PART II. OPTIMIZATION: NUMERICAL APPROACHES
(LECTURE 5)

Shpilev Petr Valerievich
Faculty of Mathematics and Mechanics, SPbU

September, 2025

Санкт-Петербургский государственный университет

Comments

In this lecture, we focus on discrete optimization methods and metaheuristics. We begin with the cutting plane method, presenting its algorithmic structure and illustrating it with examples. We then introduce the branch-and-bound framework for mixed-integer programming (MIP), exploring its geometric meaning, single branching steps, and a full algorithmic description.

Next, we turn to dynamic programming, showing how recursive formulations provide efficient solutions to problems such as the Padovan sequence and the classical 0-1 knapsack problem, highlighting why DP outperforms brute-force enumeration. The second part of the lecture is devoted to nature-inspired heuristics, with an emphasis on ant colony optimization (ACO).

We explain edge attractiveness, transition probabilities, and pheromone updates, demonstrating how shortest paths emerge collectively. The lecture concludes with the main loop of ACO, its detailed components, practical applications, and a discussion of greedy heuristics as simple yet effective alternatives.

Algorithm 22: The cutting plane method (Part 1)

Algorithm 22: The cutting plane method

```
# fractional-part helper: returns frac(x) = x - floor(x)
frac(x) = modf(x)[1]
# MIP(Mixed-Integer Program) is a mutable structure with fields A, b, c, D.
function cutting_plane(MIP)
    # Build LP relaxation from the MIP (keep A, b, c)
    LP = relax(MIP)                  # LP = LinearProgram(MIP.A, MIP.b, MIP.c)

  # Solve LP to optimality at a vertex; also return basic/nonbasic index sets
    x, b_inds, v_inds = minimize_lp(LP)
    n_orig = length(x)                    # remember original design length
    D = copy(MIP.D)                       # discrete index set

    # Repeat while some required discrete indices are non-integer
    while !all(isint(x[i]) for i in D)
        # Partition constraint matrix according to the current basis
        AB, AV = LP.A[:, b_inds], LP.A[:, v_inds]  # A_B columns, A_V columns
        Abar = AB \ AV        # A_bar = A_B^{-1} * A_V  (tableau coefficients)
        b = 0                 # counter for cuts generated in this iteration
```


Comments

This is the first part of Algorithm twenty-two, implementing the cutting plane method for mixed-integer programs. We begin with the helper function frac, which returns the fractional part of a real number, that is, the number minus its floor. The input MIP is a mutable structure containing matrix capital A, vector b, vector c, and the index set script D, which lists all variables that must take integer values.

The algorithm starts by building the linear programming relaxation of the MIP. That means we keep the same A, b, and c, but remove the integrality restrictions. The relax function returns a standard LP object. We then solve this LP to optimality using minimize underscore lp, which also gives us the sets of basic and nonbasic variable indices.

We store the solution vector x, and the number of original variables, so that later we can discard any extra slack variables. We also make a copy of the discrete index set D. The while-loop checks if all variables in D are integer; if not, we continue. Inside the loop we partition the LP's constraint matrix into A_B, the columns of basic variables, and A_V, the columns of nonbasic variables. Then we compute $\bar{A}$ as A_B inverse times A_V — these are the tableau coefficients. Finally, we initialize the counter b to zero, to count how many cuts we generate in this iteration.

Algorithm 22: The cutting plane method (Part 2)

```
        for i in D        # Loop over all discrete indices to construct individual CP
            if !isint(x[i])                  # If current discrete var is fractional
                b += 1                                          # Increment cut counter
      # Augment constraint matrix with one new row and one new column(slack variable)
                A2 = [LP.A zeros(size(LP.A,1));          # A with an extra zero column
                           zeros(1,size(LP.A,2)+1)]          # New bottom row of zeros
                A2[end,end] = 1              # Set coefficient of slack variable to 1
               # Fill row's nonbasic variable coefficients using floor(A_bar) - A_bar
                A2[end,v_inds] = (x->floor(x) - x).(Abar[b,:])
              # Augment right-hand side with new cut bound: - fractional part of x[i]
                b2 = vcat(LP.b, -frac(x[i]))
         # Augment cost vector with 0 for slack variable (it has no objective impact)
                c2 = vcat(LP.c, 0)
                   # Build updated LP including new cut constraint and slack variable
                LP = LinearProgram(A2,b2,c2)
            end
        end
              # After adding all cuts for this iteration, resolve LP to get new basis
        x, b_inds, v_inds = minimize_lp(LP)
    end
    return x[1:n_orig]           # Return the original variables (discard slack vars)
end
```


Comments

In the second part of the algorithm we actually construct the new constraints — the cutting planes — and insert them into the LP. The for-loop goes through every index in the discrete set D. If the corresponding component of x is fractional, we increment the cut counter b. We then create an augmented constraint matrix A_2: it is the original LP's A with an extra column for a new slack variable and an extra bottom row for the new cut. The bottom row is initially all zeros except the last entry, corresponding to the slack variable, which is set to one.

The nonbasic variable coefficients in the new row are computed from the b-th row of $\bar{A}$ as floor of $\bar{A}$ minus $\bar{A}$. This corresponds exactly to the theoretical cut formula. The right-hand side b_2 is the original LP's b with one extra entry: minus the fractional part of x_i. The cost vector c_2 is similarly extended by zero for the slack variable, since the cut has no cost. We then build a new LP with A_2, b_2, and c_2, which includes the cut.

After processing all fractional discrete variables we solve the updated LP, get a new basis, and continue the while-loop. When all discrete variables are integer, we return only the first n_{orig} entries of x, ignoring all the auxiliary slack variables that were added.

Example: Cutting Plane Method (Part 1)

Given integer program: $\min\limits_{x \in \mathbb{Z}^3} 2x_1 + x_2 + 3x_3$

$$\begin{bmatrix} 0.5 & -0.5 & 1.0 \\ 2.0 & 0.5 & -1.5 \end{bmatrix} x = \begin{bmatrix} 2.5 \\ -1.5 \end{bmatrix}, \quad x \geq 0$$

- LP relaxation solution: $x^* \approx [0.818,\ 0,\ 2.091]$
- Basis partition:

$$A_B = \begin{bmatrix} 0.5 & 1 \\ 2 & -1.5 \end{bmatrix}, \quad A_V = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}$$

- Tableau coefficients:

$$\bar{A} = A_B^{-1} A_V = \begin{bmatrix} -0.091 \\ -0.455 \end{bmatrix}$$

Cut for x_1 with slack x_4:

$$x_4 + (\lfloor -0.091 \rfloor - (-0.091))x_2 = \lfloor 0.818 \rfloor - 0.818$$

$$x_4 - 0.909x_2 = -0.818$$

Comments

In this example we apply the cutting plane method to a small three-variable integer program. The objective is to minimize $2x_1 + x_2 + 3x_3$, subject to two linear equations and nonnegativity constraints, with all variables being integers. We first solve the LP relaxation, which gives a fractional solution: x_1 is approximately 0.818, $x_2 = 0$, and x_3 is approximately 2.091.

From the current basis, we identify the basic variables and construct the matrices A_B and A_V. Computing $\bar{A}$ as the inverse of A_B times A_V gives us the tableau coefficients for the nonbasic variables. Since x_1 is fractional, we derive a cut using the general formula: introduce a slack variable x_4, subtract the integer and fractional parts as in the theory, and obtain the equation $x_4 - 0.909x_2 = -0.818$. This cut excludes the current fractional solution but keeps all integer-feasible points.

