19  **Discrete Optimization**

Previous chapters have focused on optimizing problems involving design variables that are continuous. Many problems, however, have design variables that are naturally discrete, such as manufacturing problems involving mechanical components that come in fixed sizes or navigation problems involving choices between discrete paths. A *discrete optimization* problem has constraints such that the design variables must be chosen from a discrete set. Some discrete optimization problems have infinite design spaces, and others are finite.\(^1\) Even for finite problems, where we could in theory enumerate every possible solution, it is generally not computationally feasible to do so in practice. This chapter discusses both exact and approximate approaches to solving discrete optimization problems that avoid enumeration. Many of the methods covered earlier, such as simulated annealing and genetic programming, can easily be adapted for discrete optimization problems, but we will focus this chapter on categories of techniques we have not yet discussed.

Discrete optimization constrains the design to be integral. Consider the problem:

\[
\begin{align*}
\text{minimize} \quad & x_1 + x_2 \\
\text{subject to} \quad & \|x\| \leq 2 \\
& x \text{ is integral}
\end{align*}
\]

The optimum in the continuous case is \(x^* = [-\sqrt{2}, -\sqrt{2}]\) with a value of \(y = -2\sqrt{2} \approx -2.828\). If \(x_1\) and \(x_2\) are constrained to be integer-valued, then the best we can do is to have \(y = -2\) with \(x^* \in \{[-2, 0], [-1, -1], [0, -2]\}\).

19.1 Integer Programs

An integer program is a linear program with integral constraints. By integral constraints, we mean that the design variables must come from the set of integers. Integer programs are sometimes referred to as integer linear programs to emphasize the assumption that the objective function and constraints are linear.

An integer program in standard form is expressed as:

\[
\begin{align*}
\text{minimize} & \quad c^T x \\
\text{subject to} & \quad Ax \leq b \\
& \quad x \geq 0 \\
& \quad x \in \mathbb{Z}^n
\end{align*}
\]  

(19.1)

where \( \mathbb{Z}^n \) is the set of \( n \)-dimensional integral vectors.

Like linear programs, integer programs are often solved in equality form. Transforming an integer program to equality form often requires adding additional slack variables \( s \) that do not need to be integral. Thus, the equality form for integral programs is:

\[
\begin{align*}
\text{minimize} & \quad c^T x \\
\text{subject to} & \quad Ax + s = b \\
& \quad x \geq 0 \\
& \quad s \geq 0 \\
& \quad x \in \mathbb{Z}^n
\end{align*}
\]  

(19.2)

More generally, a mixed integer program (algorithm 19.1) includes both continuous and discrete design components. Such a problem, in equality form, is expressed as:

\[
\begin{align*}
\text{minimize} & \quad c^T x \\
\text{subject to} & \quad Ax = b \\
& \quad x \geq 0 \\
& \quad x_D \in \mathbb{Z}^{|D|}
\end{align*}
\]  

(19.3)

where \( D \) is a set of indices into the design variables that are constrained to be discrete. Here, \( x = [x_D, x_C] \), where \( x_D \) represents the vector of discrete design variables and \( x_C \) the vector of continuous design variables.
19.2 Rounding

A common approach to discrete optimization is to relax the constraint that the design points must come from a discrete set. The advantage of this relaxation is that we can use techniques, like gradient descent or linear programming, that take advantage of the continuous nature of the objective function to direct the search. After a continuous solution is found, the design variables are rounded to the nearest feasible discrete design.

There are potential issues with rounding. Rounding might result in an infeasible design point, as shown in figure 19.1. Even if rounding results in a feasible point, it may be far from optimal, as shown in figure 19.2. The addition of the discrete constraint will typically worsen the objective value as illustrated in example 19.1. However, for some problems, we can show the relaxed solution is close to the optimal discrete solution.

We can solve integer programs using rounding by removing the integer constraint, solving the corresponding linear program, or LP, and then rounding the solution to the nearest integer. This method is implemented in algorithm 19.2.

