2025 Fall CSE5025

Combinatorial Optimization

组合优化

Instructor: 刘晟材

Lecture 5-2: Integer Programming for Combinatorial Optimization – Part II

Agenda for Today's Lecture

In this lecture, we will focus on

- The core idea of Branch and Cut (B&C)
- Constructive Heuristics
- Local Search
- Metaheuristics

Learning objectives for this lecture

- Know the performance bottleneck of B&B
- Know the basic idea of B&C
- Master the differences between different types of heuristics

Recap: The B&B Algorithm

Notations:

- Z_{IP}: the optimal value of the original IP (the one we want to solve).
- Z_{LP}: the optimal value of the LP relaxation of the original IP.
- Z_{INC}: the value of the incumbent solution (the best-so-far integer solution)
- Z_{LP}^{P}: the optimal value of the LP relaxation at node P in the B&B tree

We always have (for maximization problem):

- $Z_{INC} \leq Z_{IP} \leq Z_{LP}$
- $Z_{LP}^{P} \leq Z_{LP}$

Recap: The B&B Algorithm

In our last lecture, we saw that Branch & Bound is an intelligent search.

Its efficiency crucially depends on one thing: **Pruning** (剪枝).

- We maintain an incumbent (the best-so-far integer solution) Z_{INC}.
- At each node P, we solve the LP relaxation to get its optimal value Z_{LP}^{P}, which is an optimistic estimate of any integer solution than can be **obtained in this entire branch (any subproblem created from node *P*).**
- The pruning (by bound) rule (for maximization problem):

$$\text{IF } Z_{LP}^{P} \leq Z_{INC} \text{ THEN Prune}$$

Why is B&B Sometimes Inefficient?

The Problem: **A “Weak” or “Loose” Bound.** The pruning rule works if Z_{LP} is close to the true integer optimum Z_{IP}.

The Integrality Gap (IG): The gap between the LP relaxation and the true integer solution at the root node:

- (for minimization): $Z_{IP} - Z_{LP}$
- (for maximization): $Z_{LP} - Z_{IP}$
- the ratio definition Z_{LP}/Z_{IP} is also used in the literature

A **large IG** means the initial bound is very loose.

The Consequence of a Weak Bound

The core issue is that Z_{LP}^{P} must drop low enough (for maximization problem) to compete with the Incumbent Z_{INC}.

The Bound Monotonicity: As the B&B tree goes deeper, Z_{LP}^{P} is non-increasing because we are only adding more constraints (branching).

$$Z_{LP}^{P} \leq \ Z_{LP}^{parent\ node\ of\ P}$$

The Challenge of a Loose Root:

- Suppose $Z_{IP} = 100$ and $Z_{INC} = 95$
- If the root bound $Z_{LP} = 1000$ (a loose bound), the entire tree starts high.
- For the pruning rule $Z_{LP}^{P} \leq Z_{INC} = 95$ to activate, the B&B algorithm has to spend enormous resources (branching) to bring Z_{LP}^{P} down from the "ceiling" (1000) to a range (≤95) where effective pruning can finally occur. Without a tight initial bound, the B&B engine is severely hampered.

The Core Idea: Cutting Planes

A cutting plane (割平面), or “cut”, is a special constraint that we **add to our** LP.

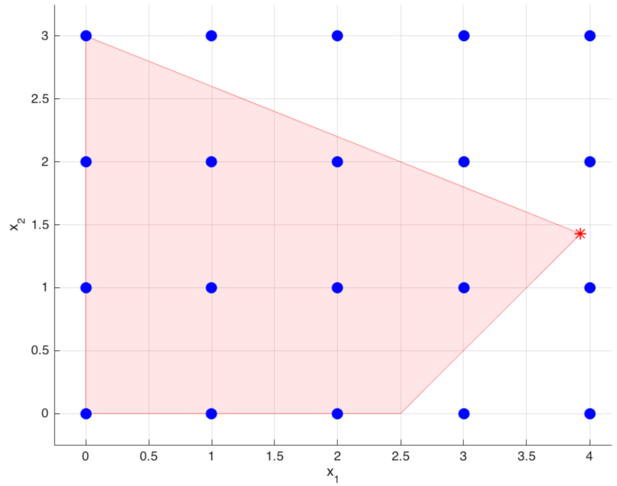
A “cut” is a **valid inequality** with two properties:

- It is **satisfied by all feasible integer solutions**. That is to say, it never cuts off the true integer optimum Z_{IP}.
- It is **violated by the current fractional solution** x^*. In other words, it always cuts off the current “bad” optimal solution of the LP).

