Programming and Randomized

Algorithms

In the preceding chapters, we covered some fundamental data structures and algorithms. The current chapter extends static algorithm (deterministic algorithm) concepts to randomized algorithms. Deterministic algorithms use polynomials of the size of the input, whereas random algorithms use random sources as input and make their own choices. The chapter will introduce the Las Vegas and the Monte Carlo randomized algorithms and their application using examples. The chapter will also introduce skip list and its extended version, randomized skip list, which uses randomization concepts to reduce the computation effort in an average case scenario. We will start with the fundamentals of programming, which can be used to reduce computational effort in intensive tasks. The current chapter will cover the concepts of dynamic programming and directed acyclic graphs ( DAGs ). The current chapter will cover following topics:

Dynamic programming The knapsack problem All-pairs shortest paths Randomized algorithms

Randomized algorithms for finding large values Skip lists

Page 239

﻿

Programming and Randomized Algorithms

Dynamic programming

Dynamic programming can be defined as an approach that uses a recurrent formula (breaking a problem into a subproblem) to solve any complex problem. A sub-solution of any problem is reconstructed using the previous state solution. Dynamic programming- based approaches are able to achieve a polynomial complexity for solving problems, and assure faster computation than other classical approaches, such as brute force algorithms. Before we get into dynamic programming, let's cover the basics of DAG, as it will help with implementation of dynamic programming. DAGs are directed acyclic topological graphs, which are defined by a number of vertices and edges such that every edge is directed from earlier to later in the sequence. An example of a DAG is shown in Figure 9.1 :

Figure 9.1: An example of DAG

Let's assume that the vertices represent cities and edges represent the path to be followed to reach a particular city. The objective is to determine the shortest path to node D from root node A in the preceding figure. This can be represented as follows:

Figure 9.2: Another representation of Figure 9.1, with the objective to reach node D from the root node, A

Let d(i,j) represent the distance from the i th vertex to the j th vertex, so in the example, d(A, B) represents the distance from node A to node B. Also, mdist(k) represents the shortest distance to the k th vertex. The minimum distance to D can be written as follows: Shortest\_Distance(A, D) = min{ mdist(B)+ d(B, D), mdist(D)+d(C, D)}

[ 228 ]

Page 240

﻿

Chapter 9

Similarly, mdist(k) can be further broken down into subproblems. DAG is implicit in dynamic programming, where each vertex acts a subproblem and an edge represents the dependency between subproblems. The approach of dynamic programming is very different from recursion. Let's consider an example for writing a function to calculate the n th Fibonacci series. The Fibonacci series is a sequence of integers such that every number is a sum of the preceding two numbers: {1, 1, 2, 3, 5, 8, … } . The function for evaluating the n th Fibonacci number can be written using recursion in R, as shown here:

nfib<-function(n){

assertthat::assert\_that(n>0) & assertthat::assert\_that(n<50) if(n==1 || n==2) return(1) val<- nfib(n-1) + nfib(n-2) return(val)

}

Let's look at how the recursive approach processes the computation of the n th Fibonacci number. Figure 9.3 shows how the preceding recursive function nfibonacci will evaluate the sixth Fibonacci number:

Figure 9.3: Number of times the nﬁbonacci function is called to calculate the sixth Fibonacci number

[ 229 ]

Page 241

﻿

Programming and Randomized Algorithms

From Figure 9.3 it is evident that in the recursive approach, the lower values are computed multiple times thus making the algorithm sub-optimal; however, computational time increases exponentially due to repeated computation of the same values, as shown in Figure 9.4 :

Figure 9.4: Computation time required to calculate the n th Fibonacci number using the recursive solution

The other approach to compute the n th Fibonacci number can be decided based on DAG. The Fibonacci DAG can be represented as shown in Figure. 9.5 .

