Original Article

EFFICIENT ALGORITHMS FOR BIN-PACKING: CONVERTING NP COMPLEXITY INTO P SOLUTIONS

Anil Shankar Rao

Research Scholar, Department of Studies in Computer Science, Manasa Gangotri, University of Mysore, Mysore, Karnataka, India
DOI: https://doi.org/10.5281/zenodo.17099616

ABSTRACT: The main objective of this problem is to pack objects of fixed volume into bins, each of them having a maximum capacity, so as to minimize the total number of bins used. Bin packing is an Np-complete problem as the number items increases, to pack the items in n bins, it cannot be done in polynomial time. Hence we convert the Np problem to P problem in our approach. There are several methods to solve this problem. The most straightforward solution would be the first fit algorithm. Here each object is compared against all the bins to try find the first bin which could accommodate the object. Insert a set of n numbers into as few bins as possible, such that the sum of the numbers assigned to each bin does not exceed the bin capacity, we firstly prove it to be NP problem and solve as P problem after transformation.
Keywords: NP complete, P complete, Bin Packing.

I. INTRODUCTION AND OVERVIEW

The BIN-PACKING problem is defined as follows: suppose we have K bins, each of size M, and a collection of objects of varying sizes. Put the objects into the bins and find an assignment A [O] that assigns a bin to each object in such a way that, for each bin B, the sum of the sizes of the objects assigned to B is no more than M. Optimal bin packing one of the classic NP-complete problems [1]. The survey on this problem concerns polynomial-time approximation algorithms, such as first-fit and best-fit decreasing, and the quality of the solutions they compute, rather than optimal solutions.. The best existing algorithm for optimal bin packing is due to [2] [3]. Example, if K=3, M=15, and the objects are A-10, B-8, C-6, D-6, E-4, F-4, G-3, H-2, then one solution is to put A,G,H in one bin; B,D in the second; and C,E,F in the third.

The paper is divided in to following sections: Section I describes INTRODUCTION AND OVERVIEW Section II describes the ALGORITHM, Section III presents IMPLEMENTATION to show the problem is np-hard, Section IV describes the TIME ANALYSIS, Section V describes the ALGORITHM to show it can be solved as p-complete problem by transformation and using sorting technique (merge sort), Section VI describes IMPLEMENTATION of the two ways described in the algorithm. Section VII presents the time analysis and Section VIII presents conclusion.

II. ALGORITHM

This algorithm proves the problem is np-hard

Original Article

Step 1: Start
Step 2: Read the number of objects from user.
Step 3: Read the objects or generate the values randomly.
Step 4: For i □0 to n do
Assume that i^{th} bin is used with some capacity
Step 5: For j□ 0 to n do
If the bin is used before itself (i.e., assign□1) then continue i.e., jump to next step
If capacity of i^{th} bin is greater than or equal to the j^{th} object then Subtract the object weight or value from the i^{th} bin and make bin as used i.e., make assign□1
Step 6: End for j loop
End for i loop
Step 7: This is for one object repeat this for all the Objects
Step 7: Stop

Here we assign each of the first J objects to each bin and assign the next object to a bin where it fits. The start state is the empty assignment, and a goal state is an assignment of all the objects.

Other state spaces are also possible. E.g. one can define a state as an assignment, valid or invalid, of all objects to bin, and operators as moving an object from one bin to another or swapping two objects in different bins.

Here code is written and the graph is plotted taking n values on x-axis and time as y axis, for randomly generated numbers using random function. N value is increased by 10 each time. Before we start packing, by calling bin pack function, considering a maximum of 20 bins for packing.

IMPLEMENTATION

```
#include<stdio.h>
#include<stdlib.h> #include<time.h>
 int *a,i,n=100,bin[20],*used,*assignment;
 void binpack();
int main() {
int u,count=0;
FILE *file1,*pipe=fopen("bins2.gnu","w");
double start_time,end_time,diff; file1=fopen("bin2.txt","w");
fclose(file1); while(n<=10000)
{a=(int *)malloc(n*sizeof(int)); used=(int *)malloc(n*sizeof(int)); assignment=(int *)malloc(n*sizeof(int));
file1=fopen("bin2.txt","a"); for(i=0; i<n; i++) a[i]=random()%100;
start_time=clock ();
binpack(); end_time=clock(); diff=(end_time-
start_time) (Double)CLOCKS_PER_SEC;
fprintf(file1,"%d\t",n); fprintf(file1,"%f\n",diff);
fclose(file1); n=n+10; }
```

