20.1 THE METHOD

In dynamic programming, as in the greedy method, we view the solution to a problem as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequences. Some examples that illustrate this point are given below.

Example 20.1 [Shortest Path] Consider the digraph of Figure 18.2. We wish to find a shortest path from the source vertex \( s = 1 \) to the destination vertex \( d = 5 \). We need to make decisions on the intermediate vertices. The choices for the first decision are 2, 3, and 4. That is, from vertex 1 we may move to any one of these vertices. Suppose we decide to move to vertex 3. Now we need to decide on how to get from 3 to 5. If we go from 3 to 5 in a suboptimal way, then the 1-to-5 path constructed cannot be optimal, even under the restriction that from vertex 1 we must go to vertex 3. For example, if we use the suboptimal path 3, 2, 5 with length 9, the constructed 1-to-5 path 1, 3, 2, 5 has length 11. Replacing the suboptimal path 3, 2, 5 with an optimal one 3, 4, 5 results in the path 1, 3, 4, 5 of length 9.

For this shortest-path problem, suppose that our first decision gets us to some vertex \( v \). Although we do not know how to make this first decision we do know that the remaining decisions must be optimal for the problem of going from \( v \) to \( d \).

Example 20.2 [0/1 Knapsack Problem] Consider the 0/1 knapsack problem of Section 18.3.2. We need to make decisions on the values of \( x_1, \ldots, x_n \). Suppose we are deciding the values of the \( x_i \)'s in the order \( i = 1, 2, \ldots, n \). If we set \( x_1 = 0 \), then the available knapsack capacity for the remaining objects (i.e., objects \( 2, 3, \ldots, n \)) is \( c \). If we set \( x_1 = 1 \), the available knapsack capacity is \( c - w_1 \). Let \( r \in \{ c, c - w_1 \} \) denote the remaining knapsack capacity.

Following the first decision, we are left with the problem of filling a knapsack with capacity \( r \). The available objects (i.e., 2 through \( n \)) and the available capacity \( r \) define the problem state following the first decision. Regardless of whether \( x_1 \) is 0 or 1, \( [x_2, \ldots, x_n] \) must be an optimal solution for the problem state following the first decision. If not, there is a solution \([y_2, \ldots, y_n]\) that provides greater profit for the problem state following the first decision. So \([x_1, y_2, \ldots, y_n]\) is a better solution for the initial problem.

Suppose that \( n = 3 \), \( w = [100, 14, 10] \), \( p = [20, 18, 15] \), and \( c = 116 \). If we set \( x_1 = 1 \), then following this decision, the available knapsack capacity is 16. \([x_2, x_3] = [0, 1] \) is a feasible solution to the two-object problem that remains. It returns a profit of 15. However, it is not an optimal solution to the remaining two-object problem, as \([x_2, x_3] = [1, 0] \) is feasible and returns a greater profit of 18. So \( x = [1, 0, 1] \) can be improved to \([1, 1, 0] \). If we set \( x_1 = 0 \), the available capacity for the two-object instance that remains is 116. If the subsequence \([x_2, x_3]\) is not an
optimal solution for this remaining instance, then \([x_1, x_2, x_3]\) cannot be optimal for the initial instance.

Example 20.3 [Airfares] A certain airline has the following airfare structure: From Atlanta to New York or Chicago, or from Los Angeles to Atlanta, the fare is \$100; from Chicago to New York, it is \$20; and for passengers connecting through Atlanta, the Atlanta to Chicago segment is only \$20. A routing from Los Angeles to New York involves decisions on the intermediate airports. If problem states are encoded as (origin, destination) pairs, then following a decision to go from Los Angeles to Atlanta, the problem state is *We are at Atlanta and need to get to New York.* The cheapest way to go from Atlanta to New York is a direct flight with cost \$100. Using this direct flight results in a total Los Angeles-to-New York cost of \$200. However, the cheapest routing is Los Angeles–Atlanta–Chicago–New York with a cost of \$140, which involves using a suboptimal decision subsequence for the Atlanta–New York problem (Atlanta–Chicago–New York).