Figure 9.5: DAG representation for generating the Fibonacci series

Figure 9.5 shows that the n th Fibonacci number depends on the last two lagged values, so we can make the computation linear by storing the last two values, as shown in the following R script:

nfib\_DP<-function(n){

assertthat::assert\_that(n>0) & assertthat::assert\_that(n<50) if(n<=2) return(1) lag2\_val<-0

[ 230 ]

Page 242

﻿

Chapter 9

lag1\_val<-1 nfibval<-1

for(i in 3:n){

lag2\_val<-lag1\_val lag1\_val<-nfibval

nfibval<-lag2\_val+lag1\_val

}

return(nfibval)

}

The approach is very powerful for solving complex computations when the problem can be split into subproblems, which need to be solved repeatedly. The next sub-section will discuss the knapsack problem, which has many different variations, and how dynamic programming is used to address the problem.

The knapsack problem

The knapsack problem is a combinatorial optimization problem, which requires a subset of some given item to be chosen such that profit is maximized without exceeding capacity constraints. There are different types of knapsack problems reported in the literature depending on the number of items and knapsacks available: the 0-1 knapsack problem , where each item is chosen at most once; the bounded knapsack problem puts a constraint on selection of each item; the multiple choice knapsack problem, where multiple knapsacks are presents and items are to be chosen from multiple sets; and the multi-constraint knapsack problem, where more than one constraint is present – such a knapsack having constraints on volume and weight.

The current section will discuss the 0-1 knapsack problem formulation and propose a solution can be addressed using dynamic programming. Let's consider an example storing a data file, n , with W total storage capacity available. Suppose F is a set of files to be stored where F = {f 1 , f 2 , … , f n } , S={s 1 , s 2 , … , s n } is a set of the amount of storage required by the i th file, and C is the computational effort required to get the files, and can be represented as C={c 1 , c 2 , … , c n } . The objective is to select files so that it minimizes the wastage of storage capacity S , and to maximize the computing time for stored files so that the minimum time is required to recompute the files that are not stored on disk. This is a 0-1 knapsack problem, so a partial file cannot be stored.

[ 231 ]

Page 243

﻿

Programming and Randomized Algorithms

The problem can be written mathematically as follows:

The aforementioned problem can be solved using dynamic programming, which stores the subproblem solution in a table that can be reused repeatedly while searching for an optimal solution. However, it will require more space to store the results of the subproblem. The implementation of this problem using dynamic programming is as follows:

knapsack\_DP<-function(W, S, C, n){

require(pracma)

K<-zeros(n+1, W+1) for(i in 1:(n+1)){ for(j in 1:(W+1)){

if(i==1 | j==1){

K[i,j]=0

} else if(S[i-1] <= j){

K[i, j] = max(C[i-1] + K[i-1, (j-S[i-1])], K[i-1, j])

} else {

K[i, j] = K[i-1, j]

} } }

return(K[n+1, W+1])

}

The problem requires us to search for the solution in a two-dimension computation of the file and the storage required. The matrix K stores intermediate results, which can be reused to satisfy the objective under given constraints.

All pairs shortest paths

The All Pairs Shortest Path ( APSP ) problem focuses on finding the shortest path between all pairs of vertices. Let's consider a directed graph G(V, E) , where for each edge , the distance d(u, v) is associated if the edges u and v are connected. For example, d(A, B) = 8 units, as shown in Figure 9.6 :

[ 232 ]

Page 244

﻿

Chapter 9

Figure 9.6: DAG representation for generating Fibonacci series

The APSP algorithm will determine the shortest path to reach from one edge to the other. For example, the shortest path to reach edge B from A is A -> E -> B with a distance of 4 units. The d(u, v) in graph G is defined as follows:

One of the approaches to solve the APSP problem is by using the Floyd-Warshall algorithm, which uses dynamic programming. The approach is based on the observation that any path linking two vertices u and v may have zero or more vertices between them, defining a path. The R implementation of the Floyd-Warshall algorithm is as follows:

# Implementation of Floyd-Warshall algorithm floydWarshall<-function(graph){

nodes<-names(graph) dist<-graph

for (n in nodes){ for(ni in nodes){ for(nj in nodes){

if((dist[[ni]][n]+dist[[n]][nj])<dist[[ni]][nj]){

dist[[ni]][nj]<-dist[[ni]][n]+dist[[n]][nj]

} } } }

return(dist)

}

[ 233 ]

Page 245

﻿

Programming and Randomized Algorithms

The implementation begins by disallowing all intermediate vertices, thus, the initial solution is simply an initial distance matrix achieved by assigning graph to the dist list. The algorithm then proceeds by introducing an additional intermediate vertex at each step and selecting the shortest path by comparing it with the previous best estimate. The approach breaks the problem into subproblems, as the shortest distance d(u, v) between the vertices u and v , passing through vertex k , is the sum of the shortest distance d(u, k) and d(k, v) . The Floyd-Warshall implementation requires a computational effort of O(n 3 ) . The APSP problem output for the graph shown in Table 9.1 can be determined using the following example script:

# Defining graph structure graph<-list()

graph[["A"]]=c("A"=0, "B"=8, "C"=Inf, "D"=Inf, "E"=1, "F"=Inf) graph[["B"]]=c("A"=Inf, "B"=0, "C"=7, "D"=6, "E"=Inf, "F"=Inf) graph[["C"]]=c("A"=Inf, "B"=Inf, "C"=0, "D"=6, "E"=Inf, "F"=Inf) graph[["D"]]=c("A"=Inf, "B"=Inf, "C"=Inf, "D"=0, "E"=Inf, "F"=4) graph[["E"]]=c("A"=Inf, "B"=3, "C"=Inf, "D"=Inf, "E"=0, "F"=9) graph[["F"]]=c("A"=Inf, "B"=3, "C"=Inf, "D"=4, "E"=9, "F"=0) APSP\_Dist<-floydWarshall(graph) # get shortest pair distance

The graph is stored as a list in R, which can be called using edge name. Similarly, the outcome is returned as a list object. The output from the Floyd-Warshall algorithm is shown in Table 9.1 :

Table 9.1: Output from the Floyd-Warshall algorithm as the shortest distance between nodes

The Inf in the preceding table shows that there is no direct or indirect connection between the two nodes.

[ 234 ]

Page 246

﻿

Chapter 9

Randomized algorithms

In scenarios where computations are very expensive, introduction of randomness can help reduce computational effort at the expense of accuracy. The algorithms can be classified into the following and are depicted in Figure 9.7 :

Deterministic algorithms Randomized algorithm

Figure 9.7: Diﬀerent types of algorithm structures

Deterministic algorithms solve the problem correctly where computational effort required is a polynomial of the size of the input, whereas random algorithms take random sources as input and make their own choices while executing.

Randomized algorithms for finding large values

The computational cost of finding the largest value in an unsorted list is O(n) . Any

deterministic algorithm will require O(n) effort to determine the maximum value. However, in scenarios where time is essence, and n is very large, approximation algorithms are used, which, instead of finding the actual solution, determine the solution that is closer to the actual solution. Randomized algorithms can be classified into the following based on their goals:

Las Vegas algorithm : The Las Vegas algorithm fails with some probability, so we could continue the algorithm until we get legitimate results, but it has an impact on time, which becomes unbounded. Thus, the Las Vegas algorithm usually uses legitimate results in defined time (it raises an error if the algorithm fails). Quicksort, covered in Chapter 5 , Sorting Algorithms , is an example of the Las Vegas algorithm.

[ 235 ]

Page 247

﻿

Programming and Randomized Algorithms

Monte Carlo algorithm : With the Monte Carlo algorithm, we cannot test when the algorithm could fail; however, the failure probability can be reduced by increasing the number of iterations, and taking expected results.

