\) is a cost function, assigning a cost value to each feasible solution \(c: F \rightarrow \mathbb{R}\)
  - The solution of the optimization problem is the feasible solution with optimal (minimal/maximal) cost
- cf., **Optimal** solutions/costs, optimal (exact) algorithms (Attn: optimal ≠ exact in the theoretic computer science community).

The Traveling Salesman Problem (TSP)

- TSP: Given a set of cities and that distance between each pair of cities, find the distance of a “minimum tour” starts and ends at a given city and visits every city exactly once.
Decision Problem

- **Decision problems**: problem that can only be answered with “yes” or “no”
  - MST: Given a graph $G=(V, E)$ and a bound $K$, is there a spanning tree with a cost at most $K$?
  - TSP: Given a set of cities, distance between each pair of cities, and a bound $B$, is there a route that starts and ends at a given city, visits every city exactly once, and has total distance at most $B$?

- A decision problem $D_\Pi$ has instances: $I = (F, c, k)$
  - The set of instances for which the answer is “yes” is given by $Y_{\Pi}$.
  - A subtask of a decision problem is solution checking: given $f \in F$, checking whether the cost is less than $k$.

- Could apply binary search on decision problems to obtain solutions to optimization problems.
- NP-completeness is associated with decision problems.

A Decision Problem

- **The Circuit-Satisfiability Problem (Circuit-SAT)**:
  - **Instance**: A combinational circuit $C$ composed of AND, OR, and NOT gates.
  - **Question**: Is there an assignment of Boolean values to the inputs that makes the output of $C$ to be 1?

- A circuit is satisfiable if there exists a set of Boolean input values that makes the output of the circuit to be 1.
  - Circuit (a) is satisfiable since $\langle x_1, x_2, x_3 \rangle = \langle 1, 1, 0 \rangle$ makes the output to be 1.

- (b) unsatisfiable circuit
Complexity Class P

- **Complexity class P** contains those problems that can be **solved** in polynomial time in the **size of input**.
  - **Input size**: size of encoded “binary” strings.
  - Edmonds: Problems in P are considered **tractable**.
- The computer concerned is a **deterministic Turing machine**
  - **Deterministic** means that each step in a computation is predictable.
  - A **Turing machine** is a mathematical model of a universal computer (any computation that needs polynomial time on a Turing machine can also be performed in polynomial time on any other machine).
- MST is in $P$.

Complexity Class NP

- **Class NP (Nondeterministic Polynomial)**: class of problems that can be **verified** in polynomial time in the size of input.
  - NP: class of problems that can be solved in polynomial time on a nondeterministic machine.
- **Solution checking** can be done in polynomial time on a deterministic machine $\Rightarrow$ the problem can be solved in polynomial time on a **nondeterministic Turing machine**.
  - **Nondeterministic**: the machine makes a guess, e.g., the right one (or the machine evaluates all possibilities in parallel).
- Is TSP $\in$ NP?
  - Need to verify a solution in polynomial time.
    - Guess a tour.
    - Check if the tour visits every city exactly once.
    - Check if the tour returns to the start.
    - Check if total distance $\leq B$.
  - All can be done in $O(n)$ time, so TSP $\not\in$ NP.
Complexity Class NP-Complete

- Still unsettled issue:  
  \[ P \subset NP \text{ or } P = NP? \]
- There is a strong belief that \( P \neq NP \), due to the existence of \( NP \)-complete problems.
- The class **NP-complete (NPC)**:
  - All problems in NPC have the same degree of difficulty: Any NPC problem can be solved in polynomial time \( \Rightarrow \) all problems in \( NP \) can be solved in polynomial time.

NP-Complete and NP-hard

- **NP-completeness**: worst-case analyses for decision problems.
- A decision problem \( L \) is **NP-complete (NPC)** if
  1. \( L \in NP \), and
  2. \( L' \leq_p L \) for every \( L' \in NP \).
- **NP-hard**: If \( L \) satisfies property 2, but not necessarily property 1, we say that \( L \) is **NP-hard**.
- Suppose \( L \in NPC \).
  - If \( L \in P \), then there exists a polynomial-time algorithm for every \( L' \in NP \) (i.e., \( P = NP \)).
  - If \( L \notin P \), then there exists no polynomial-time algorithm for any \( L' \in NPC \) (i.e., \( P \neq NP \)).
Polynomial-time Reduction

- **Motivation:** Let $L_1$ and $L_2$ be two decision problems. Suppose algorithm $A_2$ can solve $L_2$. Can we use $A_2$ to solve $L_1$?