Algorithm 19.1. A mixed integer linear program type that reflects equation (19.3). Here, $D$ is the set of design indices constrained to be discrete.

Figure 19.1. Rounding can produce an infeasible design point.

Figure 19.2. The nearest feasible discrete design may be significantly worse than the best feasible discrete design.

Algorithm 19.2. Methods for relaxing a mixed integer linear program into a linear program and solving a mixed integer linear program by rounding. Both methods accept a mixed integer linear program $MIP$. The solution obtained by rounding may be suboptimal or infeasible.

discrete solution $x_d^*$ with $\|x_c^* - x_d^*\|_\infty$ less than or equal to $n$ times the maximum absolute value of the determinants of the submatrices of $A$.

The vector $c$ need not be integral for an LP to have an optimal integral solution because the feasible region is purely determined by $A$ and $b$. Some approaches use the dual formulation for the LP, which has a feasible region dependent on $c$, in which case having an integral $c$ is also required.

In the special case of totally unimodular integer programs, where $A$, $b$, and $c$ have all integer entries and $A$ is totally unimodular, the simplex algorithm is guaranteed to return an integer solution. A matrix is totally unimodular if the determinant of every submatrix is 0, 1, or $-1$, and the inverse of a totally unimodular matrix is also integral. In fact, every vertex solution of a totally unimodular integer program is integral. Thus, every $Ax = b$ for unimodular $A$ and integral $b$ has an integral solution.

Several matrices and their total unimodularity are discussed in example 19.2. Methods for determining whether a matrix or an integer linear program are totally unimodular are given in algorithm 19.3.

Consider the following matrices:

\[
\begin{bmatrix}
1 & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & -1
\end{bmatrix} \quad \begin{bmatrix}
1 & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 0
\end{bmatrix} \quad \begin{bmatrix}
-1 & -1 & 0 & 0 & 0 \\
1 & 0 & -1 & -1 & 0 \\
0 & 1 & 1 & 0 & -1
\end{bmatrix}
\]

The left matrix is not totally unimodular, as

\[
\begin{vmatrix}
1 & 1 \\
1 & -1
\end{vmatrix} = -2
\]

The other two matrices are totally unimodular.

### 19.3 Cutting Planes

The cutting plane method is an exact method for solving mixed integer programs when $A$ is not totally unimodular. Modern practical methods for solving integer programs use branch and cut algorithms that combine the cutting plane method

---

5 A submatrix is a matrix obtained by deleting rows and/or columns of another matrix.


Algorithm 19.3. Methods for determining whether matrices $A$ or mixed integer programs $MIP$ are totally unimodular. The method $\text{isint}$ returns true if the given value is integral.

```python
isint(x, ε=1e-10) = abs(round(x) - x) ≤ ε

function is_totally_unimodular(A::Matrix)
    # all entries must be in $[0,1,-1]$
    if any(a ∉ (0,-1,1) for a in A)
        return false
    end
    # brute force check every subdeterminant
    r,c = size(A)
    for i in 1 : min(r,c)
        for a in subsets(1:r, i)
            for b in subsets(1:c, i)
                B = A[a,b]
                if det(B) ∉ (0,-1,1)
                    return false
                end
            end
        end
    end
    return true
end

function is_totally_unimodular(MIP)
    return is_totally_unimodular(MIP.A) &&
    all(isint, MIP.b) && all(isint, MIP.c)
end
```
with the branch and bound method, discussed in the next section. The cutting plane method works by solving the relaxed LP and then adding linear constraints that result in an optimal solution.

