An Improved Safety Detection Algorithm Towards Deadlock Avoidance

1st Momotaz Begum
Assistant Professor, Department of CSE
DUET
Gazipur-1707, Bangladesh
momotaz.2k3@gmail.com

2nd Omar Faruque
Lecturer, Department of CSE
DUET
Gazipur-1707, Bangladesh
faruque@duet.ac.bd

3rd Md. Waliur Rahman Miah
Associate Professor, Department of CSE
DUET
Gazipur-1707, Bangladesh
walimiah@yahoo.com

4th Bimal Chandra Das
Associate Professor
DIU, Dhaka, Bangladesh

***Abstract*—In operating system, the resource allocation to each process is a critical issue, because the resources are limited and sometimes may not be shareable among processes. An ineffective resource allocation may cause deadlock situation in the system. The banker's algorithm and some other modified algorithms are available to handle deadlock situations. However, the complexities of these algorithms are quite high. This paper presents an innovative technique for safe state detection in a system based on the maximum resource requirements of processes and the minimum resource available. In our approach, the resource requirements of each process are sorted in a linked list, where it is easy to check whether the request exceeds the available resources. In our experiments we compare our approach with some other methods including the original banker's algorithm. The results show that our proposed method provides less time complexity and less space complexity than the other methods.**

***Keywords*—** deadlock avoidance, resource allocation, time complexity, space complexity, banker's algorithm.

I. Introduction

In a multiprogramming environment, several processes may compete for a finite number of resources. Requested resource by a process may not be available at demand, and then the process enters in a wait state. Waiting processes may never again change the state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock [1], [2], [3]. A common algorithm that is used by many systems to avoid the deadlock is banker's algorithm. The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes a test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. The time complexity of the original banker's algorithm is therefore in quadratic order $O(n^3d)$, where *n* refers the number of processes and *d* denotes the types of resources in the system. The main part of the banker's algorithm is the safety algorithm that checks the safety state of the system. The time complexity of the safety algorithm is $O(n^2d)$ [1], [2].

Many researchers have also studied the banker's algorithm for deadlock handling.Youming L *et al* in [4] and [5] improved the running time of banker's algorithm for deadlock avoidance. The authors applied a graph-free and optimal algorithm based on Dijkstra's algorithm with the cost of linear dependence on the number of processes.

A modified banker's algorithm in [6] is an updated version of the original banker's algorithm. The algorithm in [6] is based on the greedy method and the permutation of resources of the system. The construction of the permutation matrix takes O(nd^2) as each entry in the matrix takes O(n) to complete. Therefore the entire safety detection algorithm has time complexity of O(nd^2 + $d\sum_{1\leq j\leq d} M[j]$). The total time complexity of the deadlock avoidance algorithm is O($n^2d^2 + ndM$) where, *M* is the total number of resources in the system. In case of space complexity it is worse than the banker's algorithm because of the permutation matrix. The space complexity of this algorithm is O(nd^2).

A new allocation method is introduced in [7] for reducing unnecessary safety checking, and to improve the banker's algorithm. In [8] a supervisory control policy is developed for avoiding deadlocks in a class of automated manufacturing system (AMS) with multiple unreliable resources. Louise E., and Moser S. in [9] presented a version of the banker's algorithm with changes in maximum resource requirements and allocations that allows each process to make resource allocation decisions freely and simultaneously. The recent trend shows that the banker's algorithm also applicable to buffer space allocation in flexible manufacturing. Mark L. *et al* in [10] focused on these algorithms when properly applied to the manufacturing environment.

Authors in [11] introduced a new natural extension of the banker's algorithm for deadlock avoidance in operating systems. The banker's algorithm has been mentioned in [12] to be useful for some other systems as well, for example: manufacturing systems with automated guided vehicle systems, and control of material flow in job shops. The authors of [12] present the results of simulation experiments to compare the performance of several deadlock handling methods. Gang

978-1-7281-5033-8/20/$31.00 ©2020 IEEE

X. and Zhiming W. in [13] offered an improvement of the banker's algorithm for deadlock avoidance algorithm used in Flexible Manufacturing System (FMS). An improved deadlock prevention algorithm for distributed system is proposed in [14]. The approach is based on wait die and wound wait algorithms. It considers the priority of the processes and look at the time stamps to decide which process is to abort and which to carry on. To make a reliable communication during resource provisioning on Multi-access edge computing (MEC) system a deadlock avoidance algorithm is proposed in [15] for industrial IoT devices. It uses software defined networking to reduce communication overhead. For an automated manufacturing system (AMS), C. Gu *et al* in [16] focused on the deadlock avoidance problem for system of simple sequential process with resources by using Petri nets structural analysis theory. To prevent deadlocks is simulations Salim *et al* in [17] used the resource request algorithm. It provides resources to the process and analyse whether the state after lending resources is in a safe state or unsafe state. Pang *et al* in [18] explored the application of Spiking neural P system (SNP systems) to the optimal revocation of deadlock process.