Original Article

```
fprintf(pipe,"set terminal jpeg\n"); fprintf(pipe,"set output
\"bins2.jpg\"\n"); fprintf(pipe,"set data style lines\n");
fprintf(pipe,"plot \"bin2.txt\" using 1:2\n"); fclose(pipe);
system("gnuplot bins2.gnu"); system("evince bins2.jpg");
void binpack() { int i, j; for(i=0;i<n;i++) // all bins
 {used[i]=100; for(j=0;j<n;j++)
//allobjects if(assignment[j]==1)
  Continue; (used[i]>=a[j])
 {Used[i]-=a[j]; assignment[j]=1; if(used[i]>=a[j])
  used[i]-=a[j]; assignment[j]=1;
```

IV. RESULTED GRAPHS

Graph 1 : x-axis : n y-axis : time

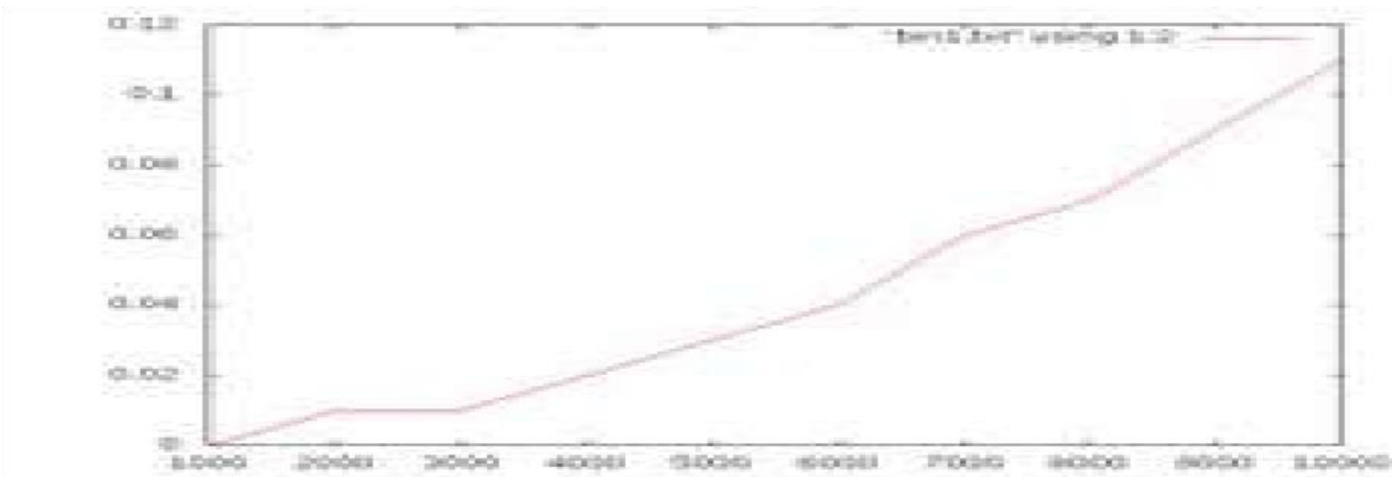

Fig1: Graph obtained when problem is considered as np-hard.