Generally, Las Vegas algorithms are preferred. However, there could be scenarios where Monte Carlo algorithms are useful, especially if we could bring down the failure probability and uncertainty is involved with the data itself. For example, say we want to determine the average height of a city using sample data. We have to run the Monte Carlo algorithm to estimate the average, using the sample, to get an expected value and the volatility factor involved. There is always some possibility of getting the wrong answer if the sample is a bad representation of the population, but we will not be able to detect it unless we have the population dataset.

The cost of randomized algorithms is based on expected bounds and high probability bounds. Expected bounds consists of the average outcomes captured using all random choices, whereas high probability bounds provide information on the upper bound of correctness, usually represented in terms of sampling size. Let's consider an example of finding the largest values from an n size array. The probabilistic algorithms could be very effective in a scenario where n is very large. The approach involves randomly picking an m element from the n size array, and deciding the max value, as demonstrated in Figure 9.8 :

Figure 9.8: Randomized algorithm for max value evaluation

As we keep increasing m , the probability that we will get the max value keeps increasing. For large n , if we use m ≈ log 2 (n) , then the results are pretty good. Let's evaluate the performance of m ≈ log 2 (n) using simulation. For example, with n=1,000,000 , the number of random sampling done is 20. Splitting the whole data into ventiles (20 blocks), the

probability of having a 5% error is , which is equal to is 0.64, and the solution will be in the first two quantiles representing a 10% error, that is 0.87. Let's conduct a simulation to get an error distribution using Monte Carlo simulation for n=10,000,000 and m selected as 24 with 10,000 iterations. The error distribution is shown in Figure 9.9 :

[ 236 ]

Page 248

﻿

Chapter 9

Figure 9.9: Error distribution with log m randomized sampling

A similar approach can be applied to other problem statements, such as picking a number from the upper half of n numbers. The probability of a number being in the upper half, when the greater of two number ( n 1 and n 2 ) is picked, will be 3/4 as there are four situations:

n 1 and n 2 both are from upper half n 1 and n 2 both are from lower half

n 1 from upper half and n 2 from lower half n 1 from lower half and n 2 from upper half

The accuracy can be further improved by increasing sampling, with the probability of getting the right solution being represented as , where k is the sample picked from n .

[ 237 ]

Page 249

﻿

Programming and Randomized Algorithms

Skip lists

Skip lists are probabilistic data structures developed in 1990 by Bill Pugh. It was developed to address search limitations in link lists and arrays. Skip lists provide an alternative to binary search trees ( BST ) and similar tree-based data structures, which tend to get unbalanced. The 2-3 tree discussed in Chapter 8 , Graphs , guarantees balancing with insertion and deletion; however, it is complicated to implement. Skip lists are easier to implement than a tree-based data structure. However, they do not guarantee optimal performance, as they use randomization in arranging entries so that the search and update time is O(log n) on average, where n is the number of entries in the dictionary. The average time in a skip list is independent of the distribution of keys and input. Instead, it only depends on the randomization seed selected during operations such as insertion. It is a good example of a compromise between implementation complexity and algorithm performance.

A skip list can be considered as an extension of the sorted link list. A sorted link list can be defined as a linked list in which nodes are arranged in sorted order. An example of a sorted linked list is shown in Figure 9.10 :

Figure 9.10: An example of a sorted link list

Any insertion or search operation in a sorted link list will require scanning the list with O(n) computational effort in an average case. The skip list is an extension that allows skipping nodes of a sorted link list. An example of a skip list is shown in Figure 9.11 :

Figure 9.11: An example of a ﬁrst order skip list

Figure 9.10 shows an example of a first order skip list, where there are two links: S 0 , which connects each node, and S 1 , connecting each alternate node of the sorted link list. The node of the skip list can be represented as shown in the following R script:

skListNode<-function(val, height=1){ # function to create empty environment

[ 238 ]

Page 250

﻿

Chapter 9

create\_emptyenv = function() {

emptyenv()