- **Polynomial-time reduction $f$ from $L_1$ to $L_2$:** $L_1 \leq_P L_2$
  - $f$ reduces input for $L_1$ into an input for $L_2$ s.t. the reduced input is a “yes” input for $L_2$ iff the original input is a “yes” input for $L_1$.
    - $L_1 \leq_P L_2$ if $\exists$ polynomial-time computable function $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ s.t. $x \in L_1$ iff $f(x) \in L_2$, $\forall x \in \{0, 1\}^*$.
    - $L_2$ is at least as hard as $L_1$.
  - $f$ is computable in polynomial time.

Significance of Reduction

- **Significance of $L_1 \leq_P L_2$:**
  - $\exists$ polynomial-time algorithm for $L_2$ $\Rightarrow$ $\exists$ polynomial-time algorithm for $L_1$ ($L_2 \in P \Rightarrow L_1 \in P$).
  - $\not\exists$ polynomial-time algorithm for $L_1$ $\Rightarrow$ $\not\exists$ polynomial-time algorithm for $L_2$ ($L_1 \not\in P \Rightarrow L_2 \not\in P$).

- $\leq_P$ is transitive, i.e., $L_1 \leq_P L_2$ and $L_2 \leq_P L_3 \Rightarrow L_1 \leq_P L_3$.
Example: HC ≤ₚ TSP

- The Hamiltonian Circuit Problem (HC)
  - **Instance:** an undirected graph \( G = (V, E) \).
  - **Question:** is there a cycle in \( G \) that includes every vertex exactly once?

- TSP (The Traveling Salesman Problem)

- How to show HC ≤ₚ TSP?
  1. Define a function \( f \) mapping any HC instance into a TSP instance, and show that \( f \) can be computed in polynomial time.
  2. Prove that \( G \) has an HC iff the reduced instance has a TSP tour with distance ≤ \( B \) (\( x \in \text{HC} \iff f(x) \in \text{TSP} \)).

HC ≤ₚ TSP: Step 1

1. Define a reduction function \( f \) for HC ≤ₚ TSP.
   - Given an arbitrary HC instance \( G = (V, E) \) with \( n \) vertices
     - Create a set of \( n \) cities labeled with names in \( V \).
     - Assign distance between \( u \) and \( v \)

   \[
   d(u, v) = \begin{cases} 
   1, & \text{if } (u, v) \in E, \\
   2, & \text{if } (u, v) \notin E.
   \end{cases}
   \]

   - Set bound \( B = n \).
   - \( f \) can be computed in \( O(V^2) \) time.
HC \leq_p TSP: Step 2

2. G has an HC iff the reduced instance has a TSP with distance \leq B.
   - \( x \in HC \Rightarrow f(x) \in TSP. 
     - Suppose the HC is \( h = <v_1, v_2, \ldots, v_n, v_1> \). Then, \( h \) is also a tour in the transformed TSP instance.
     - The distance of the tour \( h \) is \( n = B \) since there are \( n \) consecutive edges in \( E \), and so has distance 1 in \( f(x) \).
     - Thus, \( f(x) \in TSP (f(x) \) has a TSP tour with distance \leq B). 

[Diagram: HC instance \( \rightarrow \) TSP instance]

HC \leq_p TSP: Step 2 (cont’d)

2. G has an HC iff the reduced instance has a TSP with distance \leq B.
   - \( f(x) \in TSP \Rightarrow x \in HC. 
     - Suppose there is a TSP tour with distance \leq n = B. Let it be \( <v_1, v_2, \ldots, v_n, v_1> \).
     - Since distance of the tour \leq n and there are n edges in the TSP tour, the tour contains only edges in \( E \).
     - Thus, \( <v_1, v_2, \ldots, v_n, v_1> \) is a Hamiltonian cycle (\( x \in HC \)).

[Diagram: HC instance \( \rightarrow \) TSP instance]
Summary: Proving NP-Completeness

- Five steps for proving that $L$ is NP-complete:
  1. Prove $L \in NP$.
  2. Select a known NP-complete problem $L'$.
  3. Construct a reduction $f$ transforming every instance of $L'$ to an instance of $L$.
  4. Prove that $x \in L'$ iff $f(x) \in L$ for all $x \in \{0, 1\}^*$.
  5. Prove that $f$ is a polynomial-time transformation.