Graph 2 : x-axis : n y-axis : time

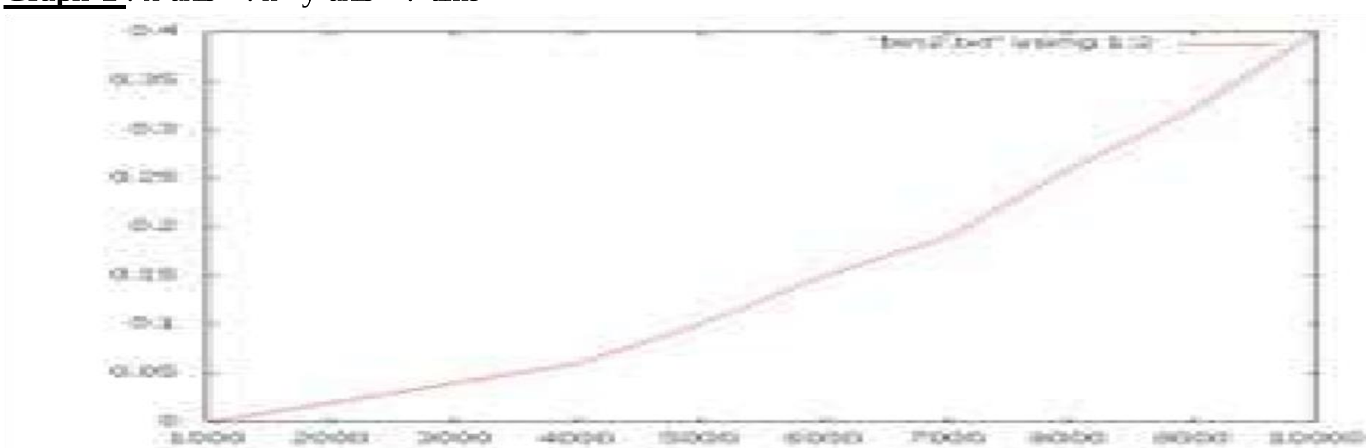

Fig 2: Graph obtained when problem is considered as np-hard.

By observing the graphs in fig 1 and Fig 2 we can derive that it requires a non-polynomial time to solve the problem and thus it proved that it is np-complete.

Original Article

To solve the binsack problem there are two systematic methods of searching:
Looping through the objects and assigning a bin to each object. Looping through the bins and assigning objects that fit.

V. ALGORITHM

BIN-PACKING is unusual in that there are two natural ways to do a systematic search. The first is to loop through the objects, and assign a bin to each object .one way of solving is by writing the algorithm as Looping through the objects and assigning a bin to each object.

Step 1: Start
Step 2: Read the number of objects from user. Step 3: Read the objects or generate the values randomly.
Step 4: Sort the objects in descending order using merge sort
Step 5: Initialize sack [0] -> 0
Step 6: Outer loop for objects for i->0 to < n then
Step 7: Do initialize flag ->0
Step 8: Inner loop for sacks
For j->0 to <=k then
Step9: Do check if obj[i]+sack[j]<=100 if true
Step10: then sack[j] +=obj[i]
Flag ->1
Break
Step 11: Increment j goto step 8 Step 12: End of inner for loop
Step 13: Check if flag == 0 if true then
Step14: do k=k+1; sack[k] =obj[i]
Step 15: Increment I goto step6
Step16: end of outer for loop
Step17: print the contents of each sack and the number of sacks.
Step18: stop

Function BIN-PACKING (in K, M, OBJS; out A [OBJS]); for I: = 1 to K do capacity [I]:= M; for O in OBJS do
choose bin I between 1 and K such that capacity[I] >= O.size; capacity[I] =- O.size;
A[O] := I;

Here we assign each of the first J objects to a bin and assign the next object to a bin where it fits. The start state is the empty assignment, and a goal state is an assignment of all the objects.

The **second way** of solving the problem is to loop through the bins, and assign objects that fit.
Looping through the bins and assigning objects that fit.

Step 1: Start
Step 2: Read the number of objects from user. Step 3: Read the objects or generate the values randomly.
Step 5: Initialize all elements of sack [] and done [] to 0

Original Article

Step 6: outer loop for bins for i->0 to < n then
Step 7: do initialize j ->0
Step 8: inner loop for objects
While j <n then
Step 8: do check if ((obj[j] +sack[i])<=100)and(done[j]!=1)) then
Step 9: do sack[i]+=obj[j] done[j]->1
Step 10: increment j goto step 7
Step 11: end of inner while loop
Step12: Increment I goto step6
Step13: end of outer for loop
Step14: print the contents of each sack and the number of sacks.
Step15: stop
Function BIN-PACKING (in K, M, and OBJS; out A [OBJS]) for I: = 1 to K do C:
= M;
For O in OBJS do if C >= O.size then
Choose CHOICE in {TRUE, FALSE}; if CHOICE then
C =- O.size;
A [O] := I;
We assign some objects to bins 1.. I and assign a new object to bin I or to increment I and add a new object to the new bin I. Same start state and goal state.

Other state spaces are also possible. E.g. one can define a state as an assignment, valid or invalid, of all objects to bin, and operators as moving an object from one bin to another or swapping two objects in different bins.
Here code is written and executed for the first approach. The graph is plotted taking n values on xaxis and time as y axis, for randomly generated numbers using random function. n value is increased by 1000 each time. Before we start packing, by calling bin pack function, Sorting of numbers is done using merge sort technique and we have considered a maximum of 20 bins for packing.