If instead we encode the problem state as a triple \((tag, origin, destination)\) where \(tag\) is 0 for connecting flights and 1 for all others, then once we reach Atlanta, the state becomes \((0, Atlanta, New York)\) for which the optimal routing is through Chicago.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations*, that enable us to solve the problem in an efficient way.

Example 20.4 [0/1 Knapsack] In Example 20.2 we saw that for the 0/1 knapsack problem, optimal decision sequences were composed of optimal subsequences. Let \(f(i, y)\) denote the value of an optimal solution to the knapsack instance with remaining capacity \(y\) and remaining objects \(i, i + 1, \ldots, n\). From Example 20.2 it follows that

\[
f(n, y) = \begin{cases} 
  p_n & y \geq w_n \\
  0 & 0 \leq y < w_n 
\end{cases}
\]  

and

\[
f(i, y) = \begin{cases} 
  \max\{f(i + 1, y), f(i + 1, y - w_i) + p_i\} & y \geq w_i \\
  f(i + 1, y) & 0 \leq y < w_i 
\end{cases}
\]

By making use of the observation that optimal decision sequences are made up of optimal subsequences, we have obtained a recurrence for \(f\). \(f(1, c)\) is the value of the optimal solution to the knapsack problem we started with. Equation 20.2 may be used to determine \(f(1, c)\) either recursively or iteratively. In the iterative approach, we start with \(f(n, *)\), as given by Equation 20.1, and then obtain \(f(i, *)\)
in the order \( i = n - 1, n - 2, \ldots, 2 \), using Equation 20.2. Finally, \( f(1, c) \) is computed by using Equation 20.2.

For the instance of Example 20.2, we see that \( f(3, y) = 0 \) if \( 0 \leq y < 10 \), and 15 if \( y \geq 10 \). Using Equation 20.2, we obtain \( f(2, y) = 0 \) if \( 0 \leq y < 10 \), 15 if \( 10 \leq y < 14 \), 18 if \( 14 \leq y < 24 \), and 33 if \( y \geq 24 \). The optimal solution has value
\[
f(1, 116) = \max\{f(2, 116), f(2, 116 - w_1) + p_1\} = \max\{f(2, 116), f(2, 16) + 20\} = \max\{33, 38\} = 38.
\]

To obtain the values of the \( x_i \)'s, we proceed as follows: If \( f(1, c) = f(2, c) \), then we may set \( x_1 = 0 \) because we can utilize the \( c \) units of capacity getting a return of \( f(1, c) \) from objects 2, \( \ldots, n \). In case \( f(1, c) \neq f(2, c) \), then we must set \( x_1 = 1 \). Next we need to find an optimal solution that uses the remaining capacity \( c - w_1 \). This solution has value \( f(2, c - w_1) \). Proceeding in this way, we may determine the value of all the \( x_i \)'s.

For our sample instance we see that \( f(2, 116) = 33 \neq f(1, 116) \). Therefore, \( x_1 = 1 \), and we need to find \( x_2 \) and \( x_3 \) so as to obtain a return of 38 - \( p_1 = 18 \) and use a capacity of at most \( 116 - w_1 = 16 \). Note that \( f(2, 16) = 18 \). Since \( f(3, 16) = 15 \neq f(2, 16) \), \( x_2 = 1 \); the remaining capacity is \( 16 - w_2 = 2 \). Since \( f(3, 2) = 0 \), we set \( x_3 = 0 \).

The principle of optimality states that no matter what the first decision, the remaining decisions must be optimal with respect to the state that results from this first decision. This principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic-programming solution are

- Verify that the principle of optimality holds.
- Set up the dynamic-programming recurrence equations.
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a traceback step in which the solution itself is constructed.