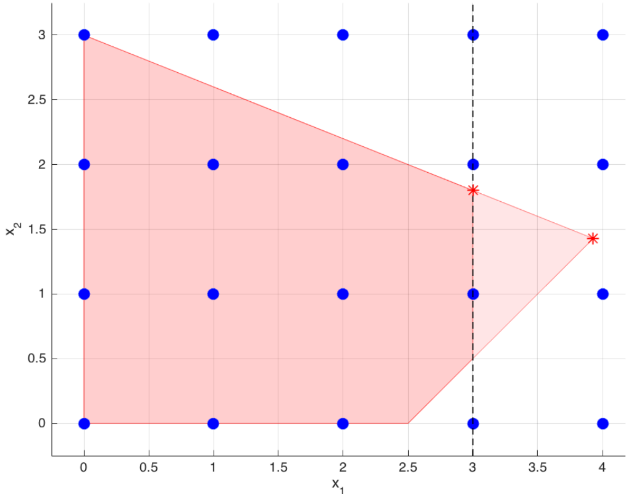
The new LP will have **a smaller feasible region** than the previous LP and hence has **a tighter bound**.

The process of finding and adding these cuts is called “**separation**”.

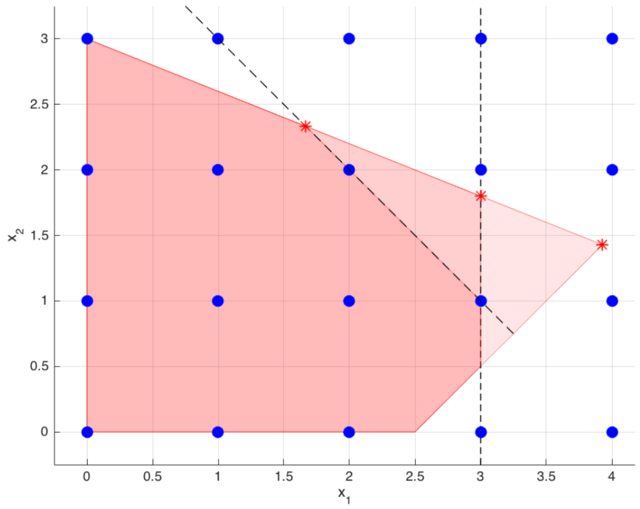
Visualizing a Cutting Plane



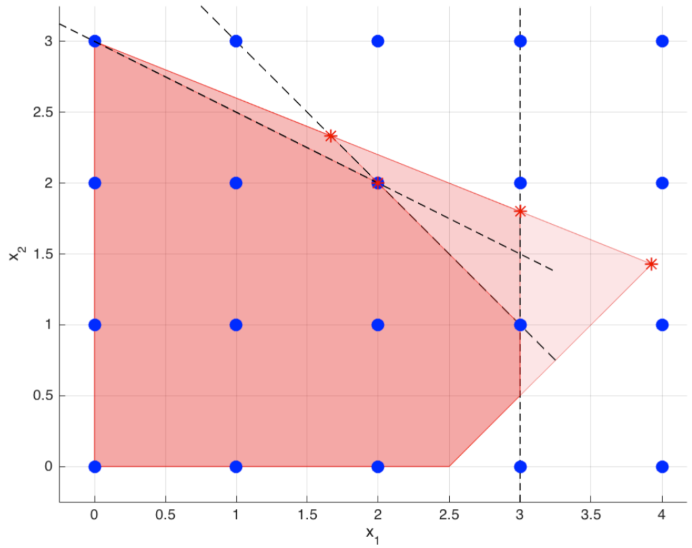
(1)



(2)



(3)



(4)

Branch & Cut

B&C **improves the bound at each node**, which in turn **improves pruning**.

Weak Bound (Original B&B):

- In this scenario, the root node Z_{LP} was very loose, and the bound at node P **is still too loose**.
- $Z_{LP}^{P} = 10 \quad \ldots \quad Z_{IP} = 100 \quad \ldots \quad Z_{INC} = 110$
- $Z_{LP}^{P} \ngeq Z_{INC} \rightarrow$ No Pruning.

An example (minimization problem)

Tightened Bound (Branch & Cut):

- We add cuts...
- $Z_{LP}^{P,\prime} = 50 \quad \ldots \quad Z_{IP} = 100 \quad \ldots \quad Z_{INC} = 110$
- $Z_{LP}^{P,\prime} \ngeq Z_{INC} \rightarrow$ No Pruning.
- We add more cuts...
- $Z_{LP}^{P,\prime\prime} = 105 \quad \ldots \quad Z_{IP} = 100 \quad \ldots \quad Z_{INC} = 110$
- $Z_{LP}^{P,\prime\prime} \ngeq Z_{INC} \rightarrow$ No Pruning.
- We add even more cuts...
- $Z_{LP}^{P,\prime\prime\prime} = 115 \quad \ldots \quad Z_{IP} = 100 \quad \ldots \quad Z_{INC} = 110$
- $Z_{LP}^{P,\prime\prime\prime} \geq Z_{INC} \rightarrow$ **PRUNE!**

Algorithmic Steps of B&C

B&C integrates the "cut loop" inside the B&B algorithm.

1) Initialize a "global incumbent" $Z_{INC} = -\infty$.

2) Add the root LP relaxation to a "**set**" of problems.

3) While **set** is not empty:

a. Select a problem P from the set.

b. Solve its LP relaxation, get Z_{LP}^{P} and $\boldsymbol{x}^*$.

c. Pruning 1 (by Bound): If $Z_{LP}^{P} \leq Z_{INC}$, prune this branch, and goto step 3).

d. Pruning 2 (Infeasible): If $Z_{LP}^{P} = -\infty$, prune this branch and goto step 3).

e. If $\boldsymbol{x}^*$ are all integers, we have found a feasible integer solution. Check if $Z_{LP}^{P} > Z_{INC}$ and update the incumbent Z_{INC} accordingly. Goto Step 3).