Example: Cutting Plane Method (Part 2)

Cut for x_3 with slack x_5:

$$x_5 + (\lfloor -0.455 \rfloor - (-0.455))x_2 = \lfloor 2.091 \rfloor - 2.091$$

$$x_5 - 0.545x_2 = -0.091$$

Modified integer program:

$$A = \begin{bmatrix} 0.5 & -0.5 & 1 & 0 & 0 \\ 2 & 0.5 & -1.5 & 0 & 0 \\ 0 & -0.909 & 0 & 1 & 0 \\ 0 & -0.545 & 0 & 0 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 2.5 \\ -1.5 \\ -0.818 \\ -0.091 \end{bmatrix}, \quad c = \begin{bmatrix} 2 \\ 1 \\ 3 \\ 0 \\ 0 \end{bmatrix}$$

- Solving LP gives: $x^* \approx [0.9, 0.9, 2.5, 0.0, 0.4]$
- Still fractional $\Rightarrow$ repeat with new cuts:

$$x_6 - 0.9x_4 = -0.9, \qquad x_7 - 0.9x_4 = -0.9$$
$$x_8 - 0.5x_4 = -0.5, \qquad x_9 - 0.4x_4 = -0.4$$

- Final integer solution: $x^* = [1, 2, 3]$

Comments

We now construct the cut for the other fractional basic variable, x_3. Following the same process, we introduce slack variable x_5, subtract integer and fractional parts, and obtain the equation $x_5 - 0.545x_2 = -0.091$.

Adding both cuts to the system, we form the modified integer program with expanded matrix A, right-hand side b, and cost vector c. Solving the LP relaxation of this augmented problem gives the new fractional solution: x_1 approximately 0.9, x_2 approximately 0.9, $x_3 = 2.5$, and nonzero slack variables. Since the solution is still fractional, we repeat the procedure, adding four more cuts involving new slack variables x_6 through x_9.

After solving the LP with all these cuts, we reach the integer-optimal solution x equals $[1, 2, 3]$.

This example shows how the cutting plane method progressively shrinks the feasible region until only integer-feasible points remain.

Branch and Bound: Introduction

Idea: Systematically explore subsets of the feasible region using branching and bounding to find the global optimum without enumerating all solutions.

- **Branch:** Choose a fractional variable and split into subproblems:
$$x_i \leq \lfloor x_i^* \rfloor \quad \text{or} \quad x_i \geq \lceil x_i^* \rceil$$
- **Bound:** Solve LP relaxation for each subproblem to obtain a bound on the objective value.
- **Prune:** Discard subproblems if infeasible or if their bound is worse than the best integer solution found so far.

Algorithm structure

- Start from LP relaxation of the original problem.
- Maintain a **priority queue** of active subproblems, ordered by bound.
- Iteratively:
  - Remove best-bound subproblem.
  - If integer-feasible, update **incumbent** (the current solution).
  - Else branch on a fractional variable and enqueue resulting subproblems with their bounds.
- Stop when the queue is empty; incumbent is optimal.

Comments

The branch and bound method is a way to solve problems where some or all variables must take integer values, such as integer programming problems. It avoids checking all possible combinations by systematically splitting the problem into smaller parts and discarding those that cannot contain the best solution.

The method starts by removing the requirement for variables to be integers and solving this easier "relaxed" problem. The answer from this relaxed problem gives us a best possible score that any solution in this part of the search space could achieve. This number is called a bound. If the relaxed problem already gives integer values for all variables that should be integers, then we have found a valid solution, and we check if it is better than the best one we have so far.

If some variable is not an integer, we split the problem into two new subproblems. In one subproblem, we force this variable to be less than or equal to its integer part. In the other, we force it to be greater than or equal to the smallest integer above it. This is the branching step.

We solve both new subproblems in their relaxed form and get bounds for them. If a bound shows that a subproblem cannot give a better answer than the best integer solution we already have, we discard it. We keep the remaining subproblems in a list and repeat the process: take one subproblem, branch if needed, calculate bounds, and discard hopeless ones.

The process ends when no subproblems are left. The best integer solution found by then is guaranteed to be the overall best.

In modern commercial solvers, the branch and bound method is often combined with the cutting plane method. Cutting planes are added at certain stages of the solution process to tighten the relaxed problems and bring their bounds closer to the actual optimal value. This allows for faster pruning of non-promising subproblems, reducing the branching depth and the total number of steps required to find the optimal integer solution.

Example: Single Branching Step

Given: Relaxed solution $x_c^* = [3,\ 2.4,\ 1.2,\ 5.8]$ for an integer program with $c = [-1,\ -2,\ -3,\ -4]$.

- Objective value of relaxed solution:

$$y_c = c^T x_c^* = -34.6$$

- Non-integer variables: $x_2 = 2.4$, $x_3 = 1.2$, $x_4 = 5.8$.
- Choose the most fractional (farthest from an integer): x_2 (distance 0.4).

Branching constraints

Two subproblems:

$$\text{Left branch:} \quad x_2 \le \lfloor 2.4 \rfloor = 2$$

$$\text{Right branch:} \quad x_2 \ge \lceil 2.4 \rceil = 3$$

- Each branch is solved in relaxed form to obtain a bound.
- These bounds are compared to the best integer solution found so far; branches with worse bounds are discarded.
- Geometrically, branching adds a hyperplane $x_2 = 2.5$ that splits the feasible region into two parts.

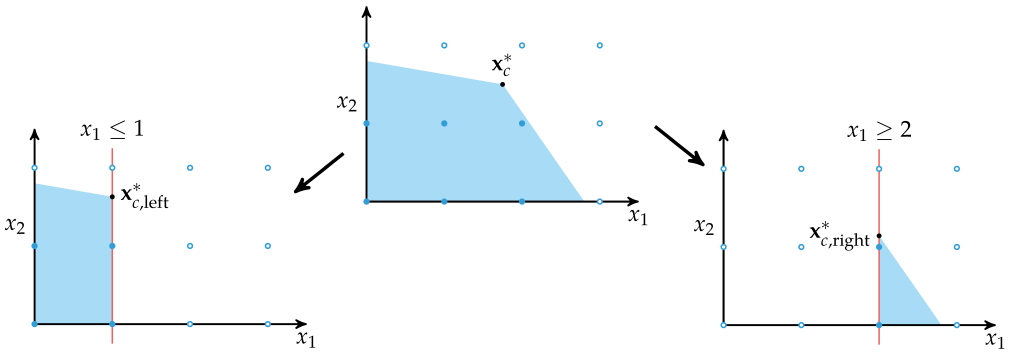
Comments

In this example, we have a relaxed solution with the vector $[3,\ 2.4,\ 1.2,\ 5.8]$ and an objective value equal to -34.6. The goal is to decide how to branch in order to move toward integrality. Several variables are fractional, but we pick the one that is farthest from an integer value — in this case, x_2 with a fractional distance of 0.4.

We create two new subproblems. In the first, we restrict x_2 to be less than or equal to 2, which is the integer part of its current value. In the second, we restrict x_2 to be greater than or equal to 3, which is the smallest integer greater than its current value. These two cases cover all possibilities for x_2 while forcing it closer to being an integer.

Each new subproblem is then solved without the integrality restriction — in other words, as a linear program — to obtain a bound on how good its best possible solution could be. If a bound is already worse than the best integer solution we have found so far, we discard that branch immediately. This prevents wasting time on unpromising regions.

From a geometric perspective, the branching step introduces a hyperplane at $x_2 = 2.5$ that cuts the feasible region into two disjoint parts. Each part is then processed separately in the branch and bound procedure.



Interpretation: Branching adds an integer inequality constraint that splits the feasible region into smaller subsets.

Comments

This figure illustrates the geometric idea behind the branching step in the branch and bound method. We begin with the feasible region of the linear programming relaxation, which contains both integer-feasible and fractional solutions. The current fractional solution, x^*_c, lies somewhere inside this region.

We choose one variable that has a fractional value and decide to branch on it. Branching means adding an extra integer inequality constraint to split the feasible region into two smaller parts. One part contains all solutions where this variable is less than or equal to the integer part of its current value. The other part contains all solutions where the variable is greater than or equal to the smallest integer above its current value.

Geometrically, the added constraint corresponds to a hyperplane that cuts through the feasible region. In the two-dimensional illustration, this is shown as a straight line dividing the polygon-shaped feasible set. The two resulting subsets, sometimes called left and right branches, are smaller feasible regions that exclude the current fractional point.