It is very tempting to write a simple recursive program to solve the dynamic-programming recurrence. However, as we will see in subsequent sections, unless care is taken to avoid recomputing previously computed values, the recursive program will have prohibitive complexity. When the recursive program is designed to avoid this recomputation, the complexity is drastically reduced. The dynamic-programming recurrence may also be solved by iterative code that naturally avoids recomputation of already computed values. Although this iterative code has the same time complexity as the “careful” recursive code, the former has the advantage of not requiring additional space for the recursion stack. As a result, the iterative code generally runs faster than the careful recursive code.
20.2 APPLICATIONS

20.2.1 0/1 Knapsack Problem

Recursive Solution

The dynamic-programming recurrence equations for the 0/1 knapsack problem were developed in Example 20.4. A natural way to solve a recurrence such as Equation 20.2 for the value \( f(1, c) \) of an optimal knapsack packing is by a recursive program such as Program 20.1. The invocation \( \text{knapsack}(p, w, c) \) returns the value of \( f(1, c) \). Although Program 20.1 is written for integer profits and weights, by changing the data type of theProfit and/or theWeight in the knapsack method header, we can obtain code for any desired primitive data type (e.g., long, float, and double).

Let \( t(n) \) be the time this code takes to solve an instance with \( n \) objects. We see that \( t(1) = a \) and \( t(n) \leq 2t(n-1) + b \) for \( n > 1 \). Here \( a \) and \( b \) are constants. This recurrence solves to \( t(n) = O(2^n) \).

Example 20.5 Consider the case \( n = 5 \), \( p = [6, 3, 5, 4, 6] \), \( w = [2, 2, 6, 5, 4] \), and \( c = 10 \). To determine \( f(1, 10) \), method \( f \) is invoked as \( f(1, 10) \). The recursive calls made are shown by the tree of Figure 20.1. Each node has been labeled by the value of \( y \). Nodes on level \( j \) have \( i = j \). So the root denotes the invocation \( f(1, 10) \). Its left and right children, respectively, denote the invocations \( f(2, 10) \) and \( f(2, 8) \). In all, 28 invocations are made. Notice that several invocations redo the work of previous invocations. For example, \( f(3, 8) \) is computed twice, as are \( f(4, 8) \), \( f(4, 6) \), \( f(4, 2) \), \( f(5, 8) \), \( f(5, 6) \), \( f(5, 3) \), \( f(5, 2) \), and \( f(5, 1) \). If we save the results of previous invocations, we can reduce the number of invocations to 19 because we eliminate the shaded nodes of Figure 20.1.

Recursive Solution without Recomputations

As observed in Example 20.5, Program 20.1 is doing more work than necessary. To avoid recomputing the same \( f(i, y) \) value, we may keep a list of \( f(i, y) \)s that have already been computed. The elements of this list are triples of the form \( (i, y, f(i, y)) \). Before making an invocation \( f(i, y) \), we see whether the list contains a triple of the form \( (i, y, *) \) where * denotes a wildcard. If so, \( f(i, y) \) is retrieved from the list. If not, the invocation is made and then the triple \( (i, y, f(i, y)) \) added to the list. The list of triples may be implemented as a hash table (see Section 11.5) or as a binary search tree (see Chapter 16).

When the weights are integer, we may use an integer array \( \text{fArray}[i][y] \) such that \( f[i][y] \) equals \(-1 \) iff \( f(i, y) \) has not been computed before. Program 20.2 gives the recursive code that avoids recomputation of previously computed \( f \) values. This code assumes that \( \text{fArray} \) is an \((n + 1) \times (c + 1) \) integer array, which is a class data member that has been initialized to \(-1 \) by the public method \( \text{knapsack} \), which invokes method \( f \).
public class RecursiveDPKnapsack {
    static int [] profit;
    static int [] weight;
    static int numberOfObjects;

    public static int knapsack(int [] theProfit, int [] theWeight,
                                int knapsackCapacity)
    {
        profit = theProfit;
        weight = theWeight;
        numberOfObjects = theProfit.length - 1;
        return f(1, knapsackCapacity);
    }

    /** recursive method to solve dynamic programming recurrence
    * @return f(i, theCapacity) */
    private static int f(int i, int theCapacity)
    {
        if (i == numberOfObjects)
            return (theCapacity < weight[numberOfObjects])
                ? 0 : profit[numberOfObjects];
        if (theCapacity < weight[i])
            return f(i + 1, theCapacity);
        return Math.max(f(i + 1, theCapacity),
                        f(i + 1, theCapacity - weight[i]) + profit[i]);
    }
}