The above mentioned researches have different focuses than our focus. However, all of them concerned with the banker's algorithm which our algorithm is based on. The focus of [6] have some similarity with the focus of our research. The authors of [6] proposed some modifications of the original banker's algorithm towards performance improvement. We also propose some different modification of banker's algorithm and achieved better time and space complexity.

Our contribution in this paper starts with pointing out some limitations of different algorithms used for safety detection towards deadlock avoidance. We propose a new improved algorithm that uses dynamic memory allocation, and reduces the space complexity during the implementation. It also improves the time complexity during execution.

The structure of this paper is as follows:

This section (section 1) provides an overall introduction with background of some related works. Section 2 provides a brief review for the readers to catch up with the original banker's algorithm. The operational steps and the details of the proposed new algorithm is provided in section 3. The complexity analysis is given in section 4. A comparison of the performances of different algorithms with our algorithm is explained in section 5. And finally section 6 concludes the paper with our future plan.

II. The banker's algorithm

The banker's algorithm [1] is a pioneer deadlock avoidance algorithm in operating system. This algorithm was developed to avoid deadlock in the systems having multiple instances of each resources type. The idea of this algorithm is similar to the regular amount allocation of the bank, that's why it is named as banker's algorithm. For example, in a bank when a customer request to allocate money, the bank first check if this requested amount is allocated, then requests from other customers can be satisfied or not. If this condition is satisfied the allocation is done against the request. Otherwise the customer need to wait.

Similarly, in banker's algorithm, resources are carefully allocated to process so that the system never gets stuck in an unsafe state. At the time of creation, each process need to declare the maximum number of instances of each resource type it will need to accomplish a task. The maximum required resource cannot be more then total resource number of the system. To avoid deadlock in the system, when any process request for allocation of some resources, the algorithm first checks whether it is possible to complete the execution of all running process with the new resources allocation. If the system finds that all processes can be completed without any blockage, then request for the resources is fulfilled. If the system find any blockage then the request will not be satisfied and the process need to wait. The algorithm is designed in two parts. First is safety detection part, which finds out that there would be no dead-lock situation (unsafe condition) for any resource request. And the second part is the resource allocation, if system is in safe condition then allocate the requested resources.

III. Proposed Algorithm

Our proposed algorithm is based on a sorting method and the linked list data structure. In this algorithm the processes are sorted in increasing order based on the maximum number of resources of any type needed by a process. Therefore, when we check the processes in safety algorithm the processes with lowest resource requirement comes first. We don't need to check the process sequence more than once. Here we use the linked list as data structure to store the additional resource requirement of each process to complete its execution. In the linked list the required resources of each process are stored in decreasing order based on the amount of resource instance of each type. This makes the comparison process much simpler in the algorithm.

By using sorting method, our algorithm reduces the complexity of the safety detection part. For a complete description of this process we briefly describe the safety detection algorithm in the following section.

A. The Safety Detection Algorithm

The data structure defines the state of the resource allocation system. To implement the banker's algorithm different researchers used different data structures. In our approach, suppose that n refers to the number of processes and d refers to the types of resources in the system. The data structure *Maximum_need[i,j] : (i = 1, 2, . . . , n, j =1, 2, . . . , m)* of integers, defines the maximum resource requirements of each type for each process. The currently allocated resources of each process is represented by *Current_allocation[i,j]:(i = 1, 2, . . . , n, j =1, 2, . . . , m)* of integers. The number of currently available resources are represented by *Available_resources[j]* of integers; where *Maximum_Needed_allocation* $\leq$ *Minimum_Available*. The identifier *Minimum_Available* of integers,

hold the minimum number from all available resources. The number of resources needed for each process to complete execution is contained in *Needed_allocation[i,j]:(i = 1, 2, . . . , n, j =1, 2, . . . , m)* of integers. The maximum number of resource of each process are stored in *Maximum_Needed_allocation[i,j] : (i = 1, 2, . . . , n, j =1, 2, . . . , m)* of integers. In the matrix the amount of resources are sorted in decreasing order. *Complete_process[i]:(i = 1, 2, . . . ,n)* of booleans, indicate process completion status, *0* means cannot be completed and *1* means can be completed. The *Resource[j]:(j =1, 2, . . . ,m)* of integers,vector stores number of maximum resources of each type exist in system. *Safe_sequence* encloses the order list of process completion sequence to avoid deadlock. In Fig. I shows the flowchart of safety detection algorithm. The algorithm returns the execution sequence of the processes if the system is in safe state otherwise says the system is not safe.