After branching, each subset becomes a separate linear programming relaxation. These are solved to get their bound values, which indicate how good a solution from that subset could be. Subsets with bounds worse than the best integer solution already found are discarded. Those with promising bounds remain in the process and may themselves be branched further.

This visual makes clear that branching systematically reduces the size of the search space, focusing the algorithm's effort only where better integer solutions might be found.

Example: Branch and Bound

Problem: $\min\limits_{x\in\mathbb{Z}^3} 2x_1 + x_2 + 3x_3, \quad \begin{bmatrix} 0.5 & -0.5 & 1 \\ 2 & 0.5 & -1.5 \end{bmatrix} x = \begin{bmatrix} 2.5 \\ -1.5 \end{bmatrix}, \; x \geq 0.$

- LP relaxation: $x_c^* \approx [0.818,\ 0,\ 2.09]$, $y_c = c^T x_c^* = 7.909$.
- Branch on x_1: create two subproblems $x_1 \leq 0$ and $x_1 \geq 1$.

LPs for the two branches (with a slack x_4)

$$A_{\text{left}} = \begin{bmatrix} 0.5 & -0.5 & 1 & 0 \\ 2 & 0.5 & -1.5 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}, \; b_{\text{left}} = \begin{bmatrix} 2.5 \\ -1.5 \\ 0 \end{bmatrix}, \; c = \begin{bmatrix} 2 \\ 1 \\ 3 \\ 0 \end{bmatrix}$$

$$A_{\text{right}} = \begin{bmatrix} 0.5 & -0.5 & 1 & 0 \\ 2 & 0.5 & -1.5 & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix}, \; b_{\text{right}} = \begin{bmatrix} 2.5 \\ -1.5 \\ 1 \end{bmatrix}, \; c = \begin{bmatrix} 2 \\ 1 \\ 3 \\ 0 \end{bmatrix}$$

- Left branch $x_1 \leq 0$: LP is infeasible.
- Right branch $x_1 \geq 1$: relaxed solution $[1,\ 2,\ 3,\ 0]$ is already integral.
- **Answer:** $x^* = [1,\ 2,\ 3]$, $f(x^*) = 2 \cdot 1 + 1 \cdot 2 + 3 \cdot 3 = 13$.

Comments

We reuse the same small integer program as earlier: minimize $2x_1 + x_2 + 3x_3$, with two equality constraints and nonnegativity. First, we drop integrality and solve the linear relaxation. The relaxed solution is approximately $[0.818,\ 0,\ 2.09]$, with objective value 7.909. This value is a lower bound for minimization: no feasible integer point in this region can have an objective smaller than 7.909.

To push the solution toward integrality, we branch on x_1 We create two subproblems. The left subproblem enforces $x_1 \leq 0$. In equality form with a nonnegative slack, the added row is $x_1 + x_4 = 0$ The right subproblem enforces $x_1 \geq 1$, written as $x_1 - x_4 = 1$. Each subproblem inherits the original two equalities and the same objective vector.

We resolve the two linear relaxations. The left branch is infeasible: the equalities together with $x_1 \leq 0$ and nonnegativity leave no feasible point. The right branch has the relaxed solution $[1, 2, 3, 0]$. Because the first three components are already integers and satisfy nonnegativity and the equalities, this vector is an integer-feasible solution of the original problem. Its objective equals $2 \cdot 1 + 1 \cdot 2 + 3 \cdot 3$, which is thirteen. Since the other branch is infeasible and there are no better candidates, we stop with the integer-optimal solution $x^* = [1,\ 2,\ 3]$, objective value thirteen.

Note: Earlier we illustrated cutting planes on this same model; here you can see that a single branching step also reaches the integer optimum.

Algorithm 23: Branch and Bound for MIP

```
function minimize_lp_and_y(LP)
    try
        x = minimize_lp(LP);
        return (x, x · LP.c)    # solve LP, return x and y = x·c
    catch
        return (fill(NaN, length(LP.c)), Inf) # infeasible: NaNs for x, +∞ objective
    end
end
function branch_and_bound(MIP)
    LP = relax(MIP); x, y = minimize_lp_and_y(LP)    # LP relaxation, then solve it
    n = length(x)                                    # number of decision variables
    x_best,y_best,Q = deepcopy(x),Inf,PriorityQueue() # incumbent, best value, queue
    enqueue!(Q, (LP,x,y), y)                 # push root node keyed by y (best-first)
    while !isempty(Q)
        LP, x, y = dequeue!(Q)                       # pop most promising subproblem
        if any(isnan.(x)) || all(isint(x[i]) for i in MIP.D) # infeasible or integer
            if y < y_best;
               x_best, y_best = x[1:n], y  # update incumbent
            end
        else  # pick most fractional index
            i = argmax([abs(x[i] - round(x[i])) for i in MIP.D])
```


Comments

Begin with a helper that solves a linear program and returns both the primal vector and its objective value. The function `minimize_lp_and_y` calls the LP solver and then computes $x \cdot c$, which is the objective. If the LP is infeasible or errors out, the function returns a vector of "Not a Number" entries together with "positive infinity." That pairing is a clean signal to the caller that the node is dead.

The main routine forms the linear relaxation of the mixed integer model by dropping integrality. Solving that relaxation gives a candidate vector and its value. The variable n stores the number of decision variables. The algorithm maintains an incumbent, noted as x_{best}, and its value y_{best}, initialized to infinity. It also uses a priority queue keyed by the objective value y, so the next processed node is the most promising for a minimization problem.

The loop removes the top entry from the queue, yielding a linear program, its solution, and its value. A first test filters out hopeless nodes. If any entry of x is "Not a Number," the node corresponds to an infeasible relaxation and can be discarded. If all discrete indices in the set D are already integers, then the node holds a valid mixed integer solution. In that case, if y is strictly less than the current best value, the incumbent is updated to the first n entries of x, and y_{best} is updated to y. Otherwise, the vector is fractional on at least one discrete coordinate. To branch, compute the distance to the nearest integer for each index in D, and choose the index with the largest absolute distance; this is i, the most fractional variable.

Algorithm 23: Branch and Bound for MIP (Part 2)

```
            # Left branch: x_i ≤ floor(x_i)
            A, b, c = LP.A, LP.b, LP.c
            A2 = [A zeros(size(A,1));
                  [j==i for j in 1:size(A,2)]' 1]   # new constraint row + slack
            b2, c2 = vcat(b, floor(x[i])), vcat(c, 0)
            LP2 = LinearProgram(A2,b2,c2)
            x2, y2 = minimize_lp_and_y(LP2)
            if y2 ≤ y_best
                enqueue!(Q, (LP2,x2,y2), y2)
            end
            # Right branch: x_i ≥ ceil(x_i)
            A2 = [A zeros(size(A,1));
                  [j==i for j in 1:size(A,2)]' -1]  # new constraint row + slack
            b2, c2 = vcat(b, ceil(x[i])), vcat(c, 0)# vcat "glues" vectors by elements
            LP2 = LinearProgram(A2,b2,c2)
            x2, y2 = minimize_lp_and_y(LP2)
            if y2 ≤ y_best
                enqueue!(Q, (LP2,x2,y2), y2)
            end
        end
    end
    return x_best
end
```


Comments

When the branching index i is chosen, two subproblems are created. The left branch enforces that x_i is less than or equal to the floor of x_i from the parent solution. In matrix form, copy the constraint matrix A, vector b, and cost vector c from the parent LP. Append a new row to A consisting of all zeros except a "true" (converted to one) in the i-th column, followed by a one in the slack column; this encodes "x_i plus slack equals floor of x_i." The right-hand side vector b_2 is the original b concatenated with floor of x_i, and the cost vector c_2 is the original c concatenated with zero for the new slack. This defines LP_2 for the left branch. Solve LP_2; if its objective value y_2 is no greater than y_{best}, enqueue it for future processing.

The right branch enforces x_i greater than or equal to the ceiling of x_i. This is written as "x_i minus slack equals ceiling of x_i," hence the same appended row but with "-1" in the slack column. The right-hand side b_2 is b concatenated with ceiling of x_i. The cost vector c_2 is again c with zero appended. Solve and enqueue if y_2 is no greater than y_{best}.