Program 20.1 Recursive method for knapsack problem

To determine the time complexity of Program 20.2, we will use an accounting
scheme in which we charge different components of the total time to different f(i, y)s
and then add up the amounts charged to each f(i, y). When computing an f(i, y),
the cost of an invocation f(i+1, z) is charged to f(i + 1, z) if f(i + 1, z) has not
been computed and to f(i, z) otherwise. (This f(i + 1, z) in turn offloads the cost
of computing new f(*, *)s to the individual f(*, *)s that are computed.) The cost
of the remainder of Program 20.2 is charged to f(i, y). This remaining cost is Θ(1).
The total amount charged to each f(i, y) is constant, and the number of f(i, y)s
is (c + 1)(n + 1). Therefore, the total time is O(cn) (recall that c denotes the
knapsack capacity and n is the number of objects). By avoiding the recomputation,
of previously computed \( f(i, y) \)s, we have cut the running time of the recursive code from an impractical \( O(2^n) \) to a very practical \( O(cn) \)!

**Iterative Solution with Integer Weights**

We can devise a fairly simple iterative algorithm (Program 20.3) to solve for \( f(1, c) \) when the weights are integer. This algorithm, based on the strategy outlined in Example 20.4, computes each \( f(i, y) \) exactly once. Program 20.3 uses a two-dimensional array \( f[i][j] \) to store the values of the function \( f \). Program 20.4 gives the code for the traceback needed to determine the \( x_i \) values that result in the optimal filling.

**Example 20.6** Figure 20.2 shows the \( f \) array computed by Program 20.3 using the data of Example 20.5. The data are computed by rows from top to bottom and within a row from left to right. The value \( f[1][10] = 15 \) is not shown.

To determine the \( x_i \) values, we begin with \( x_1 \). Since \( f(1, 10) \neq f(2, 10) \), \( f(1, 10) \) must equal \( f(2, 10 - w_1) + p_1 = f(2, 8) + 6 \). Therefore, \( x_1 = 1 \). Since \( f(2, 8) \neq f(3, 8) \), \( f(2, 8) \) must equal \( f(3, 8 - w_2) + p_2 = f(3, 6) + 3 \) and \( x_2 = 1 \). \( x_3 = x_4 = 0 \) because \( f(3, 6) = f(4, 6) = f(5, 6) \). Finally, since \( f(5, 4) \neq 0 \), \( x_5 = 1 \).

The complexity of the iterative knapsack code is \( \Theta(nc) \) and that of traceback is \( \Theta(n) \).
private static int f(int i, int theCapacity)
{
    // check if already computed
    if (fArray[i][theCapacity] >= 0)
        return fArray[i][theCapacity];

    // not yet computed
    if (i == numberOfObjects)
        // use Equation 20.1
        // compute and save f(i, theCapacity)
        fArray[i][theCapacity] = (theCapacity < weight[numberOfObjects])
                                ? 0 : profit[numberOfObjects];

        return fArray[i][theCapacity];
    }
    // use Equation 20.2
    if (theCapacity < weight[i])
        // object i does not fit
        fArray[i][theCapacity] = f(i + 1, theCapacity);
    else
        // object i fits, try both possibilities
        fArray[i][theCapacity] = Math.max(f(i + 1, theCapacity),
                                         f(i + 1, theCapacity - weight[i] + profit[i]));

    return fArray[i][theCapacity];
}

Program 20.2 Recursive method for knapsack problem with no recomputations

\[
\begin{array}{|c|cccccccccc|}
\hline
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
5 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
4 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 10 & 10 \\
3 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 10 & 10 & 11 \\
2 & 0 & 0 & 3 & 3 & 6 & 6 & 9 & 9 & 9 & 10 & 11 \\
\hline
\end{array}
\]

Figure 20.2 \( f \) function/array for Example 20.6
public static void knapsack(int[] profit, int[] weight, int knapsackCapacity, int[][] f)
{
    int numberOfObjects = profit.length - 1;

    // initialize f[numberOfObjects][]
    int yMax = Math.min(weight[numberOfObjects] - 1, knapsackCapacity);
    for (int y = 0; y <= yMax; y++)
        f[numberOfObjects][y] = 0;
    for (int y = weight[numberOfObjects]; y <= knapsackCapacity; y++)
        f[numberOfObjects][y] = profit[numberOfObjects];