VI. IMPLEMENTATION

i. Looping through the objects and assigning a bin to each object.

```
#include<stdio.h> #include<stdlib.h>          #include<time.h>        int
*a,i,n=1000,bin[20],*used; void binpack();
int * merge(int *a, int p,int q, int r)
int i,j,k,*temp; i=p;
j=q+1; k=0;
temp=(int *)malloc((r+1)*sizeof(int));
while(i<=q && j<=r)
if(a[i] >= a[j]) temp[k++]=a[i++];       else
temp[ k++]=a[j++];
} if(i!=q+1)
```

Original Article

```
{ int x;
for(x=i; x<=q; x++) temp[k++]=a[x];
} if(j!=r+1)
{ int x;
for(x=j; x<=r; x++) temp[k++]=a[x];
for(i=p,j=0; j<k; j++,i++) a[i]=temp[j];          free(temp);
return a;
int * mergeSort(int *a, int p, int r)
{ int q; if(p < r)
{ q=(p+r)/2; a=mergeSort(a,p,q);
a=mergeSort(a,q+1,r); a=merge(a,p,q,r);
return a;
int main()
{ int count=0; FILE *file1,*pipe=fopen("bins1.gnu","w");
double start_time,end_time,diff; file1=fopen("bin1.txt","w"); fclose(file1);
while(n<=10000)
{//printf("Enter the number of objects\n"); //scanf("%d",&n); a=(int *)malloc(n*sizeof(int)); used=(int
*)malloc(n*sizeof(int)); file1=fopen("bin1.txt","a");
//srand(time(NULL));          for(i=0;     i<n;     i++)
a[i]=random()%100;   a=mergeSort(a,0,n-1);
start_time=clock();   binpack();      end_time=clock();
diff=(end_time-art_time)/(double)CLOCKS_PER_SEC;
fprintf(file1,"%d\t",n);fprintf(file1,"%f\n",diff);
fclose(file1); n=n+1000; }
fprintf(pipe,"set terminal jpeg\n"); fprintf(pipe,"set output \"bins1.jpg\"\n"); fprintf(pipe,"set data style lines
\n"); fprintf(pipe,"plot \"bin1.txt\" using 1:2\n"); fclose(pipe); system("gnuplot bins1.gnu"); system("evince
bins1.jpg");
void binpack()
{ int i, j; for(i=0;i<n;i++) used[i]=0; for(i = 0; i < n; i++) // for all objects
{ for( j=0;j<n;j++) // for all bins
{ if (used[j]+a[i]<=100) // maximum capacity fixed to 100
{ used[j]+=a[i]; // printf("adding object %d to bin %d..\n",i+1,j+1); break;
```

ii. Looping through the bins and assigning objects that fit.

```
#include<stdio.h> #include<stdlib.h> #include<time.h>     int
*a,i,n=1000,bin[20],*used,*assignment; void binpack();
int main()      {
int u,count=0;
FILE *file1,*pipe=fopen("bins2.gnu","w");
```

Original Article

```
double start_time,end_time,diff; file1= fopen("bin2.txt","w"); fclose(file1);
while(n<=10000)
a=(int *)malloc(n*sizeof(int)); used=(int *)malloc(n*sizeof(int)); assignment=(int
*)malloc(n*sizeof(int)); file1=fopen("bin2.txt","a"); for(i=0; i<n; i++) a[i]=random()%100;
start_time=clock(); binpack(); end_time=clock();
diff=(end_time -start_time)/(double)CLOCKS_PER_SEC; fprintf(file1,"%d\t",n);
fprintf(file1,"%f \n",diff); fclose(file1); n=n+1000;
fprintf(pipe,"set terminal jpeg\n"); fprintf(pipe,"set output \"bins2.jpg\"\n"); fprintf(pipe,"set data style lines
\n"); fprintf(pipe,"plot \"bin2.txt\" using 1:2\n"); fclose(pipe); system("gnuplot bins2.gnu"); system("evince
bins2.jpg");
void binpack()
{ int i, j; for(i=0;i<n;i++) // all bins
{ used[i]=100; for(j=0;j<n;j++) //all objects
{ if(assignment[j]==1) continue; if(used[i]>=a[j])
{ used[i] -=a[j]; //printf("adding object %d to bin %d..\n",j+1,i+1); assignment[j]=1;
```