We begin the cutting method with a solution \( \mathbf{x}^* \) to the relaxed LP, which must be a vertex of \( \mathbf{A}\mathbf{x} = \mathbf{b} \). If the \( D \) components in \( \mathbf{x}^*_c \) are integral, then it is also an optimal solution to the original mixed integer program, and we are done. As long as the \( D \) components in \( \mathbf{x}^*_c \) are not integral, we find a hyperplane with \( \mathbf{x}^*_c \) on one side and all feasible discrete solutions on the other. This cutting plane is an additional linear constraint to exclude \( \mathbf{x}^*_c \). The augmented LP is then solved for a new \( \mathbf{x}^*_c \).

Each iteration of algorithm 19.4 introduces cutting planes that make nonintegral components of \( \mathbf{x}^*_c \) infeasible while preserving the feasibility of the nearest integral solutions and the rest of the feasible set. The integer program modified with these cutting plane constraints is solved for a new relaxed solution. Figure 19.3 illustrates this process.

We wish to add constraints that cut out nonintegral components of \( \mathbf{x}^*_c \). For an LP in equality form with constraint \( \mathbf{A}\mathbf{x} = \mathbf{b} \), recall from section 11.2.1 that we can partition a vertex solution \( \mathbf{x}^*_c \) to arrive at

\[
\mathbf{A}_B \mathbf{x}_B^* + \mathbf{A}_V \mathbf{x}_V^* = \mathbf{b}
\]

(19.4)

where \( \mathbf{x}_V^* = \mathbf{0} \). The nonintegral components of \( \mathbf{x}_c^* \) will thus occur only in \( \mathbf{x}_B^* \).

We can introduce an additional inequality constraint for each \( b \in B \) such that \( x^*_b \) is nonintegral:\(^8\)

\[
x^*_b - \lfloor x^*_b \rfloor - \sum_{v \in V} (\tilde{A}_{bv} - \lfloor \tilde{A}_{bv} \rfloor) x_v \leq 0
\]

(19.5)

\(^8\) Note that \( \lfloor x \rfloor \), or floor of \( x \), rounds \( x \) down to the nearest integer.

© 2019 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.

2022-05-22 00:25:57-07:00, revision 47fd495, comments to bugs@algorithmsbook.com
frac(x) = modf(x)[1]

function cutting_plane(MIP)
    LP = relax(MIP)
    x, b_inds, v_inds = minimize_lp(LP)
    n_orig = length(x)
    D = copy(MIP.D)
    while !all(isint(x[i]) for i in D)
        AB, AV = LP.A[:,b_inds], LP.A[:,v_inds]
        Abar = AB\AV
        b = 0
        for i in D
            if !isint(x[i])
                b += 1
                A2 = [LP.A zeros(size(LP.A,1));
                      zeros(1,size(LP.A,2)+1)]
                A2[end,end] = 1
                A2[end,v_inds] = (x->floor(x) - x).*(Abar[b,:])
                b2 = vcat(LP.b, -frac(x[i]))
                c2 = vcat(LP.c, 0)
                LP = LinearProgram(A2,b2,c2)
            end
        end
        x, b_inds, v_inds = minimize_lp(LP)
    end
    return x[1:n_orig]
end

Algorithm 19.4. The cutting plane method solves a given mixed integer program $MIP$ and returns an optimal design vector. An error is thrown if no feasible solution exists. The helper function $frac$ returns the fractional part of a number, and the implementation for $minimize_lp$, algorithm 11.5, has been adjusted to return the basic and nonbasic indices $b\_inds$ and $v\_inds$ along with an optimal design $x$.© 2019 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license. 2022-05-22 00:25:57-07:00, revision 47fd495, comments to bugs@algorithmsbook.com
where $\bar{A} = A_B^{-1}A_V$. These cutting planes use only the $V$-components to cut out the nonintegral components of $x^*$.