Steps:

1 *for (1≤ i ≤n){ Completed_process[i]:= 0 }*
2 *for (1≤i≤n,1≤ j ≤d) {*
if (Maximum_need[i,j]>Resource[j])
{ return unsafe_state} }
3 *Radix_Short(Process[n] ,Maximum_Needed_allocation[n]) [Sort process list in increasing order by maximum needed resources and also Maximum_Needed_allocation vector itself]*
4 *for (1≤ i ≤ n) {*
if (Maximum_Needed_allocation[i] ≤Minimum_Available and Completed_process[Process[i]] == 0){
Completed_process[Process[i]]:= 1
Safe_sequence:= { Safe_sequence U Process[i]}
Available_resources[d]:=Available_resources[d] + Current_allocation[Process[i],d]
Compute(Minimum_Available) }
else {
for (1≤j≤d)
If (Needed_allocation[Process[i],j]≤ Available_resources[j])
{ Continue ;}
else { Break ;}
if (j == d) {
Completed_process[Process[i]]:= 1
Safe_sequence := { Safe_sequence U U Process[i]}
Available_resources[d]:=Available_resources[d] + Current_allocation[Process[i],d]
Compute(Minimum_Available) }
else {return unsafe state ;} } }
5 *for (1≤ i ≤ n) {*
if (Completed_process[i]== 0)
{return unsafe state } }
6 *return safe state and Safe_sequence.*

Radix Sort Algorithm:
Radix_Sort(Process[n] Maximum_needed_allocation[n])-
This algorithm is used to sort *Process[n]* and *Maximum_needed_allocation[n]* array in increasing order based on the values of maximum number of any types of resource needed by a process to complete execution, where *n* is the number of process.

Steps:

1 Create temporary array to store sorted processes and maximum resource requirements of all processes and initialize both array by 0.
Output_process[n]={0} ,Temp_maximum_Needed_allocation[n]={0}
2 To count occurrence of each key in values of *Maximum_Needed_allocation[n]* array need another array of length 10 and initialize this array by 0.
Count[10]={0}
3 Now find the length of largest value in the *Maximum_Needed_allocation[n]* array and store this value in variable *Run_times.*
Run_times = Length(Max(Maximum_Needed_allocation[n]))
4 Now execute step 5 to 8 for *i = 1 to Run_times,increment by 1*
5 Increment count of occurrence in *Count[10]* for *i* digit of each values in *Maximum_Needed_allocation[n]* array for all processes. *Count[Maximum_Needed_allocation[j]%(10^i)]++ ,For j=1 to n ,increment by 1*
6 Now calculate the position the values of processes and maximum resource requirements arrays after sorting values.
Count[j] = Count[j] + Count[j-1] ,For j=1 to 10 ,increment by 1
7 This step rearranges all values in processes and maximum resource requirements arrays based on the values of occurrences of *i th* digit in the *Count[10].*
j = n-1 to 0, decrement by 1
Temp_maximum_Needed_allocation[Count[Maximum_Needed_allocation[j]%(10^i)]] = Maximum_Needed_allocation[j]
Output_process[Count[Maximum_Needed_allocation[j]%(10^i)]]= Output_process[j] Count[Maximum_Needed_allocation[j]%(10^i)]- - [decrement the counter by 1 each time]
8 Now copy the sorted arrays into original arrays and initialize temporary arrays again for next iteration.
Process[n]= Output_process[n]
Maximum_Needed_allocation[n] = Temp_maximum_Needed_allocation[n]
Count[10]={0}
9 After all iteration in step 4 these two arrays *Process[n]* and *Maximum_Needed_allocation[n]* are already sorted and return the result to safety detection algorithm.