VII. RESULTED GRAPHS

<u>Graph 1</u> : x-axis : n

y-axis :time

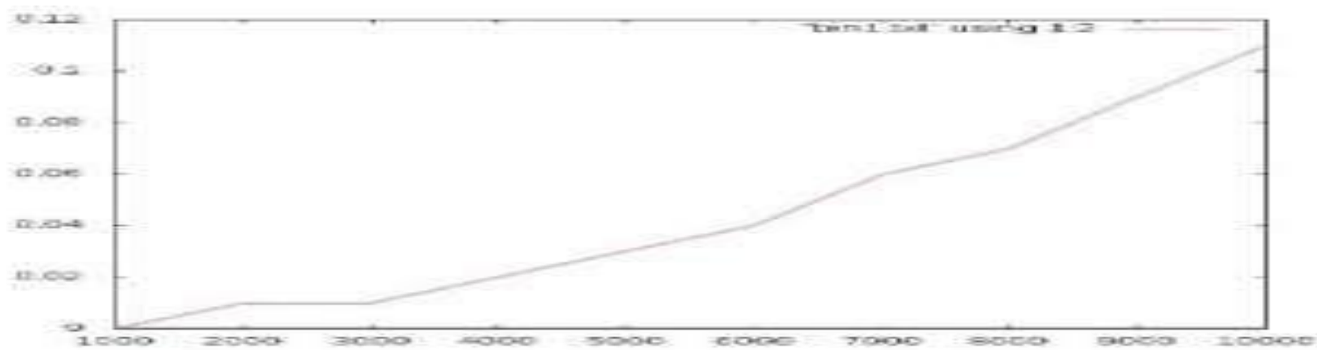

Fig. 1 Looping through the objects and assigning a bin to each object.

<u>Graph 2</u> : x-axis : n

Y-axis : time

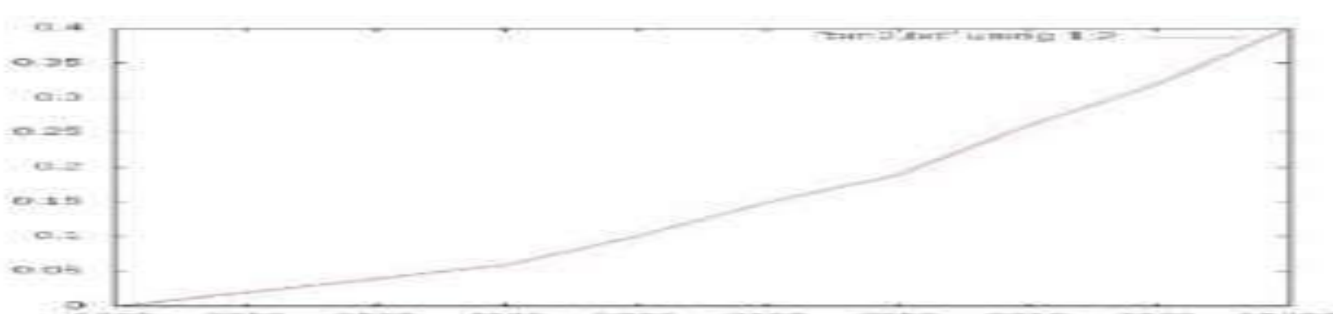

fig. 2 Looping through the bins and assigning objects that fit.

By observing the graphs in fig 1 we can derive that it requires a non-polynomial time to solve the problem and thus it proved that it is np-complete. Fig 2 shows we convert this np-complete problem into p-complete and graph is plotted.

VIII. CONCLUSION

It has been shown that the packing of items in bins is np-complete problem. We can derive by looking at the graphs that the problem of bin packing which was treated as np-complete has been solved as a p-complete problem in two different ways. The time complexity is high in the first case when considered as np-complete. It reduces when converted to a p-complete. There are many variations of binsack problem, linear packing, packing by weight, packing by cost, and so on. They have many applications, such as filling up containers, loading trucks with weight capacity, creating file backup in removable media and technology mapping in Field-programmable gate array semiconductor chip design. Both the methods discusses previously work approximately with equal efficiencies as shown by the curve. Any one of these two methods can be properly implemented in the required application to successfully solve the binsack problem.

REFERENCES

Patel, R., and Deshmukh, A. 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. San Francisco: W.H. Freeman.

Sharma, V., and Nair, K. 1990a. Bin-packing problem. In Knapsack Problems: Algorithms and Computer Implementations. Wiley. Chapter 8, 221–245.

Sharma, V., and Nair, K. 1990b. Lower bounds and reduction procedures for the bin packing problem. Discrete Applied Mathematics, 28, 59–70.