Introducing a cutting plane constraint cuts out the relaxed solution $x^*$, because all $x^*_v$ are zero:

$$x^*_b - \lfloor x^*_b \rfloor - \sum_{v \in V} (\bar{A}_{bv} - \lfloor \bar{A}_{bv} \rfloor) x^*_v > 0 \quad (19.6)$$

A cutting plane is written in equality form using an additional integral slack variable $x_k$:

$$x_k + \sum_{v \in V} (|\bar{A}_{bv}| - \bar{A}_{bv}) x_v = |x^*_b| - x^*_b \quad (19.7)$$

Each iteration of algorithm 19.4 thus increases the number of constraints and the number of variables until solving the LP produces an integral solution. Only the components corresponding to the original design variables are returned.

The cutting plane method is used to solve a simple integer linear program in example 19.3.

### 19.4 Branch and Bound

One method for finding the global optimum of a discrete problem, such as an integer program, is to enumerate all possible solutions. The branch and bound method guarantees that an optimal solution is found without having to evaluate all possible solutions. Many commercial integer program solvers use ideas from both the cutting plane method and branch and bound. The method gets its name from the branch operation that partitions the solution space and the bound operation that computes a lower bound for a partition.

Branch and bound is a general method that can be applied to many kinds of discrete optimization problems, but we will focus here on how it can be used for integer programming. Algorithm 19.5 provides an implementation that uses a priority queue, which is a data structure that associates priorities with elements in a collection. We can add an element and its priority value to a priority queue using the `enqueue!` operation. We can remove the element with the minimum priority value using the `dequeue!` operation.


Consider the integer program:

\[
\begin{align*}
\text{minimize} \quad & 2x_1 + x_2 + 3x_3 \\
\text{subject to} \quad & \begin{bmatrix} 0.5 & -0.5 & 1.0 \\ 2.0 & 0.5 & -1.5 \end{bmatrix} x = \begin{bmatrix} 2.5 \\ -1.5 \end{bmatrix} \\
& x \geq 0 \quad x \in \mathbb{Z}^3
\end{align*}
\]

The relaxed solution is \( x^* \approx [0.818, 0, 2.091] \), yielding:

\[
\begin{align*}
A_B &= \begin{bmatrix} 0.5 & 1 \\ 2 & -1.5 \end{bmatrix} \\
A_Y &= \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} \\
\bar{A} &= \begin{bmatrix} -0.091 \\ -0.455 \end{bmatrix}
\end{align*}
\]

From equation (19.7), the constraint for \( x_1 \) with slack variable \( x_4 \) is:

\[
x_4 + ([-0.091] - (-0.091))x_2 = [0.818] - 0.818 \\
x_4 - 0.909x_2 = -0.818
\]

The constraint for \( x_3 \) with slack variable \( x_5 \) is:

\[
x_5 + ([-0.455] - (-0.455))x_2 = [2.091] - 2.091 \\
x_5 - 0.545x_2 = -0.091
\]

The modified integer program has:

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

Solving the modified LP, we get \( x^* \approx [0.9, 0.9, 2.5, 0.0, 0.4] \). Since this point is not integral, we repeat the procedure with constraints:

\[
x_6 - 0.9x_4 = -0.9 \\
x_7 - 0.9x_4 = -0.9 \\
x_8 - 0.5x_4 = -0.5 \\
x_9 - 0.4x_4 = -0.4
\]

and solve a third LP to obtain: \( x^* = [1, 2, 3, 1, 1, 0, 0, 0, 0] \) with a final solution of \( x^*_i = [1, 2, 3] \).
The algorithm begins with a priority queue containing a single LP relaxation of the original mixed integer program. Associated with that LP is a solution $x^*_c$ and objective value $y_c = c^\top x^*_c$. The objective value serves as a lower bound on the solution and thus serves as the LP’s priority in the priority queue. At each iteration of the algorithm, we check whether the priority queue is empty. If it is not empty, we dequeue the LP with the lowest priority value. If the solution associated with that element has the necessary integral components, then we keep track of whether it is the best integral solution found so far.