- We have shown that TSP is NP-complete (reducing from HC).

Ch 5. Optimization Algorithms

- Continuous optimization problems
  - Variables are real numbers
- Combinatorial optimization problems
  - Variables are discrete values
  - Useful in EDA
Coping with Optimization Problems

- **Exact solution (may not applicable to big problems)**
  - **Exhaustive search**
    - Feasible only when the problem size is small.
  - **Visit only part of search space**
    - E.g. Branch and bound, dynamic programming, integer linear programming.

- **Approximation algorithms**
  - Guarantee to be a fixed percentage away from the optimum.
  - No general purpose approx. algorithms
  - E.g., MST for the minimum Steiner tree problem.

- **Heuristics**
  - No guarantee of performance
  - E.g. Greedy algorithm, Local search, Tabu search, Simulated annealing (hill climbing), Genetic algorithms, etc.

Algorithmic Paradigms

- **Branch and bound**: A search technique with pruning.
- **Mathematical programming**: A system of solving an objective function under constraints.
- **Dynamic programming**: Partition a problem into a collection of sub-problems, the sub-problems are solved, and then the original problem is solved by combining the solutions. (sub-problems are NOT independent)
- **Divide and Conquer**: Partition problems into independent sub-problems.

- **Greedy**: Pick a locally optimal solution at each step.
- **Simulated annealing**: An adaptive, iterative, non-deterministic algorithm that allows “uphill” moves to escape from local optima.
- **Tabu search**: Similar to simulated annealing, but does not decrease the chance of “uphill” moves throughout the search.
- **Genetic algorithm**: A population of solutions is stored and allowed to evolve through successive generations via mutation, crossover, etc.
Exhaustive search

- **General principle**
  - Systematically assign values to unspecified variables
  - Until a single point in search space is identified, or
  - An implicit constraint makes it impossible to continue
    - backtracking
- **Example: TSP**
  - X means backtracking

Branch and Bound

- **General principle**
  - Estimate the cost lower bound
  - Kill partial solutions higher than the lowest cost
- **Example: TSP**
  - Use MST as cost lower bound
  - E.g. A → B → C → F = 22; MST of {CDEA} = 6
    - 22 + 8 > 27 ➔ Killed
Exhaustive Search vs. Branch and Bound

- TSP example

Backtracking/exhaustive search  72 nodes!

Branch and bound  only 27 nodes

Pseudo Code of B&B

```plaintext
float best_cost;
solution_element val[n], best_solution[n];

b.and.b(int k)
{
    float new_cost;
    if (k == n)
    {
        new_cost := cost(val);
        if (new_cost < best_cost)
        {
            best_cost := new_cost;
            best_solution := copy(val);
        }
    }
    else if (lower_bound_cost(val,k) >= best_cost)
    /* No action, node is killed. */
    else
    {
        for each (el ∈ allowed(val, k))
        {
            val[k] := el;
            b.and.b(k + 1);
        }
    }
}

main()
{
    best_cost := ∞;
    b.and.b(0);
    report(best_solution);
}
```

Disadvantage:
- Finding lower bound is not easy all the time
Greedy Algorithms

- General principle:
  - Pick a locally optimal solution at each step
- Greedy method does not guarantee performance
  - sometimes is correct: e.g. Prim’s algorithm for MST
  - sometimes is not correct: e.g. Nearest neighbor for TSP

Nearest Neighbor for TSP

1. pick and visit an initial point $p_0$;
2. $P \leftarrow p_0$;
3. $i \leftarrow 0$;
4. while there are unvisited points do
5. visit $p_i$’s closest unvisited point $p_{i+1}$;
6. $i \leftarrow i + 1$;
7. return to $p_0$ from $p_i$.

- Simple to implement and very efficient, but not optimal!
Dynamic Programming (DP) vs. Divide-and-Conquer

- Both solve problems by combining the solutions to subproblems.
- Divide-and-conquer algorithms
  - Partition a problem into **independent** subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
  - Inefficient if they solve the same subproblem more than once.
- Dynamic programming (DP)
  - Applicable when the subproblems are **not independent**.
  - DP solves each subproblem just once.