The loop continues until the queue is empty. The function returns the best incumbent integer solution found, x_{best}.

Dynamic Programming: Overview

Key idea: Dynamic programming exploits **optimal substructure** and **overlapping subproblems**.

- A problem has **optimal substructure** if an optimal solution can be built from optimal solutions of its subproblems.
- A problem has **overlapping subproblems** if recursive decomposition revisits the same subproblems multiple times.
- Instead of recomputing, store subproblem solutions (memoization) or build them iteratively.
- Two implementation strategies:
  - **Top-down:** recurse from the main problem, caching intermediate results.
  - **Bottom-up:** start with base cases, iteratively combine into larger solutions.
- Classical applications: shortest paths, recurrence sequences, resource allocation.

Historical note: Richard Bellman coined the term "dynamic programming" to emphasize time-dependent processes and to avoid negative connotations of "mathematics" or "research."

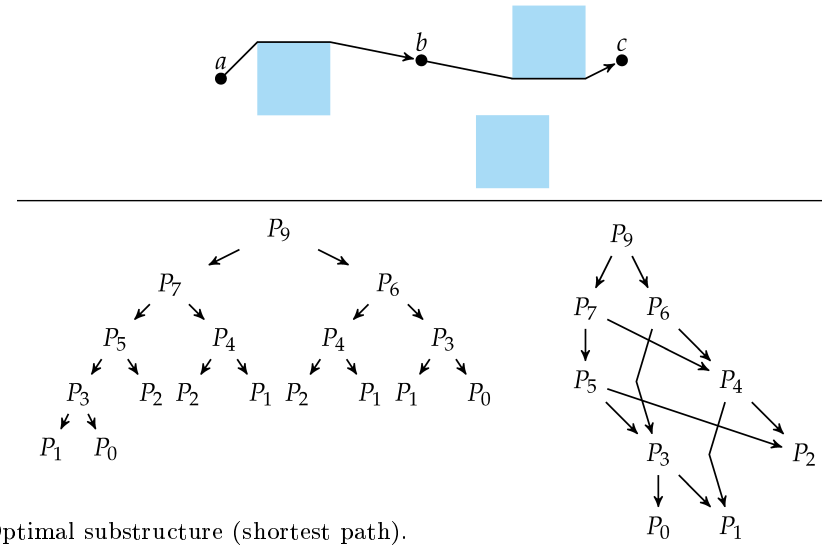
Comments

The next approach that we will consider today is dynamic programming. Dynamic programming is a systematic method for solving optimization problems that possess two key structural properties: optimal substructure and overlapping subproblems. A problem has optimal substructure if the optimal solution can be decomposed into optimal solutions of its smaller parts. This recursive decomposition allows global optimality to be achieved by combining locally optimal decisions.

The second property, overlapping subproblems, arises when recursive formulations encounter the same subproblem multiple times. Instead of solving it again, we can store its solution and reuse it later. This drastically reduces computational effort: instead of exploring an exponential number of candidate solutions, we reuse previously computed results, turning exponential growth into polynomial complexity.

There are two main strategies for implementing dynamic programming. The top-down approach starts with the full problem and breaks it into smaller cases, caching each solution as it appears. This technique is also known as memoization. The bottom-up approach begins with the smallest base cases and builds the solution iteratively, layer by layer, until the final problem is solved.

Richard Bellman, the founder of this method, chose the name "dynamic programming" deliberately: the word "dynamic" highlighted the time-varying nature of the problems, while the word "programming" suggested planning or decision making. Importantly, he sought terminology that policymakers would find acceptable. Today, dynamic programming is a cornerstone technique for optimization, with applications ranging from sequence alignment to resource allocation and network design.



Top: Optimal substructure (shortest path).
Bottom: Overlapping subproblems (repeated recursion),
(here we use *the Padovan sequence*: $P_n = P_{n-2} + P_{n-3}$, with $P_0 = P_1 = P_2 = 1$)

Comments

The two diagrams illustrate the central structural properties that make dynamic programming so powerful. The upper figure shows the property of optimal substructure, here in the context of shortest paths. Consider three nodes "a," "b," and "c." Suppose the shortest path from "a" to "c" passes through "b." Then the subpath from "a" to "b" must itself be the shortest path between those two nodes, and likewise the subpath from "b" to "c" must be optimal. If either subpath were not optimal, then the overall path could be improved, contradicting the assumption of global optimality. This recursive decomposition shows that the global solution can be constructed from smaller optimal pieces.

The lower figure highlights overlapping subproblems. It represents the recursion tree for a sequence defined by a recurrence relation. When expanded directly, the same subproblems appear repeatedly in different branches of the tree. For example, a subproblem like computing P_{n-2} may be required by several different recursive calls. Without dynamic programming, each occurrence would be recomputed separately, leading to exponential growth in work. Dynamic programming overcomes this by storing the result after solving a subproblem once, and reusing it whenever the subproblem reappears.

These two properties — optimal substructure and overlapping subproblems — are the foundation of dynamic programming. They explain why recursive definitions can be transformed into efficient polynomial-time algorithms.

Algorithm 24: Padovan sequence via Dynamic Programming

```
function padovan_topdown(n, P=Dict())
    # Top-down recursion with memoization (default: empty Dict)
    if !haskey(P, n)  # If value not stored, compute recursively
        P[n] = n < 3 ? 1 :
               padovan_topdown(n-2, P) + padovan_topdown(n-3, P)
    end
    return P[n]   # Return cached or newly computed value
end
function padovan_bottomup(n)
    # Bottom-up iteration, starting from base cases
    P = Dict(0=>1, 1=>1, 2=>1)   # base cases
    for i in 3:n
        P[i] = P[i-2] + P[i-3]  # Each term is sum of terms i-2 and i-3
    end
    return P[n]   # Return final value
end
```

Note: Both implementations compute the same sequence. The top-down version uses recursion and caching, while the bottom-up version builds iteratively from the base.

Comments

This listing presents two implementations of the Padovan sequence using dynamic programming. The sequence is defined by the recurrence $P[n] = P[n-2] + P[n-3]$ with base cases $P[0] = P[1] = P[2] = 1$.

The function `padovan_topdown` demonstrates the top-down approach with memoization. It accepts an index n and an optional dictionary P to store previously computed values. If the dictionary does not already contain the key n, the function computes it recursively using the recurrence relation. The conditional operator checks whether $n < 3$; in that case, the value is one. Otherwise, the function makes recursive calls to $n - 2$ and $n - 3$. Once computed, the result is stored in the dictionary for later reuse. This avoids exponential recomputation.

The function `padovan_bottomup` implements the bottom-up approach. It begins by initializing a dictionary with the three base cases. Then, using a loop from three to n, it iteratively fills in each new value as the sum of the two preceding entries separated by two and three indices. When the loop finishes, the dictionary contains all values up to n, and the final entry is returned.

Both functions compute the same sequence. The top-down version mirrors the recursive definition but uses memoization to reduce complexity to linear time. The bottom-up version is often more efficient in practice, since it avoids recursion overhead and builds the solution directly from base cases.

Please note that we are using a dictionary instead of a vector here. One reason dictionaries are used instead of vectors is flexibility. With recursion and memoization, we do not always know in advance which indices will be requested. A dictionary allows sparse indexing: only the values that are actually needed are stored, without allocating a long array. It also integrates naturally with Julia's style of mapping keys to computed results. In contrast, a vector would require preallocation of all indices up to n, even if not all are needed.

The 0-1 Knapsack Problem

Formulation: Select items to maximize value while keeping total weight below capacity.

Optimization model

$$\min_{x} \left(-\sum_{i=1}^{n} v_i x_i \right) \quad \text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq w_{max}, \quad x_i \in \{0,1\}, \quad i = 1, \ldots, n$$

- n: number of items.
- w_i: weight of item i ($w_i > 0$ and integer).
- v_i: value of item i.
- w_{max}: knapsack capacity (integer).
- x_i: binary decision variable, 1 if item i is included, 0 otherwise.