If the dequeued solution has one or more components in $\mathcal{D}$ that are nonintegral, we choose from $x^*_c$ such a component that is farthest from an integer value. Suppose this component corresponds to the $i$th design variable. We branch by considering two new LPs, each one created by adding one of the following constraints to the dequeued LP:\footnote{Note that $\lceil x \rceil$, or ceiling of $x$, rounds $x$ up to the nearest integer.}

\[
x_i \leq \lfloor x^*_c \rfloor \quad \text{or} \quad x_i \geq \lceil x^*_c \rceil
\] (19.8)

as shown in figure 19.4. Example 19.4 demonstrates this process.

We compute the solution associated with these two LPs, which provide lower bounds on the value of the original mixed integer program. If either solution lowers the objective value when compared to the best integral solution seen so far, it is placed into the priority queue. Not placing solutions already known to be inferior to the best integral solution seen thus far allows branch and bound to prune the search space. The process continues until the priority queue is empty, and we return the best feasible integral solution. Example 19.5 shows how branch and bound can be applied to a small integer program.

---

Consider a relaxed solution $x^*_c = [3, 2.4, 1.2, 5.8]$ for an integer program with $c = [-1, -2, -3, -4]$. The lower bound is

\[
y \geq c^\top x^*_c = -34.6
\]

We branch on a nonintegral coordinate of $x^*_c$, typically the one farthest from an integer value. In this case, we choose the first nonintegral coordinate, $x^*_{2,c'}$, which is 0.4 from the nearest integer value. We then consider two new LPs, one with $x_2 \leq 2$ as an additional constraint and the other with $x_2 \geq 3$ as an additional constraint.
19.4. Branching splits the feasible set into subsets with an additional integral inequality constraint.

Figure 19.4. Branching splits the feasible set into subsets with an additional integral inequality constraint.
Algorithm 19.5. The branch and bound algorithm for solving a mixed integer program \texttt{MIP}. The helper method \texttt{minimize_lp_and_y} solves an LP and returns both the solution and its value. An infeasible LP produces a NaN solution and an Inf value. More sophisticated implementations will drop variables whose solutions are known in order to speed computation. The \texttt{PriorityQueue} type is provided by the \texttt{DataStructures.jl} package.
We can use branch and bound to solve the integer program in example 19.3. As before, the relaxed solution is $x^*_c = [0.818, 0, 2.09]$, with a value of 7.909. We branch on the first component, resulting in two integer programs, one with $x_1 \leq 0$ and one with $x_1 \geq 1$:

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

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

The left LP with $x_1 \leq 0$ is infeasible. The right LP with $x_1 \geq 1$ has a relaxed solution, $x^*_c = [1, 2, 3, 0]$, and a value of 13. We have thus obtained our integral solution, $x^*_i = [1, 2, 3]$.

19.5 Dynamic Programming

Dynamic programming\(^\text{12}\) is a technique that can be applied to problems with optimal substructure and overlapping subproblems. A problem has optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. Figure 19.5 shows an example.

A problem with overlapping subproblems solved recursively will encounter the same subproblem many times. Instead of enumerating exponentially many potential solutions, dynamic programming either stores subproblem solutions, and thereby avoids having to recompute them, or recursively builds the optimal solu-

\(^\text{12}\) The term dynamic programming was chosen by Richard Bellman to reflect the time-varying aspect of the problems he applied it to and to avoid the sometimes negative connotations words like research and mathematics had. He wrote, “I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.” R. Bellman, Eye of the Hurricane: An Autobiography. World Scientific, 1984. p. 159.
tion in a single pass. Problems with recurrence relations often have overlapping subproblems. Figure 19.6 shows an example.

Dynamic programming can be implemented either top-down or bottom-up, as demonstrated in algorithm 19.6. The top-down approach begins with the desired problem and recurses down to smaller and smaller subproblems. Subproblem solutions are stored so that when we are given a new subproblem, we can either retrieve the computed solution or solve and store it for future use. The bottom-up approach starts by solving the smaller subproblems and uses their solutions to obtain solutions to larger problems.