Algorithmic Steps of B&C

f. **Cut (the new part)**: If $\boldsymbol{x}^*$ is fractional.

- Enter the “Cut Loop”:
  - **Find Cuts**: Run “separation algorithms” to find valid cuts that x^* violates.
  - **If Cuts Found**: Add them to the LP and go back to Step 3-b (Solve).
  - **If No Cuts Found**: Exit the Cut Loop.

e. Branch**: If we exit the Cut loop and $\boldsymbol{x}^*$ is still fractional**, then we create two new subproblems with added constraints, respectively. Add both to the set. Goto Step 3).

4) Return Z_{INC}.

B&C Applied to the DFJ formulation of TSP

The DFJ Formulation (Dantzig-Fulkerson-Johnson) is a strong IP model for TSP, providing a very tight LP relaxation bound.

- Model Strength: The LP Relaxation (Z_{LP}) of the full DFJ formulation is extremely tight, meaning Z_{LP} is very close to Z_{IP}. This provides excellent pruning potential.
- The Constraint Issue: DFJ achieves this strength by including Subtour Elimination Constraints (SECs). There are $O(2^n)$ such constraints:

$$\sum_{i\in S}\sum_{j\notin S} x_{ij} \geq 1 \text{ for all } S \subseteq V, S \neq \emptyset, S \neq V$$

The Paradox: The strongest model is **computationally difficult** (impossible) to load into the solver upfront.

The Full DFJ Formulation of TSP

$$\text{Minimize } Z = \sum_{i=1}^{n} \sum_{j \neq i} c_{ij} x_{ij}$$

$$\text{s.t.} \quad \sum_{i \neq j} x_{ij} = 1 \ \text{ for each } j = 1, \ldots, n$$

$$\sum_{j \neq i} x_{ij} = 1 \ \text{ for each } i = 1, \ldots, n$$

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 1 \ \text{ for all } S \subseteq V, S \neq \emptyset, S \neq V$$

$$x_{ij} \in \{0,1\} \text{ for all } i \neq j$$

The Core Idea: Lazy Constraint Generation

B&C resolves this paradox by treating the $O(2^n)$ constraints as “Lazy Constraints” that are generated on demand.

Algorithmic steps **at a node**:

1) **Start**: The LP contains the basic assignment constraints (enter/leave once) and only those SECs inherited from its parent, and a branching constraint.

2) **Solve LP**: The solver returns a fractional solution x^* (which contains subtours).

3) **Separate (Cut Loop)**: We run an efficient **separation algorithm** (e.g., min-cut) on x^* to find a violated SEC (a subtour S), e.g., for a subtour $1 \to 2 \to 1$, we identify $S = \{1, 2\}$.

4) **Add Cut**: We add only the one violated constraint for that S back to the LP model.

5) **Re-Solve & Repeat**: The solver is forced to find a new, valid fractional solution x^{**} that avoids S. This loop continues until no more violated SECs can be found.

Summary: B&C

Pure B&B is slow if LP relaxation is "weak". B&C = B&B + Cutting Planes.

How does it work?

- At each node P, it solves the LP, finds violated cuts, adds those cuts, and re-solves. This tightens the bound Z_{LP}^{P}

Why is it efficient?

- A tighter bound → more pruning.
- It allows us to use "strong" but exponentially large formulations (like TSP's DFJ) in a computationally feasible way. Note for TSP, more powerful cuts than the SECs (cuts) are used.

B&C is the core technology inside modern IP solvers (Gurobi, CPLEX) and specialized solvers (Concorde for TSP).

Lecture 6-1: Heuristics and Metaheuristics for Combinatorial Optimization

Beyond Exact Algorithms

Recap: We just learned B&C. It can solve TSP optimally.

The Reality:

- B&C works for TSP up to thousands, but it takes massive computing power.
- For many other NP-hard problems, exact methods fail at $n = 50$ or $n = 100$.

Real-World Needs:

- Scale: What if we have 1000000 cities?
- Time: What if we need an answer in seconds, not days?
- Complexity: What if the constraints are messy and hard to model in IP?

The **Trade-off**: We sacrifice Optimality for Speed. We seek a “Good Enough” solution in “Reasonable Time”.

Terminology Check

Approximation Algorithm (Previous Lectures):

- A polynomial-time algorithm with a provable performance guarantee. e.g., Christofides for Metric TSP: always within $1.5 \times OPT$.
- **Focus: Worst-case analysis.**

Heuristic (This Lecture):

- An algorithm designed to find good solutions empirically.
- **No provable guarantee** on solution quality (in the general case).
- It might find the optimal solution, or it might be far off.
- **Focus: Average-case empirical performance and practical speed.**

Metaheuristic (This lecture):

- A general-purpose high-level framework or strategy to guide heuristic search.