Dijkstra’s Shortest Path

- Reduce search space by dynamic programming
  - Shortest path from \( s \rightarrow v = s \rightarrow u \) plus \( u \rightarrow v \)
    - Since shortest path \( s \rightarrow u \) is already known (8), calculation is eliminated
    - Shortest path from \( s \rightarrow v \) is 8+1
Linear Programming

- **General principle**
  - Convert problems into the mathematic format
  - Canonical form of LP
    - \( AX \leq b \)
    - \( X \geq 0 \)

- **Integer Linear Programming (ILP)**
  - Variables are restricted to integers

- **0-1 ILP problems**
  - Solutions are restricted to 0, 1

- **Why ILP useful for EDA?**
  - ILP solvers are widely available
  - Problem independent

---

**Example 1: TSP**

- **Variables** \( x_1 \ldots x_{12} \)
  - \( x_i = 1 \) means the edge is traveled
  - \( x_i = 0 \) means the edge is not traveled

- **Minimize cost of travel**
  \[
  \sum_{i=1}^{12} w(e_i) x_i
  \]

- **Subject to constraints**
  - Every vertex has exactly two edges traveled

\[
\begin{align*}
  v_1 : x_1 + x_2 + x_3 + x_4 &= 2 \\
  v_2 : x_1 + x_5 + x_6 + x_7 &= 2 \\
  v_3 : x_2 + x_5 + x_8 + x_9 &= 2 \\
  v_4 : x_4 + x_7 + x_{10} + x_{11} &= 2 \\
  v_5 : x_3 + x_8 + x_{10} + x_{12} &= 2 \\
  v_6 : x_6 + x_9 + x_{11} + x_{12} &= 2
\end{align*}
\]
Example 1: TSP (cont’d)

• However, not enough constraints
  – Multiple disjoint tour

  \[ \{v_1, v_2, v_3\} \{v_4, v_5, v_6\} : x_3 + x_4 + x_6 + x_7 + x_8 + x_9 \geq 2 \]
  \[ \{v_1, v_3, v_5\} \{v_2, v_4, v_6\} : x_1 + x_4 + x_5 + x_9 + x_{10} + x_{12} \geq 2 \]
  \[ \{v_1, v_2, v_4\} \{v_3, v_5, v_6\} : x_2 + x_3 + x_5 + x_6 + x_{10} + x_{11} \geq 2 \]

• Add more constraints to avoid multiple disjoint tour

Approximation Algorithm
Spanning Tree vs. Steiner Tree

• **Manhattan distance**: If two points (nodes) are located at coordinates \((x_1, y_1)\) and \((x_2, y_2)\), the Manhattan distance between them is given by
  \[ d_{12} = |x_1 - x_2| + |y_1 - y_2| \]

• **Rectilinear spanning tree**: a spanning tree that connects its nodes using Manhattan paths (Fig. (b) below).

• **Steiner tree**: a tree that connects its nodes, and additional points (Steiner points) are permitted to used for the connections.

• The minimum rectilinear spanning tree problem is in P, while the minimum rectilinear Steiner tree (Fig. (c)) problem is NP-complete.
  – The spanning tree algorithm can be an approximation for the Steiner tree problem (at most 50% away from the optimum).
Local Search

• General principle
  – Search only the neighbors of the current solution
  – Move to the neighbor with lower cost than the current solution

• Problem
  – Can be trapped in local minimum

```c
local_search()
{
    struct feasible_solution f;
    set of struct feasible_solution G;

    f ← initial_solution();
    do {
        G ← \{\{g|g ∈ \mathcal{N}(f), c(g) < c(f)\};
        if (G ≠ ∅)
            f ← "any element of G";
    } while (G ≠ ∅);
    "report f";
}
```

Figure 5.8 The pseudo-code description of local search.

Simulated Annealing (SA)

• General Principle: mimic material cooling process
  – Search the neighbors of current solution
  – A good move means the cost decreases
    ■ always accepted
  – A bad move means the cost increases
    ■ Accepted with probability \( \exp(-\Delta c/T) \)

• Analogy
  – Energy = cost function
  – Temperature = controlling parameter T

• Advantage: can escape local minimum
  – ‘Uphill climbing’ is possible
**Pseudo Code of SA**

```c
int accept(struct feasible_solution f, g)
{
    float Δc;

    Δc ← c(g) − c(f);
    if (Δc ≤ 0)
        return 1;
    else return (e^{−Δc/T} > random(1));
}

simulated.annealing()
{
    struct feasible_solution f, g;
    float T;

    f ← initial.solution();
    do {
        do {
            g ← “some element of N(f)”;
            if (accept(f, g))
                f ← g
        while (!thermal.equilibrium());
    T ← new.temperature(T);
        while (!stop());
            “report f”;  
    }
}
```