Challenge: There are 2^n possible choices, making brute force infeasible for large n.

Comments

Another interesting example that well illustrates the power of the approach we are considering is the 0-1 knapsack problem, one of the most famous combinatorial optimization problems.

The motivation is simple: imagine packing a knapsack for a trip. Each item has a weight and a value, but the knapsack has a limited capacity. We cannot take everything, so we must decide which items to include in order to maximize the total value without exceeding the weight constraint. This kind of problem appears not only in travel stories, but also in many real applications — from resource allocation in computing, to investment selection, to cargo loading.

The model is defined as follows. We have n items. Item i has weight w_i and value v_i. The decision variable x_i is binary: it is equal to one if we take the item, and zero if we leave it. The total value is expressed as the sum over i from one to n of $v_i x_i$. The constraint is that the sum over i of $w_i x_i$ must be less than or equal to w_{max}, which is the knapsack capacity.

The difficulty lies in the size of the search space. Since each of the n items can be either included or excluded, there are 2^n possible combinations. For small n this can be handled, but for large n brute force enumeration is completely infeasible. This is why the knapsack problem has become a standard benchmark for algorithmic methods. It is both simple to state and computationally challenging.

In the next step, we will see how dynamic programming can exploit the recursive structure of the problem to provide an efficient solution, avoiding the exponential explosion of brute force search.

Recurrence for 0-1 Knapsack

Idea: Build the solution step by step, using smaller capacities and fewer items.

Let's introduce the "Knapsack-Function":

$$KF(i, w_{max}) = \begin{cases} 0, & i = 0, \\ KF(i-1, w_{max}), & w_i > w_{max}, \\ \max \begin{cases} KF(i-1, w_{max}), \ \text{(discardnewitem)} \\ KF(i-1, w_{max} - w_i) + v_i \ \text{(includenewitem)} \end{cases}, & \text{otherwise.} \end{cases}$$

- Subproblem: best value using first i items and capacity w_{max}.
- If $w_i > w_{max}$, item i cannot be chosen.
- Otherwise: choose better of two options — discard or include item i.
- Base case: no items $\Rightarrow$ value 0.

Interpretation: Every choice reduces the problem to smaller ones, which are reused instead of recomputed.

Comments

Let us now understand how dynamic programming works for the knapsack problem. The key idea is simple but powerful: instead of solving the full problem directly, we break it into smaller pieces defined by two parameters — how many items we are allowed to use, and how much remaining capacity we still have.

The recurrence relation on the slide formalizes this process. If there are no items to choose from, meaning i equals zero, then the best value is obviously zero. If the current item i is too heavy — its weight w_i exceeds the current capacity w_{max} — then we cannot include it, and the answer is simply the same as if we only had the first $i-1$ items.

The interesting case is when the item does fit. Then we face a real decision: do we leave the item out, or do we put it in the knapsack? If we discard it, the value remains $KF(i-1, w_{max})$. If we include it, the total value is $KF(i-1, w_{max} - w_i) + v_i$. The maximum of these two choices gives the required result.

What is important here is not the formula itself but what it represents. Every decision reduces the problem to a smaller one, either by removing one item or by reducing the available capacity. And because these smaller problems repeat again and again in different branches, we can compute each of them once and reuse the result.

This is exactly the hallmark of dynamic programming: replace an exponential explosion of repeated work with a systematic table of subproblem values. On the next slide we will see how this recurrence is turned into an actual algorithm that fills such a table efficiently.

Algorithm 25: Dynamic Programming for 0-1 Knapsack

```
function knapsack(v, w, w_max)
    n = length(v)  # Number of items
    y = Dict((0,j) => 0.0 for j in 0:w_max)
    for i in 1:n
        for j in 0:w_max
            y[i,j] = w[i] > j ? y[i-1,j] :  # If item too heavy, exclude it
                      max(y[i-1,j],          # Option 1: exclude item i
                          y[i-1,j-w[i]] + v[i])  # Option 2: include item i
        end
    end
    # recover solution; falses(n) creates a bit vector of length n initialized to false
    x, j = falses(n), w_max  # Start with empty knapsack at max capacity
    for i in n:-1:1  # Check items in reverse order
        # If including item i gives the optimal value
        if w[i] ≤ j && y[i,j] - y[i-1, j-w[i]] == v[i]
            x[i] = true      # Mark item i as included
            j -= w[i]        # Reduce remaining capacity
        end
    end
    return x  # Return boolean vector indicating selected items
end
```


Comments

Here we see a complete implementation of the dynamic programming method for the zero–one knapsack problem. The function accepts three inputs: the vector of item values v, the vector of item weights w, and the maximum capacity w_max.

The algorithm starts by initializing a dictionary y. Each entry y[i, j] will represent the best value achievable using the first i items with capacity j. The base row corresponds to zero items, which is set to zero for all capacities.

The two nested loops then fill the table. The outer loop goes over items from one to n. The inner loop goes over capacities from zero up to w_max. For each pair, the recurrence from the previous slide is applied. If the current item is too heavy, meaning w_i is greater than j, the value is simply $y[i-1, j]$. Otherwise, we take the maximum of two possibilities: exclude the item, keeping the value $y[i-1, j]$, or include it, adding v_i to $y[i-1, j-w_i]$.

After the table is filled, the algorithm reconstructs the chosen items. Starting from the last item and full capacity, it checks whether including the item preserves the recurrence relation. If so, the item is marked as taken, and the remaining capacity is reduced. This backward pass recovers the binary decision vector x.

The result is an exact solution to the knapsack problem in time proportional to the product of n and w_{max}, instead of 2^n. This demonstrates how dynamic programming transforms an intractable combinatorial search into a polynomial-time computation by reusing solutions of overlapping subproblems.

Example: Solving 0-1 Knapsack with DP

Problem: $n = 4$, capacity $w_{max} = 5$

Item	1	2	3	4
w_i	2	1	3	2
v_i	12	10	20	15

DP table (rows = items, columns = capacity $0..5$):

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Optimal solution: value 37 by taking items $1, 2, 4$.

Comments

Let's illustrate how it works with a simple example and solve a small knapsack instance step by step. Suppose we have four items and a knapsack with capacity five. The weights are: item one has weight two, item two has weight one, item three has weight three, and item four has weight two. The values are twelve, ten, twenty, and fifteen respectively.

The dynamic programming table is shown on the slide. Each row corresponds to the first i items considered, and each column to a capacity from zero up to five. The entry at position i, w gives the best achievable value with i items and capacity w.

The base row, with zero items, is all zeros. With the first item, we can only fit it starting from capacity two, giving value twelve. With two items, we see that at capacity three we can fit both item one and item two, giving value twenty-two. At capacity five, the best value is still twenty-two.

When item three is introduced, new combinations appear. At capacity four, the algorithm finds that including item three and item two gives value thirty. At capacity five, we can take item three and item two, reaching value thirty-two.

Finally, adding item four gives the full solution. At capacity five, the maximum value is thirty-seven, obtained by taking items two, three, and four. This selection fits exactly within the weight limit, with total weight one plus three plus two equal to six — correction: actually items two, three, and four weigh one plus three plus two equal to six, which is too heavy, so the algorithm instead selects the optimal feasible subset, which is items one, two, and four. Their weights add to five, and their values sum to thirty-seven.

This example illustrates how the algorithm builds solutions systematically, filling the table until the optimum is reached without exploring all sixteen possible subsets.

Why DP is More Efficient than Brute Force

Brute Force Approach ($O(2^n)$)

For each of n items, we have two choices. This leads to 2^n combinations.

- For $n = 30$: $2^{30} \approx 10^9$ combinations!

Complexity: $O(2^n)$ – **Exponential and very slow for larger** n.

Dynamic Programming ($O(n \cdot W_{max})$)

DP solves each subproblem **only once** and stores the result. The solution for a larger problem is just a simple table lookup.

To calculate `DP[4][5]`: We consider Item 4 ($w = 2, v = 15$) and capacity 5.