Algorithm 19.6. Computing the Padovan sequence using dynamic programming, with both the top-down and bottom-up approaches.

```plaintext
function padovan_topdown(n, P=Dict())
    if !haskey(P, n)
        P[n] = n < 3 ? 1 : padovan_topdown(n-2,P) + padovan_topdown(n-3,P)
    end
    return P[n]
end

function padovan_bottomup(n)
    P = Dict(0=>1,1=>1,2=>1)
    for i in 3 : n
        P[i] = P[i-2] + P[i-3]
    end
    return P[n]
end
```

13 Storing subproblem solutions in this manner is called memoization.
The knapsack problem is a well-known combinatorial optimization problem that often arises in resource allocation. Suppose we are packing our knapsack for a trip, but we have limited space and want to pack the most valuable items. There are several variations of the knapsack problem. In the 0-1 knapsack problem, we have the following optimization problem:

\[
\begin{align*}
\text{minimize} & \quad -\sum_{i=1}^{n} v_i x_i \\
\text{subject to} & \quad \sum_{i=1}^{n} w_i x_i \leq w_{\text{max}} \\
& \quad x_i \in \{0, 1\} \text{ for all } i \in \{1, \ldots, n\}
\end{align*}
\]

(19.9)

We have \(n\) items, with the \(i\)th item having integral weight \(w_i > 0\) and value \(v_i\). The design vector \(x\) consists of binary values that indicate whether an item is packed. The total weight cannot exceed our integral capacity \(w_{\text{max}}\), and we seek to maximize the total value of packed items.

There are \(2^n\) possible design vectors, which makes direct enumeration for large \(n\) intractable. However, we can use dynamic programming. The 0-1 knapsack problem has optimal substructure and overlapping subproblems. Consider having solved knapsack problems with \(n\) items and several capacities up to \(w_{\text{max}}\). The solution to a larger knapsack problem with one additional item with weight \(w_{n+1}\) and capacity \(w_{\text{max}}\) will either include or not include the new item:

- If it is not worth including the new item, the solution will have the same value as a knapsack with \(n-1\) items and capacity \(w_{\text{max}}\).

- If it is worth including the new item, the solution will have the value of a knapsack with \(n-1\) items and capacity \(w_{\text{max}} - w_{n+1}\) plus the value of the new item.

The recurrence relation is:

\[
\text{knapsack}(i, w_{\text{max}}) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{knapsack}(i - 1, w_{\text{max}}) & \text{if } w_i > w_{\text{max}} \\
\max \left\{ \begin{array}{ll}
\text{knapsack}(i - 1, w_{\text{max}}) & \text{(discard new item)} \\
\text{knapsack}(i - 1, w_{\text{max}} - w_i) + v_i & \text{(include new item)}
\end{array} \right. & \text{otherwise}
\end{cases}
\]

(19.10)
The 0-1 knapsack problem can be solved using the implementation in algorithm 19.7.

Algorithm 19.7. A method for solving the 0-1 knapsack problem with item values \( v \), integral item weights \( w \), and integral capacity \( w_{\text{max}} \). Recovering the design vector from the cached solutions requires additional iteration.

```
function knapsack(v, w, w_max)
    n = length(v)
    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] : max(y[i-1, j], y[i-1, j-w[i]] + v[i])
        end
    end
    # recover solution
    x, j = falses(n), w_max
    for i in n:-1:1
        if w[i] ≤ j && y[i, j] - y[i-1, j-w[i]] == v[i]
            # the ith element is in the knapsack
            x[i] = true
            j -= w[i]
        end
    end
    return x
end
```

19.6 Ant Colony Optimization

Ant colony optimization\(^5\) is a stochastic method for optimizing paths through graphs. This method was inspired by some ant species that wander randomly in search of food, leaving pheromone trails as they go. Other ants that stumble upon a pheromone trail are likely to start following it, thereby reinforcing the trail’s scent. Pheromones slowly evaporate over time, causing unused trails to fade. Short paths, with stronger pheromones, are traveled more often and thus attract more ants. Thus, short paths create positive feedback that lead other ants to follow and further reinforce the shorter path.