Taxonomy of Heuristics

Constructive Heuristics:

- Start from an empty solution.
- Build the solution piece by piece (greedily).
- **Pros**: Extremely fast, easy to implement.
- **Cons**: Often lower quality, decisions are myopic.

Improvement Heuristics (Local Search):

- Start from a complete solution.
- Iteratively make small changes to improve it.
- **Pros**: Better quality than constructive methods.
- **Cons**: Susceptible to getting trapped in Local Optima.

Recap: TSP Constructive Heuristics

Nearest Neighbor (NN):

- Logic: Always travel to the closest unvisited city.
- Weakness: Forced to take very long edges at the end to close the tour.

Cheapest Insertion (CI):

- Logic: Find the unvisited city k and the specific position (i, j) in the current tour that minimizes the increase in tour cost.
- Strength: Directly optimizes the objective function at each step.

Farthest Insertion (FI):

- Logic: Start with a subtour. Insert the node that is farthest from the current tour to minimize cost increase.
- Strength: Establishes the general shape of the tour early

Constructive Example: Graph Coloring

Problem: Assign a color to each vertex of a graph such that no adjacent vertices share the same color, minimizing the total number of colors.

The "Welsh-Powell" Heuristic:

- Calculate Degrees: Compute the degree of every vertex.
- Sort: Order vertices by degree in descending order.
  - Rationale: High-degree nodes are the hardest to color.
  - Coloring Loop:
    - Assign the first available color (Color 1) to the head of the list.
    - Traverse the list: Color any non-adjacent uncolored node with Color 1.
    - Move to Color 2, repeat until all nodes are colored.

Improvement Heuristics: Local Search

Concept: Instead of building a solution from scratch, we improve or repair an **existing one**.

The General Algorithm:

1. **Initialization:** Generate an initial solution S (random or constructive).
2. **Evaluation:** Calculate the cost objective $f(S)$.
3. **Neighborhood Search:** Examine solutions S' that are "near" S.
4. **Move:** If a neighbor S' exists such that $f(S') < f(S)$:
   - Update current solution: $S \leftarrow S'$.
   - Repeat step 3.
5. **Termination:** If no neighbor is better, STOP.
   - The current solution S is a **Local Optimum**.

Defining the Neighborhood Structure

The definition of “Neighborhood” $N(S)$ defines the search landscape. $N(S)$ is the set of solutions reachable from S by a move operator.

Common Neighborhoods for TSP:

- 2-Opt: Delete 2 edges and reconnect the path (reverse a tour segment). Size: $O(n^2)$.
- Swap: Exchange the positions of two cities. Size: $O(n^2)$.
- Relocate (Insert): Move a city to a new position in the tour. Size: $O(n^2)$.

Common Neighborhoods for Binary Problems (e.g., Knapsack):

- Bit Flip: Change x_i from 0 to 1 (or 1 to 0). Size: $O(n)$.
- Pair Swap: Exchange a selected item with an unselected one. Size: $O(n^2)$.

Illustration of 2-OPT

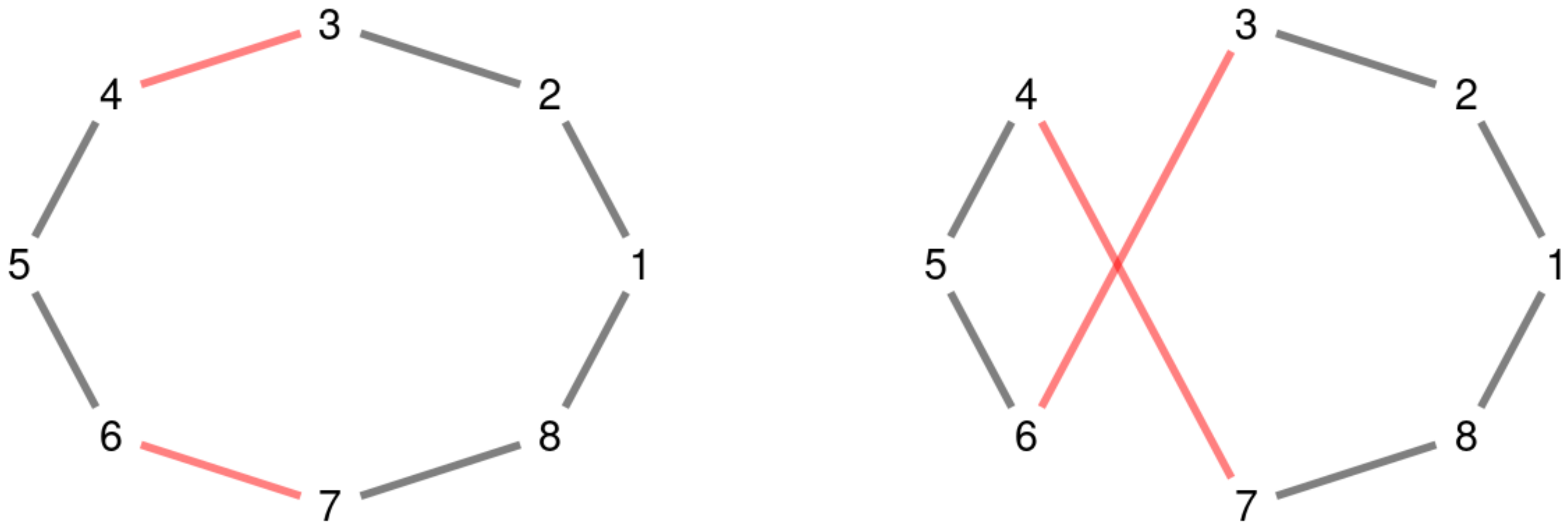
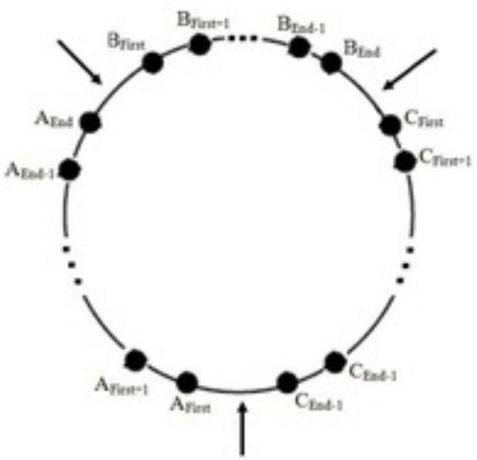
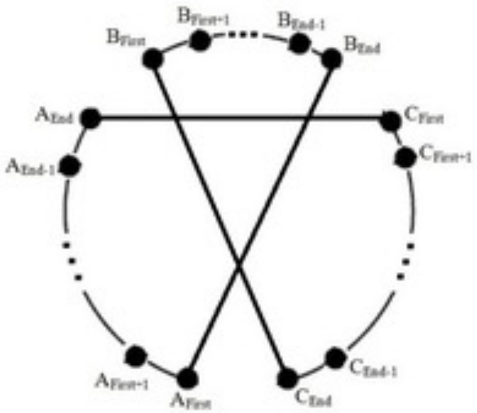