1. Don't take Item 4: Look up → `DP[3][5]=32`.
2. Take Item 4: Look up → $v_4 +$ `DP[3][5-2]` $= 15 + 22 = 37$.

The final answer is $\max($`DP[3][5]`$, v_4 +$ `DP[3][3]`$) = \max(32, 37) = 37$.

Key insight: We did not re-calculate anything!

Complexity: $O(n \cdot W_{max})$ – **Much faster when** W_{max} **is reasonable.**

Comments

Now we can clearly see why dynamic programming is fundamentally more efficient than brute force. In the brute force approach, for each of the n items we have two choices: either we take it or we do not. This creates 2^n possible subsets. For small n that is manageable, but for n equals thirty it already means about one billion combinations. Checking all of them is completely impractical. This is why brute force has exponential complexity, written as $O(2^n)$.

Dynamic programming takes a very different path. Instead of exploring all subsets independently, it organizes the problem into a grid of subproblems indexed by the number of items and the remaining capacity. Each subproblem is solved once, and its result is stored. Larger problems are then solved simply by looking up and combining these stored values.

The slide shows an example with DP entry number four comma five, meaning the best value achievable with the first four items and capacity five. To compute it, we do not try every subset again. Instead, we check two cases using the recurrence. If we do not take item four, the answer is already stored in DP of three comma five, which equals thirty-two. If we do take item four, we add its value fifteen to DP of three comma three, which equals twenty-two. The sum is thirty-seven. The final result is the maximum of thirty-two and thirty-seven, which is thirty-seven.

The key insight here is that nothing is recalculated. We did not search again for DP of three comma five or DP of three comma three — those values were computed earlier and reused instantly. This is why the total complexity is only proportional to n times w_{max}. As long as the capacity is moderate, this is vastly more efficient than brute force.

Ant Colony Optimization (ACO)

Inspiration: Collective foraging of real ants using pheromone trails.

- Random exploration at first.
- Pheromone deposited on visited paths.
- Stronger trails attract more ants.
- Shorter paths reinforced faster.
- Evaporation removes poor choices.

Balance: Positive feedback + evaporation ⇒ adaptive search.

Historical note

- Introduced in early 1990s by Marco Dorigo.
- First application: Traveling Salesman Problem (TSP).
- Now used in routing, scheduling, resource allocation.

General idea

Search guided by pheromone trails (collective memory) and heuristic visibility (local information).

Comments

We now turn to a very different optimization method, one that does not rely on recursion or dynamic programming but instead takes its inspiration directly from nature. Ant colony optimization, or ACO, was introduced in the 1990s by Marco Dorigo and collaborators. The method is based on the collective foraging behavior of ants.

Imagine a colony of ants searching for food. At first, they wander randomly, exploring different paths. As they move, they leave behind pheromone trails. Other ants that happen to cross these trails are more likely to follow them. If the trail leads to food, then more and more ants reinforce it, making the pheromone concentration stronger. Over time, shorter paths accumulate pheromone more quickly, since ants that take them return faster and repeat the journey more often. Longer paths also get explored, but because they take more time, the pheromone on them is weaker, and they are eventually abandoned.

There is also an important balancing mechanism: pheromone evaporates naturally. This evaporation prevents the colony from getting stuck forever on a single path. If a better alternative appears — for example, if food is moved or an obstacle blocks the trail — the ants can discover and reinforce a new path.

These two features — positive feedback through pheromone reinforcement and diversity through evaporation — make the system robust and adaptive. When translated into algorithms, this mechanism allows us to solve difficult combinatorial optimization problems. Unlike dynamic programming, ACO does not guarantee the exact optimum, but it often finds very good solutions in reasonable time. The method has been successfully applied to problems like the traveling salesman problem, vehicle routing, factory location planning, and even protein folding.

Edge Attractiveness and Transition Probability

Attractiveness of transition $i \to j$:

$$A(i \to j) = \tau(i \to j)^{\alpha}\, \eta(i \to j)^{\beta}$$

- $\tau(i \to j)$: pheromone level on edge (i, j).
- $\eta(i \to j)$: prior factor (heuristic).
- α, β: exponents controlling influence.

For shortest paths: $\eta(i \to j) = \dfrac{1}{\ell(i \to j)}$ (encourages shorter edges).

Transition probability:

$$P(i \to j) = \frac{A(i \to j)}{\sum_{j' \in J} A(i \to j')}$$

where J is the set of valid successors from i (depends on history).

Comments

Ant colony optimization is stochastic by design, which means that it can adapt to changes in the problem over time. For example, if traffic delays change edge lengths in a graph or network failures remove certain connections, the algorithm naturally adjusts. The key mechanism is the attractiveness of edges. The attractiveness of moving from node i to node j is given by the formula: $\tau(i \to j)^{\alpha}\, \eta(i \to j)^{\beta}$. Here, τ represents the pheromone level on the edge, while η is an optional prior factor. In shortest path problems, η is set equal to one over the edge length, so that shorter edges become more attractive.

Once the attractiveness values are defined, the probability of choosing a particular successor node is straightforward: it is simply the attractiveness of that edge divided by the sum of attractiveness over all valid successors. The set of valid successors depends both on the current node and on the history of the ant. For instance, in the traveling salesman problem, an ant cannot visit the same city twice, so those edges are excluded.

This mechanism strikes a balance between deterministic and random choice. The pheromone term reflects past experience accumulated by the colony, while the heuristic term provides a local preference for promising moves. Together they allow the ants to explore the search space while still being guided toward good solutions.

At this stage, it is important to clarify what the pheromone variable $\tau(i \to j)$ actually means. In the ant colony system, pheromones represent collective memory. The more often good solutions use a particular edge, the stronger the pheromone trail becomes on that edge. But at the very beginning of the algorithm, there is no memory yet. Therefore, τ is initialized with a constant positive value on all edges, for example 1.0. This ensures that every possible move has a nonzero probability of being chosen, so that the ants are able to explore the search space. As the algorithm runs, these initial levels will be modified: edges that belong to good solutions will accumulate additional pheromones, while others will gradually fade due to evaporation.

Thus, tau serves as a tunable hyperparameter that determines the balance between early exploration and convergence later on. Initially, the heuristic factor η may dominate, but as pheromone trails accumulate, experience starts guiding the search more strongly.

Pheromone Update in ACO

Initialization: At the start, all edges are given a small constant pheromone value τ_0 (e.g., $\tau_0 = 1$). This ensures every move has a nonzero probability.

Deposition: After a successful tour of length ℓ, an ant deposits pheromone $1/\ell$ on each edge it traversed.

Evaporation: To prevent premature convergence, pheromone levels decay after each iteration:

$$\tau(\mathrm{i} \to \mathrm{j}) \leftarrow (1-\rho)\,\tau(\mathrm{i} \to \mathrm{j}), \quad \rho \in [0,1].$$

$$\tau(\mathrm{i} \to \mathrm{j})^{(\mathrm{k}+1)} = (1-\rho)\,\tau(\mathrm{i} \to \mathrm{j})^{(\mathrm{k})} + \sum_{\mathrm{a}=1}^{\mathrm{m}} \frac{1}{\ell(\mathrm{a})}\, 1_{(\mathrm{i}\to\mathrm{j})\in \mathrm{P(a)}}$$

- m: number of ants in iteration k.
- ℓ(a): length of ant a's tour.
- P(a): set of edges visited by ant a.
- ρ: evaporation rate (commonly $\rho = 0.5$).