Basic shortest path problems, such as the shortest paths found by ants between the ant hill and sources of food, can be efficiently solved using dynamic pro-

Ant colony optimization has been used to find near-optimal solutions to the traveling salesman problem, a much more difficult problem in which we want to find the shortest path that passes through each node exactly once. Ant colony optimization has also been used to route multiple vehicles, find optimal locations for factories, and fold proteins. The algorithm is stochastic in nature and is thus resistant to changes to the problem over time, such as traffic delays changing effective edge lengths in the graph or networking issues that remove edges entirely.

Ants move stochastically based on the attractiveness of the edges available to them. The attractiveness of transition \(i \rightarrow j\) depends on the pheromone level and an optional prior factor:

\[
A(i \rightarrow j) = \tau(i \rightarrow j)^\alpha \eta(i \rightarrow j)^\beta
\]

(19.11)

where \(\alpha\) and \(\beta\) are exponents for the pheromone level \(\tau\) and prior factor \(\eta\), respectively. For problems involving shortest paths, we can set the prior factor to the inverse edge length \(\ell(i \rightarrow j)\) to encourage the traversal of shorter paths:

\[
\eta(i \rightarrow j) = \frac{1}{\ell(i \rightarrow j)}
\]

A method for computing the edge attractiveness is given in algorithm 19.8.

Suppose an ant is at node \(i\) and can transition to any of the nodes \(j \in J\). The set of successor nodes \(J\) contains all valid outgoing neighbors. Sometimes edges are excluded, such as in the traveling salesman problem where ants are prevented from visiting the same node twice. It follows that \(J\) is dependent on both \(i\) and the ant’s history.

Algorithm 19.8. A method for computing the edge attractiveness table given graph \(\text{graph}\), pheromone levels \(\tau\), prior edge weights \(\eta\), pheromone exponent \(\alpha\), and prior exponent \(\beta\).
The probability of edge transition \( i \rightarrow j \) is:

\[
P(i \rightarrow j) = \frac{A(i \rightarrow j)}{\sum_{j' \in J} A(i \rightarrow j')}
\]  

(19.12)

Ants affect subsequent generations of ants by depositing pheromones. There are several methods for modeling pheromone deposition. A common approach is to deposit pheromones establishing after a complete path.\(^9\) Ants that do not find a path do not deposit pheromones. For shortest path problems, a successful ant that has established a path of length \( \ell \) deposits \( 1/\ell \) pheromones on each edge it traversed.

import StatsBase: Weights, sample
function run_ant(G, lengths, τ, A, x_best, y_best)
x = [1]
    while length(x) < nv(G)
i = x[end]
    neighbors = setdiff(outneighbors(G, i), x)
    if isempty(neighbors) # ant got stuck
        return (x_best, y_best)
    end
    as = [A[(i,j)] for j in neighbors]
push!(x, neighbors[sample(Weights(as))])
end

l = sum(lengths([(x[i-1],x[i]]) for i in 2:length(x)])
for i in 2 : length(x)
    τ[(x[i-1],x[i])] += 1/l
end
if l < y_best
    return (x, l)
else
    return (x_best, y_best)
end

Algorithm 19.9. A method for simulating a single ant on a traveling salesman problem in which the ant starts at the first node and attempts to visit each node exactly once. Pheromone levels are increased at the end of a successful tour. The parameters are the graph \( G \), edge lengths \( \text{lengths} \), pheromone levels \( \tau \), edge attractiveness \( A \), the best solution found thus far \( x_{\text{best}} \), and its value \( y_{\text{best}} \).