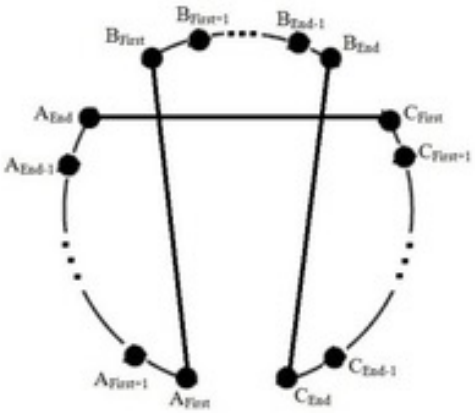
Illustration of 3-OPT



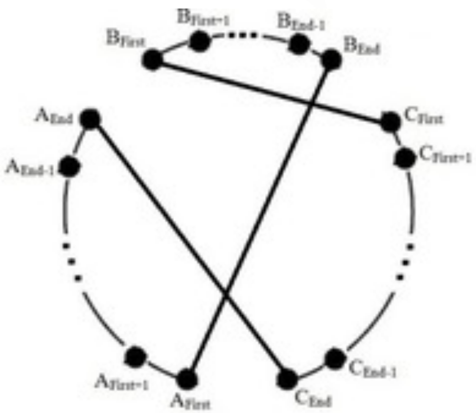
Original tour : ABC



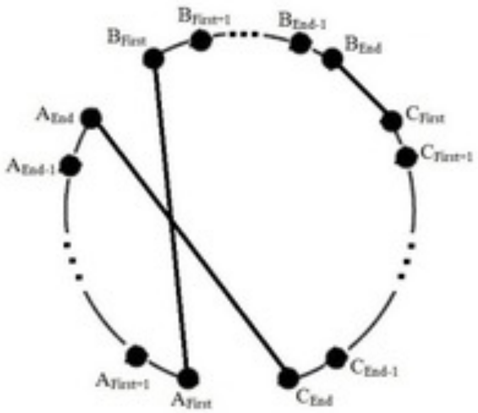
Method 1. ACB



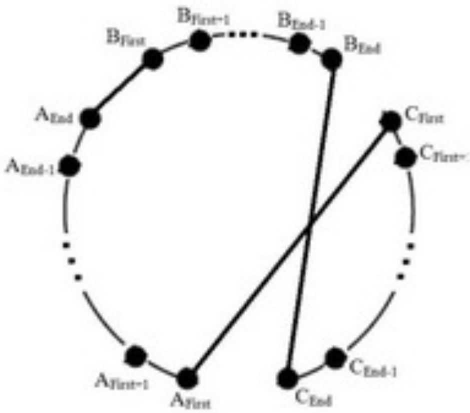
Method 2. ACB^{-1}



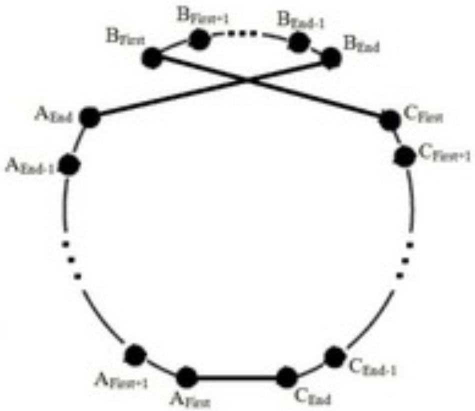
Method 3. AC^{-1}B



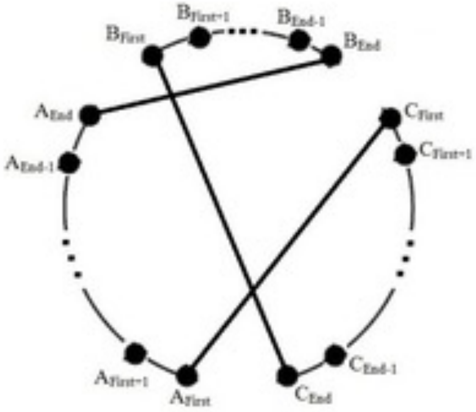
Method 4. AC^{-1}B^{-1}



Method 5. ABC^{-1}



Method 6. AB^{-1}C



Method 7. AB^{-1}C^{-1}

Search Strategy 1: First Improvement

Definition: Scan neighbors in $N(S)$ sequentially or randomly and accept the first one found that improves the objective function.

Algorithm:

1. Order the neighbors in $N(S)$ (e.g., random permutation).
2. Evaluate neighbor S'.
3. If $f(S') < f(S)$, move to S' immediately.
4. Restart search from the new solution.

Pros & Cons:

- **Pro**: Very fast per iteration. Ideal for large neighborhoods.
- **Con**: The improvement gain at each step might be small.

Search Strategy 2: Best Improvement

Definition: Evaluate **ALL** neighbors in $N(S)$ and move to the one that yields the maximum improvement in objective function (e.g., reduction in cost).

Pros & Cons:

- **Pro**: Maximizes the gain at every single step; Deterministic path for a given starting point
- **Con**: Very slow per iteration if $|N(S)|$ is large (e.g., $O(n^3)$).

Algorithm:

1. Set `best_neighbor` = null, `min_cost` = $f(S)$ (Current Cost).
2. **For every** neighbor S' in $N(S)$:
   - If $f(S') < min\_cost$:
     - Update `min_cost` = $f(S')$.
     - Update `best_neighbor` = S'.
3. **Move:**
   - **If** `best_neighbor` is not null (Improvement found):
     - Update $S \leftarrow$ `best_neighbor`.
   - **Else** (No neighbor is better):
     - **STOP**. Current S is a Local Optimum.

Comparison of Two Improvements

The choice determines the **trade-off between speed per step and gain per step**.

Feature	First Improvement (First Accept)	Best Improvement (Steepest Descent)