}

# Create skiplist node

skiplist <- new.env(parent=create\_emptyenv()) skiplist$element <- val

skiplist$nextnode<-rep(list(NULL), height) class(skiplist) <- "skiplist" skiplist

}

The nextnode in the skipListNode node function can be of variable length depending on the order of the skip list. A skip list can be of n th order; an example of a second order skip list is shown in Figure 9.11 :

Figure 9.11: An example of a second order skip list

The higher order allows a larger jump in the skip list, which, in turn, helps to reduce the execution time of operations such as searching, insertion, and deletion. An example of a search operation for value 101 within a skip list is shown in Figure 9.12 :

Figure 9.12: An example of search in a skip list

The dotted line in the graph shows the path that will be followed while searching for the value 101 in the skip list.

The script for searching a value in a skip list is shown next:

# Function to find a value in skip list findvalue<-function(skiplist, searchkey){

for (i in level:1){

[ 239 ]

Page 251

﻿

Programming and Randomized Algorithms

skiplist<-skiplist$nextnode[[i]] # head values while(!is.null(node) && node$element>searchkey){

skiplist<-skiplist$nextnode[[i]]

}

skiplist = skiplist$nextnode[0]

if(!is.null(skiplist) && searchkey==skiplist$element){

return(skiplist$element) # Return element

} else {

return(NULL) # Element not found

} } }

The skip list is stored as an array of environment in nextnode . The search operation starts with the highest level s 2 in case of example shown in Figure 9.12 . The while loop keeps moving from the head to 31 and then to 117 before exiting the while loop. The search operation then switches to the s 1 level, and the search moves from 31 -> 99 -> 117. At 117, the nodes again exit the while loop, the pointer is set at node 99, and height is adjusted to s 0 . The node goes to 117 before exiting and comparing with searchkey to stop at 101. The preceding examples of skip lists fall into ideal skip lists where the s 0 layer connects n nodes, s 1 connects 1/2 nodes, and the layer s k connects nodes. The distances are equally spaced and they are called a perfectly balanced skip list. However, maintaining this balance is an expensive process, especially during insertion and deletion operations, as all connections need to be updated accordingly to ensure a balanced skip list. To address the issue, a randomized skip list is created, which assigns a random level to the node. Let S represent a randomized skip list with height h for a dataset D consisting of the series {s 0 , s 1 , … , s h } . List s i maps to the subset of entries of D in a sorted order, starting at the head and ending at the tail of D . Also, S should satisfy following properties:

List s 0 should connect each node of the dataset in a sorted order.

For i = 1, … , h – 1 , list s i consists of randomly selected nodes from D. The selection is conducted using geometric distribution, such as level s 1 assigned 50% probability of selection, s 2 with 25% selection probability, and so on. Geometric distribution can be simulated using the rgeom(n, prob) function from the stat R library.

[ 240 ]

Page 252

﻿

Chapter 9

An example of a randomized skip list is shown in Figure 9.13 :

Figure 9.13: An example of a randomized skip list

The search algorithm holds for randomized skip lists. The operation for the Insertion of a node also uses randomization to decide the height of the new node. The search operation can be used to determine the place where the node is to be inserted. The search strategy is similar to the findvalue function to determine the insertion position. The connections are updated based on backward and forward scanning for randomized height. An example of the insertion operation is shown in Figure 9.14 :

Figure 9.14: An insertion example of a node with value 30 in a randomized skip list with height 2. The dashed lines shows a new connection inserted into an already existing skip

list

The randomized height is adjusted to a maximum value to ensure that any random value is not picked. The approach of having a fixed height to make it as a function of n such as h = max(l0, 2logn) or any other distribution constrained at maximum value. The deletion algorithm follows a structure similar to insertion, and is quite simple to implement, as the value will be removed and the incoming input is linked to the outgoing node.