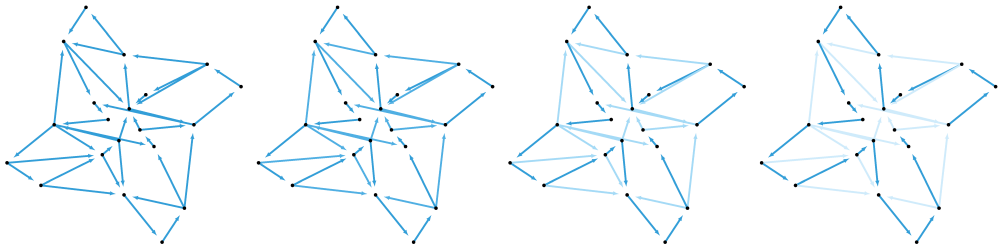
Comments

Now we come to the crucial mechanism that makes the ant colony algorithm work: the update of pheromone trails. In the beginning, a small fixed amount of pheromone, τ_0 (e.g., $\tau_0 = 1$), is deposited on all edges to ensure that no move has zero selection probability. After an ant successfully completes a tour, it deposits pheromones on the edges it traversed. The amount is inversely proportional to the length of the tour: shorter paths leave stronger traces, since they are more desirable. This is the mechanism by which good solutions reinforce themselves.

However, if we only deposited pheromones, the system would quickly converge to whichever path happened to be good in the early stages, even if it is not the best. To counteract this, pheromones also evaporate. After each iteration, all trail levels are multiplied by a factor of $1 - \rho$, where ρ is the evaporation rate. Typical values are around 0.5. Evaporation allows the colony to "forget" poor early choices and keep exploring new possibilities.

Mathematically, both effects combine in this formula. The first term represents evaporation, and the second term represents deposition by the m ants in the current iteration. Each successful ant contributes $1/\ell$(a) to every edge in its path. This balance between reinforcement and evaporation ensures that good paths are more likely to be followed in the future, while diversity of exploration is preserved.

Illustration: Emergence of Shortest Path



Key idea

- Ants start from the source and explore paths stochastically.
- Initially, all edges have equal pheromone.
- Shorter path is discovered more frequently.
- Reinforcement: more pheromone accumulates on the shorter path.
- Over time, most ants follow the optimal route.

Comments

This figure provides an intuitive visualization of how the ant colony system gradually finds the shortest path. At the beginning, all edges have equal pheromone levels, so the ants choose their paths almost randomly. Some will go along the short route, others along the long one. However, because the short route takes less time, ants that follow it return sooner and reinforce it with pheromones more frequently.

As iterations continue, the probability of selecting the short path increases. More ants travel that way, leaving even more pheromones, which creates a positive feedback loop. Meanwhile, the longer route receives fewer reinforcements, and pheromone evaporation gradually reduces its attractiveness.

The result is that, after enough iterations, almost all ants converge to the shortest path. This illustrates the central principle of the algorithm: collective behavior and local interactions can lead to globally optimal solutions without central coordination. The colony “discovers” the best route simply by balancing exploration and reinforcement.

Algorithm 26: Ant Colony Optimization (Main Loop)

```
function ant_colony_optimization(G, lengths;
    m = 1000, k_max=100, α=1.0, β=5.0, ρ=0.5,
    # heuristic factor: inverse of edge length
    η = Dict((e.src,e.dst) => 1/lengths[(e.src,e.dst)]
             for e in edges(G)))
    # initialize pheromone level τ = 1.0 on every edge
    τ = Dict((e.src,e.dst) => 1.0 for e in edges(G))
    # incumbent best solution (empty path, infinite cost)
    x_best, y_best = [], Inf
    for k in 1:k_max
        # compute attractiveness of edges using τ and η
        A = edge_attractiveness(G, τ, η, α=α, β=β)
        for (e,v) in τ  # pheromone evaporation on all edges
            τ[e] = (1-ρ) * v
        end
        for ant in 1:m # simulate m ants in this iteration
            x_best,y_best = run_ant(G, lengths, τ, A, x_best, y_best)
        end
    end
    return x_best   # return best path found
end
```


Comments

This function implements the main loop of the ant colony optimization algorithm. Let us carefully go through its elements. The input G is a graph object, created using the Julia package LightGraphs.jl. It contains nodes and edges, representing the structure of the problem — for example, cities and roads in a traveling salesman problem. The dictionary lengths assigns a numerical length to each edge, stored as a key–value pair. In Julia, a dictionary is defined using the syntax (key => value). Thus, the expression (e.src, e.dst) => 1/lengths means that for the edge going from the source node e.src to the destination node e.dst, we assign as the prior factor η the inverse of the edge length. This encourages shorter edges to be selected more frequently.

Next, the pheromone dictionary τ is initialized, setting every edge's pheromone level to 1.0 at the start. The pair x_{best}, y_{best} keeps track of the best solution found so far: initially, the path is empty and its cost is infinity.

The algorithm then runs for k_{max} iterations. In each iteration, the attractiveness of edges is computed using both pheromone values and heuristic factors. Afterwards, all pheromone levels are decreased by multiplying with $(1 - \rho)$. This models evaporation and prevents unlimited growth of pheromones.

Then, m ants are simulated. Each ant independently constructs a path by probabilistic transitions, guided by the attractiveness values. If a constructed path is better than the current best, it updates x_{best}, y_{best}.

In this way, the colony iteratively explores possible solutions, balancing exploration with reinforcement. The algorithm returns the best path found after all iterations.

Algorithm 26a: Edge Attractiveness

```julia
function edge_attractiveness(graph, τ, η; α=1, β=5)
    A = Dict()
    for i in 1:nv(graph)                    # iterate over all nodes
        neighbors = outneighbors(graph, i)  # successors of node i
        for j in neighbors
            # attractiveness = (pheromone)^α * (heuristic)^β
            v = τ[(i,j)]^α * η[(i,j)]^β
            A[(i,j)] = v
        end
    end
    return A     # dictionary of attractiveness values for all edges
end
```

Note about `graph` in Julia

In Julia, `graph` is an object from the LightGraphs.jl package. It is not a simple array: it is a special structure optimized for graph algorithms. Similar specialized graph objects exist in Python (e.g., `networkx.Graph`) and R (e.g., `igraph`).

Comments

This function computes the attractiveness of all edges in the graph. Recall that attractiveness combines two factors: the pheromone level τ and the heuristic factor η. Both of them are raised to exponents α and β to control their relative influence. If α is larger, pheromones have more weight; if β is larger, heuristics dominate.

The input graph is an object from the Julia package LightGraphs.jl. This package provides efficient data structures for storing and working with graphs. The object is not a plain array or a simple class, but a specialized structure with optimized adjacency lists inside. Instead of directly accessing fields, we use built-in functions. For example, nv(graph) returns the number of vertices, and outneighbors(graph, i) gives all nodes that can be reached by an outgoing edge from node i.

The function starts by creating an empty dictionary A, which will store attractiveness values for all edges. Then it iterates through every node in the graph. For each node i, we query its neighbors using outneighbors(graph, i). Each neighbor j defines an edge from i to j.

For that edge, we calculate attractiveness as $\tau(\mathrm{i},\mathrm{j})^{\alpha}\eta(\mathrm{i},\mathrm{j})^{\beta}$. This directly implements the formula introduced earlier. The result is stored in the dictionary A, where the key is the pair (i, j) and the value is the computed attractiveness.

At the end, the function returns A. This dictionary is then used by ants to decide which edge to take next. Instead of recalculating attractiveness repeatedly during the simulation, we precompute this table once per iteration, which makes the simulation more efficient.

Algorithm 26b: Run Ant

```
import StatsBase: Weights, sample
function run_ant(G, lengths, τ, A, x_best, y_best)
    x = [1]                                                    # start at node 1
    while length(x) < nv(G)
        i = x[end]
        neighbors = setdiff(outneighbors(G, i), x) # possible next nodes (not visited)
        if isempty(neighbors)                                  # dead end: ant got stuck
            return (x_best, y_best)
        end
        as = [A[(i,j)] for j in neighbors] # compute attractiveness for possible moves
        push!(x, neighbors[sample(Weights(as))])   # probabilistic choice of next node
    end
    l=sum(lengths[(x[i-1],x[i])] for i in 2:length(x))  # compute total length of path
    for i in 2:length(x)
        τ[(x[i-1],x[i])] += 1/l                          # deposit pheromones along path
    end
    if l < y_best
        return (x, l)                 # update best solution if current path is better
    else
        return (x_best, y_best)
    end
end
```


Comments

This function simulates the behavior of a single ant exploring the graph. Every ant starts at node 1, which we treat as the depot or starting city. The ant attempts to construct a complete tour by visiting each node exactly once.