Move Acceptance	Moves to the **first** neighbor S' found such that $f(S') < f(S)$.	Moves to the neighbor S' that yields the **maximum** improvement in the entire neighborhood.
Iteration Behavior	**Stops scanning** the current neighborhood immediately once an improving move is found.	**Must scan the entire neighborhood** $N(S)$ completely before making a move.
Computational Cost	Low cost per iteration (often just a few checks).	High cost per iteration (requires $
Convergence Path	Takes many small steps. Path depends on check order (stochastic).	Takes fewer, larger steps. Path is deterministic.

When to Choose Which Strategy

The optimal choice depends on the neighborhood size and time budget.

- Prioritize first improvement for large neighborhoods
  - If your move operator (e.g., 3-Opt for TSP) generates $O(n^3)$ or more neighbors, running best improvement becomes prohibitively slow.
  - Randomization is important: To prevent the search from following a predictable, low-quality path, the order of checking neighbors in first improvement should be randomized at the start of each new iteration.
- Use best improvement for solution refinement on small neighborhoods.
- The Time-to-Quality Curve (Empirical):
  - While best improvement takes fewer steps, first improvement often achieves a high-quality solution faster in terms of wall-clock time.

Impact of Neighborhood Size

Small Neighborhoods (e.g., 2-Opt, 1-Flip):

- Size: Polynomial, usually low degree ($O(n)$, $O(n^2)$)
- Speed: Fast to evaluate each step.
- Landscape: "Rugged" with many small valleys.
- Result: The search gets stuck in local optima very easily.

Large Neighborhoods (e.g., 3-Opt):

- Size: Larger polynomial $O(n^3)$ or exponential.
- Speed: Slow to evaluate.
- Landscape: "Smoother". The move allows jumping over small hills.
- Result: Finds higher quality local optima but is computationally expensive.

Implementation Tip: Delta Evaluation

Naive Evaluation:

- Recalculate the objective function $f(S)$ from scratch after every move.
- Cost: $O(n)$ per neighbor.