    // compute f[i][y], 1 < i < numberOfObjects
    for (int i = numberOfObjects - 1; i > 1; i--)
    {
        yMax = Math.min(weight[i] - 1, knapsackCapacity);
        for (int y = 0; y <= yMax; y++)
            f[i][y] = f[i + 1][y];
        for (int y = weight[i]; y <= knapsackCapacity; y++)
            f[i][y] = Math.max(f[i + 1][y],
                               f[i + 1][y - weight[i]] + profit[i]);
    }

    // compute f[1][knapsackCapacity]
    f[1][knapsackCapacity] = f[2][knapsackCapacity];
    if (knapsackCapacity >= weight[1])
    {
        f[1][knapsackCapacity] = Math.max(f[1][knapsackCapacity],
                                           f[2][knapsackCapacity - weight[1]] + profit[1]);
    }
}

Program 20.3 Iterative computation of f

Tuple Method (Optional)

The code of Program 20.3 has two drawbacks. First, it requires that the weights be integer. Second, it is slower than Program 20.1 when the knapsack capacity is large. In particular, if \(c > 2^n\), its complexity is \(\Omega(n2^n)\). We can overcome both of these shortcomings by using a tuple approach in which for each \(i\), \(f(i,y)\) is stored as an ordered list \(P(i)\) of pairs \((y, f(i,y))\) that correspond to the \(y\) values at which the function \(f\) changes. The pairs in each \(P(i)\) are in increasing order of \(y\). In addition, since \(f(i,y)\) is a nondecreasing function of \(y\), the pairs are also in increasing order of \(f(i,y)\).
```java
public static void traceback(int [][] f, int [] weight,
   int knapsackCapacity, int [] x)
{
    int numberOfObjects = weight.length - 1;

    for (int i = 1; i < numberOfObjects - 1; i++)
    {
      if (f[i][knapsackCapacity] == f[i+1][knapsackCapacity])
        // do not include object i
        x[i] = 0;
      else
        // include object i
        x[i] = 1;
        knapsackCapacity -= weight[i];
    }

    x[numberOfObjects] = (f[numberOfObjects][knapsackCapacity] > 0)
      ? 1 : 0;
}
```

**Program 20.4 Iterative computation of x**

**Example 20.7** Consider the $f$ function of Figure 20.2. When $i = 5$, the function $f$ is completely specified by the pairs $P(5) = [(0, 0), (4, 6)]$. The pairs $P(i)$ for $i = 4, 3,$ and $2$ are $[(0, 0), (4, 6), (9, 10)], [(0, 0), (4, 6), (9, 10), (10, 11)]$, and $[(0, 0), (2, 3), (4, 6), (6, 9), (9, 10), (10, 11)]$.

To compute $f(1, 10)$, we use Equation 20.2, which yields $f(1, 10) = \max\{f(2, 10), f(2, 8) + p_1\}$. From $P(2)$ we get $f(2, 10) = 11$, and $f(2, 8) = 9$ ($f(2, 8) = 9$ comes from the pair $(6, 9)$). Therefore, $f(1, 10) = \max\{11, 15\} = 15$.

To determine the $x_i$ values, we begin with $x_1$. Since $f(1, 10) = f(2, 6) + p_1$, $x_1 = 1$. Since $f(2, 6) = f(3, 6 - w_2) + p_2 = f(3, 4) + p_2$, $x_2 = 1$. $x_3 = x_4 = 0$ because $f(3, 4) = f(4, 4) = f(5, 4)$. Finally, since $f(5, 4) \neq 0$, $x_5 = 1$.

If we examine the pairs in each $P(i)$, we see that each pair $(y, f(i, y))$ corresponds to a different combination of 0/1 assignments to the variables $x_i, \cdots, x_n$. Let $(a, b)$ and $(c, d)$ be pairs that correspond to two different 0/1 assignments to $x_i, \cdots, x_n$. If $a \geq c$ and $b < d$, then $(a, b)$ is dominated by $(b, c)$. Dominated assignments do not contribute pairs to $P(i)$. If two or more assignments result in the same pair, only one is in $P(i)$.