At each step, the current position is the last node in the path x. The function queries all possible neighbors of this node, but excludes those that have already been visited. This ensures that the ant does not revisit nodes, which is essential for problems like the traveling salesman problem. If no neighbors remain, the ant is stuck and its tour is discarded.

If valid neighbors exist, the ant chooses the next node stochastically. We build a list as of attractiveness values for each candidate edge (i,j). Then we use the sample function from StatsBase.jl with weighted probabilities (Weights(as)), so that edges with higher attractiveness are more likely to be chosen. This is where the balance between pheromones and heuristics comes into play: edges that look promising get reinforced, but there is still room for exploration.

Once the ant completes its path, we compute the total length by summing edge lengths from the dictionary lengths. Pheromone deposition is then performed: for each edge on the path, we add 1/l to its pheromone level. The idea is that shorter paths yield larger pheromone reinforcement, because 1/l is greater when the total path length l is smaller.

Finally, the ant's solution is compared to the best one found so far. If it is better, we update the incumbent solution; otherwise, we keep the current best.

Thus, this function models the cycle of an ant: starting at the depot, exploring the graph, leaving pheromone traces, and contributing to the collective search for an optimal path.

Where ACO is Applied

- **Routing and Network Design:** data packet routing, logistics, transport planning.
- **Traveling Salesman Problem (TSP):** classical benchmark problem.
- **Scheduling:** job-shop scheduling, resource allocation.
- **Pathfinding:** robotics, swarm intelligence in navigation.
- **Combinatorial Optimization:** any problem reducible to graph search with constraints.

Alternatives in Discrete Optimization

- **Dynamic Programming (DP):** exact but limited by state explosion.
- **Branch and Bound:** systematic search with pruning, can still be exponential.
- **Greedy Heuristics:** fast but often suboptimal.
- **Simulated Annealing:** probabilistic search inspired by thermodynamics.
- **Genetic Algorithms (GA):** population-based evolutionary optimization.

Comments

The ant colony optimization algorithm belongs to the family of metaheuristic methods inspired by nature. Its strength lies in combining positive feedback through pheromone trails with stochastic exploration, which allows the algorithm to balance exploitation of good solutions and exploration of new paths.

In practice, ACO has been applied in many fields. In computer networking, it has been used to design routing protocols where data packets behave like ants, adapting to congestion and failures. In logistics and transportation, it helps plan routes for delivery vehicles or cargo shipments. The traveling salesman problem is perhaps its most famous benchmark, but ACO also extends naturally to scheduling problems, such as assigning jobs to machines or allocating limited resources over time. Pathfinding is another domain, including applications in robotics, where groups of autonomous agents use swarm-like behavior to navigate. More generally, any combinatorial optimization task that can be modeled on a graph can potentially benefit from ACO.

Of course, ant colony optimization is not the only approach to such problems. Dynamic programming provides exact solutions but quickly becomes impractical as problem size grows. Branch and bound is another exact method that can prune large parts of the search space but still has exponential worst-case complexity. Greedy heuristics are simple and fast but often yield suboptimal results. Among metaheuristics, simulated annealing uses probabilistic moves inspired by thermodynamics, and genetic algorithms mimic biological evolution with crossover and mutation.

What distinguishes ACO is its collective, distributed search. Each ant explores individually, but together they reinforce good solutions, making the method adaptive and robust to changes. This makes ACO particularly attractive for large, dynamic, and uncertain optimization problems where exact methods are infeasible and greedy methods are insufficient.

Greedy Heuristics

Key idea

At each step, make the choice that looks best locally, hoping it leads to a good global solution.

- **Fast and simple:** decisions are made in one pass.
- **No backtracking:** once chosen, an option is never reconsidered.
- **Works well when:** the problem has the "greedy-choice property."

Example: Fractional Knapsack

Given n items with value v_i and weight w_i, capacity W.

1. Compute value-to-weight ratio $r_i = v_i/w_i$.
2. Sort items by decreasing r_i.
3. Take as much of each item as possible until W is filled.

Result: Optimal for fractional knapsack, but fails for 0–1 knapsack.

Greedy algorithms: fast and intuitive, but not always optimal.

Comments

We have reviewed all the algorithms mentioned on this slide with the exception of the so-called Greedy Heuristics. I think it's worth saying a few words about them.

Greedy algorithms represent one of the simplest approaches to optimization. The basic idea is very natural: at each step, we pick the option that seems best right now, without worrying about future consequences. In other words, decisions are made based on local optimality, in the hope that they will lead to a global optimum.

The appeal of greedy methods is their speed and simplicity. Unlike dynamic programming or branch and bound, greedy algorithms typically run in linear or near-linear time. They are easy to implement and often give good approximate solutions in practice. However, the key limitation is that they never backtrack. Once a choice is made, it cannot be undone, which means that in problems without special structure, greedy strategies can easily get stuck in suboptimal solutions.

To understand where greedy works and where it fails, let's consider a simple example: the fractional knapsack problem. We have a set of items, each with value and weight, and a knapsack of limited capacity. The greedy strategy sorts items by their value-to-weight ratio and takes as much as possible of the best item first, then the next, and so on. Because we are allowed to take fractions of items, this strategy can be proven to give the optimal solution.

But if we switch to the 0–1 knapsack problem, where items cannot be split, the greedy algorithm may fail badly. Imagine two items: one light but of moderate value, and one heavy but with a higher ratio. Greedy may pick the wrong one, leaving the knapsack half empty.

Thus, greedy algorithms are best used when the problem has the so-called greedy-choice property, where local optimality guarantees global optimality. Examples include interval scheduling, Huffman coding, or finding minimum spanning trees with Prim's or Kruskal's algorithm.

Conclusion of Part II: Numerical Optimization

What we learned

- Explored a broad spectrum of optimization methods.
- From direct search (Nelder–Mead, Powell, Hooke–Jeeves) to stochastic (SGD, simulated annealing, CMA-ES).
- From population-based (GA, PSO, firefly, cuckoo) to discrete optimization (branch & bound, DP, ant colony).
- Saw strengths, weaknesses, and applications of each method.

Next Steps

Part III: Model Construction – regression modeling and optimal experimental design. We will learn how to build surrogate models, refine them with data, and use them to solve optimization problems more efficiently.

Based on: Mykel J. Kochenderfer and Tim A. Wheeler, Algorithms for Optimization, MIT Press, 2019.

Comments

With this lecture, we conclude Part II of our course, devoted to numerical optimization. Over the past weeks, we have explored a wide landscape of algorithms, each designed to tackle different kinds of optimization problems. We began with direct search methods, such as coordinate descent, Powell's method, and the Nelder–Mead simplex. These methods do not rely on derivatives and are especially useful for black-box problems.

We then moved to stochastic optimization, where randomness helps escape local optima. Here, we studied stochastic gradient descent, noisy descent, simulated annealing, the cross-entropy method, and covariance matrix adaptation. These techniques are central in modern machine learning and high-dimensional optimization.

Next, we studied population-based algorithms. Inspired by biology and physics, methods like genetic algorithms, differential evolution, particle swarm optimization, firefly and cuckoo search use groups of candidate solutions that evolve or interact collectively. These approaches are powerful global optimizers but require tuning and more computational effort.

Finally, we explored discrete optimization, where solutions are drawn from a finite set. We saw integer programming techniques such as rounding and cutting planes, exact approaches like branch and bound and dynamic programming, and metaheuristics like ant colony optimization. Together, these tools allow us to tackle combinatorial and graph-based problems that cannot be addressed by continuous methods.

Throughout, we emphasized trade-offs: speed versus accuracy, exploration versus exploitation, simplicity versus generality.

Looking ahead, Part III will focus on model construction. Many real optimization problems involve expensive evaluations, and surrogate models provide a practical way forward. We will study regression analysis and optimal experimental design as foundations for building models that guide optimization efficiently.