---

**Tabu Search**

- General Principle: avoid staying in the ‘taboo’ solutions
  - Keep a list of visited solutions: Taboo list
  - Always move to a new solution other than the taboo list
    - even the new solution is poor than the current solution

```c
tabu.search()
{
    struct feasible_solution f, g, b;
    set of struct feasible_solution G;
    “k-element FIFO queue of” feasible_solution Q;

    Q ← “empty”;
    b ← initial.solution();
    f ← initial.solution();
    do {
        G ← “some subset of N(f) such that ∀ s ∈ Q, s ∉ G”;
        if (G ≠ ∅) {
            g ← “cheapest element of G”;
            “shift g into Q”;
            f ← g;
            if (c(f) < c(b))
                b ← f;
        }
    }
    while (G ≠ ∅ or stop());
```
Genetic Algorithms (GA)

• General Principle
  ⦿ Survival of the fittest
  ⦿ Keep a group of feasible solutions
    • ‘population’
  ⦿ ‘Parent’ population generates the ‘child’ population
    • Keep only the best children

Important steps in GA

• Cross over: Two feasible solutions generate their child by switching chromosomes

![Diagram of crossover process](image)

**Figure 5.11** The generation of a pair of children by crossover.

• Mutation: some chromosomes can change by probability
Pseudo Code of GA

```c
procedure genetic()
int pop_size;
set of struct chromosome pop, newpop;
struct chromosome parent1, parent2, child;
pop ← ∅;
for (i ← 1; i ≤ pop_size; i ← i + 1)
    pop ← pop ∪ ("chromosome of random feasible solution");
do {
    newpop ← ∅;
    for (i ← 1; i ≤ pop.size; i ← i + 1) {
        parent1 ← select(pop);
        parent2 ← select(pop);
        child ← crossover(parent1, parent2);
        newpop ← newpop ∪ {child};
    }
    pop ← newpop;
} while (!stop());
"report best solution";
```

Figure 5.12  The pseudo-code description of a genetic algorithm.

Example 2: Bin Packing

- **The Bin-Packing Problem** \( \Pi \): Items \( U = \{ u_1, u_2, \ldots, u_n \} \), where \( u_i \) is of an integer size \( s_i \); set \( B \) of bins, each with capacity \( b \).
- **Goal**: Pack all items, minimizing # of bins used. (NP-hard!)

```
\[
\begin{array}{cccccc}
\text{bin} & \text{size} = b \\
\text{Exp: } b = 6, & \{u_1, u_4, u_2, u_3, u_5\} \\
\text{optimal:} & 2 & 3 & 1 & 2 & 3 \end{array}
\]
```
Example 2: Bin Packing (cont’d)

- Greedy approximation alg.: First-Fit Decreasing (FFD)
  \[ FFD(\Pi) \leq 11 \text{OPT}(\Pi)/9 + 4 \]

- Use integer linear programming (ILP) to find a solution using \(|B|\) bins, then search for the smallest feasible \(|B|\).

Example 2: Bin Packing (cont’d)

- 0-1 variable: \(x_{ij}=1\) if item \(u_i\) is placed in bin \(b_j\), 0 otherwise.

\[
\begin{align*}
\text{max} & \quad \sum_{(i,j) \in E} w_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{i \in U} w_{ij} x_{ij} \leq b_j, \forall j \in B / \ast \text{capacity constraint} / (1) \\
& \quad \sum_{j \in B} x_{ij} = 1, \forall i \in U / \ast \text{assignment constraint} / (2) \\
& \quad \sum_{i} x_{ij} = n / \ast \text{completeness constraint} / (3) \\
& \quad x_{ij} \in \{0, 1\} / \ast \text{0, 1 constraint} / (4)
\end{align*}
\]

- **Step 1**: Set \(|B|\) to the lower bound of the \# of bins.
- **Step 2**: Use the ILP to find a feasible solution.
- **Step 3**: If the solution exists, the \# of bins required is \(|B|\). Then exit.
- **Step 4**: Otherwise, set \(|B| \leftarrow |B| + 1\). Goto Step 2.