Under the assumption that $w_n \leq c$, $P(n) = [(0, 0), (w_n, p_n)]$. These two pairs correspond to $x_n$ equal to 0 and 1, respectively. For each $i$, $P(i)$ may be obtained from $P(i + 1)$. First, compute the ordered set of pairs $Q$ such that $(s, t)$ is a pair of $Q$ if $w_i \leq s \leq c$ and $(s - w_i, t - p_i)$ is a pair of $P(i + 1)$. Now $Q$ has the pairs
with $x_i = 1$, and $P(i + 1)$ has those with $x_i = 0$. Second, merge $Q$ and $P(i + 1)$ eliminating dominated as well as duplicate pairs to get $P(i)$.

**Example 20.8** Consider the data of Example 20.7. $P(5) = [(0, 0), (4, 6)]$, so $Q = [(5, 4), (9, 10)]$. When merging $P(5)$ and $Q$ to create $P(4)$, the pair $(5, 4)$ is eliminated because it is dominated by the pair $(4, 6)$. As a result, $P(4) = [(0, 0), (4, 6), (9, 10)]$. To compute $P(3)$, we first obtain $Q = [(6, 5), (10, 11)]$ from $P(4)$. Next merging with $P(4)$ yields $P(3) = [(0, 0), (4, 6), (9, 10), (10, 11)]$. Finally, to get $P(2)$, $Q = [(2, 3), (6, 9)]$ is computed from $P(3)$. Merging $P(3)$ and $Q$ yields $P(2) = [(0, 0), (2, 3), (4, 6), (6, 9), (9, 10), (10, 11)]$.

Since the pairs in each $P(i)$ represent different 0/1 assignments to $x_1, \ldots, x_n$, no $P(i)$ has more than $2^{n-i+1}$ pairs. When computing $P(i)$, $Q$ may be computed in $\Theta(|P(i + 1)|)$ time. The time needed to merge $P(i + 1)$ and $Q$ is also $\Theta(|P(i + 1)|)$. So all the $P(i)$s may be computed in $\Theta(\sum_{i=2}^{n} |P(i + 1)|) = O(2^n)$ time. When the weights are integer, $|P(i)| \leq c + 1$. In this case the complexity becomes $O(\min\{nc, 2^n\})$.

### 20.2.2 Matrix Multiplication Chains

**Problem Description**

An $m \times n$ matrix $A$ and an $n \times p$ matrix $B$ can be multiplied in $\Theta(mnp)$ time (see Exercise 2.24). We will use $mnp$ as a measure of the time needed to multiply the two matrices. Suppose we are to multiply three matrices $A$, $B$, and $C$. There are two ways in which we can accomplish this task. In the first, we multiply $A$ and $B$ to get the product matrix $D$ and then multiply $D$ and $C$ to get the desired result. This multiplication order can be written as $(A \times B) \times C$. The second way is $A \times (B \times C)$. Although both multiplication orders obtain the same result, one may take a lot more computing time than the other.

**Example 20.9** Suppose that $A$ is a $100 \times 1$ matrix, $B$ is a $1 \times 100$ matrix, and $C$ is a $100 \times 1$ matrix. Then the time needed to compute $A \times B$ is 10,000. Since the result is a $100 \times 100$ matrix, the time needed to perform the multiplication with $C$ is 1,000,000. The overall time needed to compute $(A \times B) \times C$ is therefore 1,010,000. $B \times C$ can be computed in 10,000 units of time. Since the result is a $1 \times 1$ matrix, the time needed for the multiplication with $A$ is 100. The total time needed to compute $A \times (B \times C)$ is therefore 10,100! Furthermore, when computing $(A \times B) \times C$, we need 10,000 units of space to store $A \times B$; however, when $A \times (B \times C)$ is computed, only one unit of space is needed for $B \times C$.

As an example of a real problem that can benefit from computing the matrix product $A \times B \times C$ in the proper order, consider the registration of 2 three-dimensional images. In the registration problem, we are to determine the amount by which one image needs to be rotated, translated, and shrunk (or expanded) so that it approximates the second. One way to perform this registration involves doing about...