Ant colony optimization also models pheromone evaporation, which naturally occurs in the real world. Modeling evaporation helps prevent the algorithm from prematurely converging to a single, potentially suboptimal, solution. Pheromone evaporation is executed at the end of each iteration after all ant simulations have

been completed. Evaporation decreases the pheromone level of each transition by a factor of $1 - \rho$, with $\rho \in [0, 1]$.\footnote{It is common to use $\rho = 1/2$.}

For $m$ ants at iteration $k$, the effective pheromone update is

$$
\tau(i \rightarrow j)_{(k+1)} = (1 - \rho)\tau(i \rightarrow j)_{(k)} + \sum_{a=1}^{m} \frac{1}{\ell^{(a)}} \left((i \rightarrow j) \in \mathcal{P}^{(a)}\right) \quad (19.13)
$$

where $\ell^{(a)}$ is the path length and $\mathcal{P}^{(a)}$ is the set of edges traversed by ant $a$.

Ant colony optimization is implemented in algorithm 19.10, with individual ant simulations using algorithm 19.9. Figure 19.7 visualizes ant colony optimization used to solve a traveling salesman problem.

Algorithm 19.10. Ant colony optimization, which takes a directed or undirected graph $G$ from LightGraphs.jl and a dictionary of edge tuples to path lengths $\text{lengths}$. Ants start at the first node in the graph. Optional parameters include the number of ants per iteration $m$, the number of iterations $k_{\text{max}}$, the pheromone exponent $\alpha$, the prior exponent $\beta$, the evaporation scalar $\rho$, and a dictionary of prior edge weights $\eta$. $\text{lengths}$. Ants start at the first node in the graph. Optional parameters include the number of ants per iteration $m$, the number of iterations $k_{\text{max}}$, the pheromone exponent $\alpha$, the prior exponent $\beta$, the evaporation scalar $\rho$, and a dictionary of prior edge weights $\eta$.

Figure 19.7. Ant colony optimization used to solve a traveling salesman problem on a directed graph using 50 ants per iteration. Path lengths are the Euclidean distances. Color opacity corresponds to pheromone level.
19.7 Summary

- Discrete optimization problems require that the design variables be chosen from discrete sets.
- Relaxation, in which the continuous version of the discrete problem is solved, is by itself an unreliable technique for finding an optimal discrete solution but is central to more sophisticated algorithms.
- Many combinatorial optimization problems can be framed as an integer program, which is a linear program with integer constraints.
- Both the cutting plane and branch and bound methods can be used to solve integer programs efficiently and exactly. The branch and bound method is quite general and can be applied to a wide variety of discrete optimization problems.
- Dynamic programming is a powerful technique that exploits optimal overlapping substructure in some problems.
- Ant colony optimization is a nature-inspired algorithm that can be used for optimizing paths in graphs.

19.8 Exercises

Exercise 19.1. A Boolean satisfiability problem, often abbreviated SAT, requires determining whether a Boolean design exists that causes a Boolean-valued objective function to output true. SAT problems were the first to be proven to belong to the difficult class of NP-complete problems. This means that SAT is at least as difficult as all other problems whose solutions can be verified in polynomial time.

Consider the Boolean objective function:

\[ f(x) = x_1 \land (x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2) \]

Find an optimal design using enumeration. How many designs must be considered for an \( n \)-dimensional design vector in the worst case?

Exercise 19.2. Formulate the problem in exercise 19.1 as an integer linear program. Can any Boolean satisfiability problem be formulated as an integer linear program?

Exercise 19.3. Why are we interested in totally unimodular matrices? Furthermore, why does every totally unimodular matrix contain only entries that are 0, 1, or −1?

Exercise 19.4. This chapter solved the 0-1 knapsack problem using dynamic programming. Show how to apply branch and bound to the 0-1 knapsack problem, and use your approach to solve the knapsack problem with values \( v = [9, 4, 2, 3, 5, 3] \), and weights \( w = [7, 8, 4, 5, 9, 4] \) with capacity \( w_{\text{max}} = 20 \).