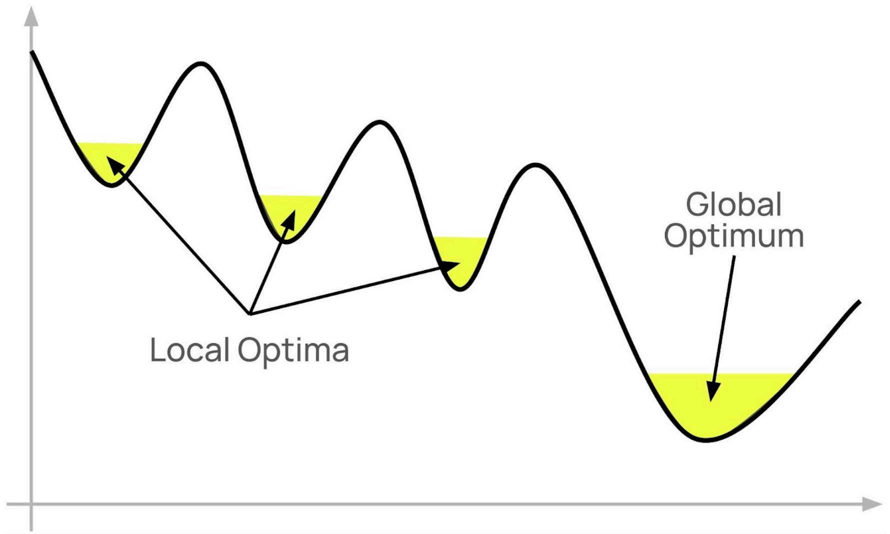
Delta (Δ) Evaluation:

- Calculate only the change caused by the move.
- Example (TSP 2-Opt):
  - Remove edges (A,B) and (C,D). Add (A,C) and (B,D).
  - $\Delta = dist(A, C) + dist(B, D) - dist(A, B) - dist(C, D)$\$.
- Cost: $O(1)$ per neighbor.

Impact: This speeds up Local Search by a factor of n. This optimization is critical for solving large problem instances.

The Fundamental Problem: Local Optima

Definition: A solution S^* is a Local Optimum if $f(S^*) \leq f(S')$ for all neighbors $S' \in N(S^*)$.



The Trap:

- Local Search algorithms are Greedy.
- They only accept improving moves.
- Once they hit a local optimum (a "valley"), they stop.
- They cannot temporarily climb "uphill" to reach a deeper valley (global optimum) located elsewhere in the search space.
- To solve this, we need **Metaheuristics**.

Metaheuristics: Overview

Definition: A high-level strategy that guides other heuristics to search for solutions beyond local optimality.

Key Feature: They allow **non-improving moves** (deterioration) to escape local optima.

The **Core Duality**:

- Exploitation (Intensification):
  - Searching carefully around good solutions to refine them.
  - Mechanism: local search (Descent).
- Exploration (Diversification):
  - Visiting entirely new regions of the search space.
  - Mechanism: Random moves, large jumps, restarts.

Classification of Metaheuristics

Trajectory-Based (a single solution)

- The algorithm maintains and modifies a **single solution** at a time.
- It traces a “trajectory” or path through the search space.
- **Focus: Depth-oriented. Good at Exploitation.**
- **Examples: Simulated Annealing (模拟退火), Tabu Search (禁忌搜索).**

Population-Based (A set/population of solutions)

- The algorithm maintains a set (population) of solutions.
- Solutions interact, combine, or compete to create new solutions.
- Focus: Breadth-oriented. Good at Exploration.
- **Examples: Genetic Algorithms (遗传算法).**

Trajectory Method: Simulated Annealing (SA)

Inspiration: Physical annealing in metallurgy. Heating metal and cooling it slowly allows atoms to settle into a strong, low-energy crystal structure.

Analogy:

- **Energy**: Objective Function Cost $f(S)$.
- **Temperature**: A control parameter determining the probability of accepting bad moves.

Core Idea:

- To escape local optima, we must occasionally accept uphill (worse) moves.
- The probability of accepting a worse move should depend on the size of the deterioration and the current “Temperature”.

Simulated Annealing: The Algorithm

1. Initialize solution S and Temperature T.
2. **While** $T > T_{min}$:
3. **Repeat** L times (Inner Loop):
   - Pick a random neighbor $S' \in N(S)$.
   - Calculate $\Delta = f(S') - f(S)$.
   - **If** $\Delta < 0$ **(Improvement):**
     - Always accept: $S \leftarrow S'$.
   - **If** $\Delta \geq 0$ **(Worsening):**
     - Accept with probability $P = e^{-\Delta/T}$.
     - (Generate random $r \in [0, 1]$. If $r < P$, move to S').
4. **Cool Down:** Decrease T (e.g., $T \leftarrow 0.99 \cdot T$).

SA: The Logic of temperature

The Metropolis Criterion: $P(\text{accept}) = e^{-\Delta/\mathrm{T}}$

High Temperature ($\mathrm{T} \to \infty$):

- $e^{-\Delta/\mathrm{T}}$ is close to 1.
- The algorithm accepts almost all moves, even terrible ones.
- Behavior: random walk (pure exploration).

Low Temperature ($\mathrm{T} \to 0$)

- $e^{-\Delta/\mathrm{T}}$ is close to 0.
- The algorithm rejects almost all worsening moves.
- Behavior: local search / hill climbing (pure exploitation).

Strategy: Start with high exploration to find the right region, then transition to high exploitation to refine the solution.

Trajectory Method: Tabu Search (TS)

Philosophy: "Intelligent" search using **Memory** rather than randomness.