[ 241 ]

Page 253

﻿

Programming and Randomized Algorithms

An example of deletion is shown in Figure 9.15 :

Figure 9.15: An example of the deletion of node 19 in a randomized skip list. The dashed line in the ﬁgure shows the connection removed and curved lines are extended to the

closest node

Probabilistic analysis of skip lists

Skip lists are very simple to implement. However, skip lists may not be the best data structure in a scenario where insertion is not prevented from going above the specified maximum height, as it will lead to an infinite loop. Let's consider a scenario of a skip list with height h , and n entries. Then, the worst case scenario for insertion, deletion, and search operations is O(n+h) . The worst case scenario for a skip list is quite inferior to other implementations discussed in this book. However, this is highly overestimated, because operations are randomized. Thus, the analysis requires probability to get a better estimate for the computational effort required for insertion, deletion, and search operations.

The probability that an entry can be reached at height k is where k>1 . In case of n entries probability P k of an entry reaching k height is . The probability that the height h of the skip list is greater than k is equal to the probability that the k th level has at least one position that is no greater than P k , that is, h is larger than 3logn with the probability given as follows:

The preceding equation can be generalized for any constant c as follows:

[ 242 ]

Page 254

﻿

Chapter 9

For a skip list with 10,000 entries, the probability would be one in 100 million. Thus, with very high probability, the height of skiplist is O(log n) . Similarly, as time spent in scanning any height in a skip list is O(1) , and a skip list would have O(log n) levels, which is high probability, the search is expected to take O(log n) computational effort.

As we have shown, the expected number of entries at position k is , which can be used to evaluate the space required for a skip list with n entries:

The preceding equation can be reduced using geometric summation as follows:

Thus, the expected space requirement is O(n) .

Exercises

1. One of the classic problems is the Tower of Hanoi, inspired by Hindu temples, where priests are provided with three poles of 64 gold disks, and each disk is a little smaller than the one beneath it, as shown in the following figure:

[ 243 ]

Page 255

﻿

Programming and Randomized Algorithms

The task is to transfer all 64 disks from one pole to another with two constraints, that is, only one disk can be moved at a time, and that they cannot place a larger disk on smaller one. Write a dynamic

programming-based approach to solve the Tower of Hanoi problem. 2. Implement a function that gets the edit distance of two input strings, where the edit distance is defined as insertion, deletion, and substitution. The function should determine the minimal number of edit distance required to modify one distance to another.

3. Given a rope of length n , write a function to determine how to cut the rope into m parts with length l[0], l[1] , and so on until l[m] so that the product of each part is maximum. The rope needs to be cut as an integer with a minimum of one cut. 4. Construct an all-pair shortest distance matrix for the following graph:

5. In a class, there are m students. An exam is conducted with 30 questions. The examiner picked random eight questions and graded them. The grades will be either A or D. A will be assigned if more than 50% questions are answered correctly; otherwise, grade D will be assigned. Set up a Monte Carlo simulation to determine the expected error for a person getting A instead of D.

6. Write a function to delete a node from a skip list.

7. Write a function to develop a skip list to contain numbers in the range [1, m] where m is not constant, and a support query to find an element e1 so that e1.key=q is a given pointer to an element e2 , e2.key=p so that p<q in O(log k) expected time, where k is the distance between elements e1 and e2 .

[ 244 ]

Page 256

﻿

Chapter 9

Summary

This chapter focused on the fundamentals of programming and randomized algorithms. The chapter built on the programming concepts of dynamic programming, and presented the difference between dynamic programming and recursion. The DAGs was also introduced in this chapter, and how it can be used to set up dynamic programming was discussed. Two popular examples of the knapsack problem and two APSP were covered in this chapter, and their solutions using dynamic programming were presented. Randomized algorithms, Las Vegas and Monte Carlo, were introduced with their application in

determining max value using randomization. This chapter also introduced skip lists and the extension to randomized skip lists. The probabilistic analysis of skip list was covered for major operations and data storage. The next chapter will introduce functional algorithms concepts. Functional algorithms provide the ability to write clean and clear code by eliminating state during runtime so that output is always determined based on input.

[ 245 ]