Core Mechanism:

1) Generate all (or many) neighbors of the current solution S.

2) Move to the best neighbor S', even if it is worse than S.

- This automatically allows climbing out of local optima.

The Cycling Problem:

- If we move $S \rightarrow S'$ (worse), the best neighbor of S' is usually S (the previous optimum).
- The search will cycle $S \rightarrow S' \rightarrow S$ indefinitely
- Solution: We need a **Tabu List**.

Tabu Search: Memory Structures

Short-Term Memory (The Tabu List):

- Records recent moves or solution attributes.
- **Rule**: A move is **Tabu** (forbidden) if it is in the list.
- Effect: Forces the search to be away from the current local optimum (prevents cycling).

Aspiration Criteria (渴望准则、特赦准则):

- A rule to override the Tabu status.
- Rule: If a move is **Tabu**, but it produces a solution better than the global best found so far, allow it anyway.

Long-Term Memory:

- Frequency-based: Track how often edges/nodes are used.
- Penalize frequently used features to force the search into unexplored regions.

Population-based Method: Genetic Algorithms (GA)

Inspiration: Darwinian Natural Selection.

Key Difference from single solution-based methods: operates on a **population of solutions**, not just one.

Terminology:

- Chromosome (染色体): Representation of a solution (e.g., permutation [1, 4, 2, 3]).
- Fitness (适应度): The objective function value (quality of the solution).
- Generation (代): One iteration of the algorithm.

Hypothesis: By combining the “genetic material” (sub-structures) of two high-quality parent solutions, we may create an offspring solution that inherits the strengths of both.

The GA

1. **Initialization:** Generate a random population of N individuals.
2. **Evaluation:** Compute Fitness for every individual.
3. **Selection:** Choose "parents" for the next generation.
   - Biased towards better individuals (Survival of the Fittest).
4. **Crossover (Recombination):** Combine Parent A and Parent B to create a Child.
   - This is the **primary** search operator.
5. **Mutation:** Randomly tweak the child (e.g., flip a bit, swap cities).
   - Maintains diversity and prevents premature convergence.
6. **Replacement:** Select survivors to form the next generation.

GA: Selection Strategies

Goal: Select parents such that better individuals have a higher chance, but weak ones are not totally excluded (to maintain diversity).

Roulette Wheel Selection (轮盘赌选择):

- Probability of selecting S_i is proportional to its fitness: $P(S_i) = f(S_i)/\sum f(S_j)$
- **Risk**: One super-individual can dominate early, reducing diversity.

Tournament Selection (锦标赛选择):

- Randomly pick k individuals from the population.
- Select the best one among the k to be a parent.
- Tunable: Large k = High pressure (Exploitation). Small k = Low pressure (Diversity or Exploration).

GA: Crossover Operators (交叉算子)

Binary Crossover (e.g., Knapsack):

- Parent A: 11111 | 00000
- Parent B: 00000 | 11111
- **One-Point Crossover**: Cut at a random line, swap tails.
- Child: 1111111111 (Inherits first half of A, second half of B).

Permutation Crossover (e.g., TSP):

- **Challenge**: Standard crossover creates invalid tours (duplicates/missing cities).
- **Ordered Crossover (OX1):**
  - Copy a random segment from Parent A to the Child.
  - Fill remaining slots with cities from Parent B; **crucially, fill them in the order they appear in Parent B, skipping cities already in the segment.**
- This preserves relative ordering information.

GA: Mutation & Elitism

Mutation (变异):

- Role: diversity maintenance.
- Without mutation, GA can only shuffle existing genes. If a necessary gene is missing from the entire population, crossover cannot recover it.
- Mechanism: Randomly modify the child with low probability.
- TSP Example: Randomly swap two cities in the tour.

Elitism (精英):

- Problem: Crossover and Mutation are destructive. We might accidentally destroy the best solution found so far.
- Mechanism: Always copy the top $k\%$ of individuals from the current generation directly to the next generation, unchanged.
- Guarantees that the best solution quality never decreases.

Hybridization: Memetic Algorithms

The complementary strengths:

- GA: Excellent at global exploration. They find the "right hills" but are slow at climbing to the exact peak.
- Local Search: Excellent at local exploitation. It climbs peaks very fast but gets stuck on the wrong hill.

Memetic Algorithm (GA + Local Search):

- Strategy: Apply local search to some solutions generated by crossover/mutation:

  1) Child = Crossover(Parent A, Parent B).

  2) Child = Mutation(Child).

  3) Child = Local_Search(Child).

Memetic algorithms are the state-of-the-art for many combinatorial problems.

Summary: Choosing a Method

Constructive Heuristics:

- Use for initialization or extremely time-constrained applications.

Local Search:

- Use to refine constructive/random solutions.
- Implement with Delta Evaluation for speed.

Metaheuristics:

- Simulated Annealing: best for problems with messy constraints. Easy to implement.
- Tabu Search: excellent for routing (TSP, VRP) and scheduling.
- Genetic Algorithms and Memetic Algorithms: The typical high-performance choice for hard problems, combining evolutionary diversity with local optimization.