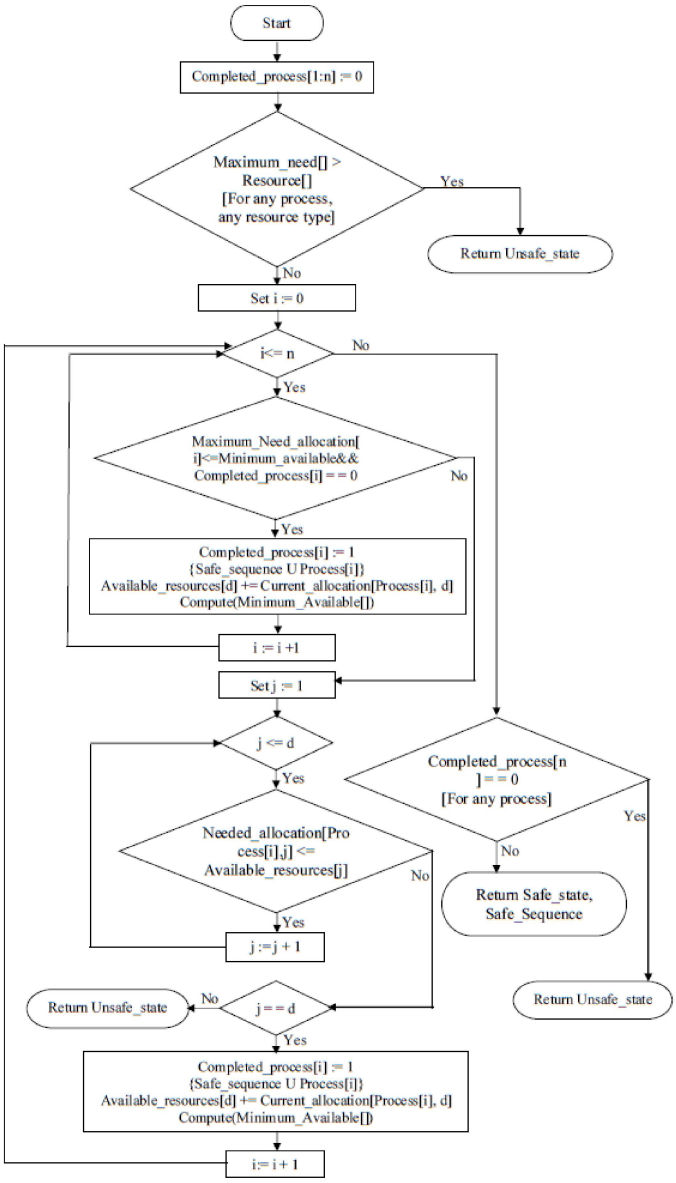

Fig. 1: Flowchart of Safety Detection Algorithm

B. Algorithm Verification

We consider a system with ten processes ($P_0 \sim P_9$) and five resource types (A $\sim$ E). Each resource has 25, 30, 20, 30, and 35 instances respectively. Fig. II shows the resource allocation graph for all processes. For each process the maximum resource requirement and resource allocation vector is given in Table I. In resource allocation graph, a directed edge from a resource type R to a process P means that, resource R is already allocated to process P. Again a directed edge from a process P to a resource type means process P needs resource R to complete its execution. From current state of resource allocation graph, we find that, process P_0 need some resources of type A and already assigned some resources of type C. Process P_2 need some resources of type D and already assigned some resources of type A. Process P_7 need some resources of type C and already assigned some resources of type D. There is a cycle of waiting processes, $P_0 \rightarrow P_2 \rightarrow P_7 \rightarrow P_0$. So processes P_0, P_2,P_3,P_6,P_7 and P_9 are deadlocked here.

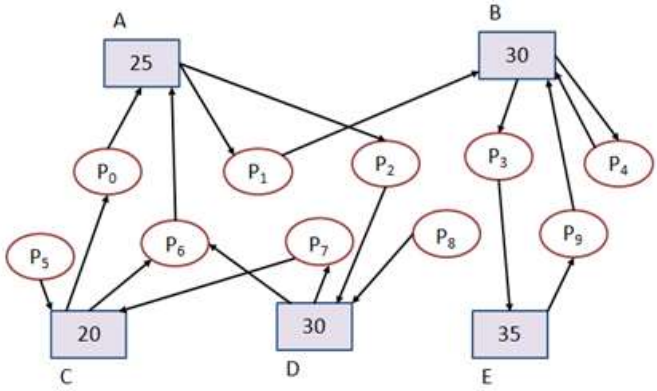

Fig. 2: Resource allocation graph

TABLE I: Maximum resource requirement and resource allocation for each process.

Process No	*Maximum_need*					*Current_allocation*				
	A	B	C	D	E	A	B	C	D	E
P_0	7	0	6	0	4	3	0	3	0	3
P_1	3	8	4	0	0	2	1	1	0	0
P_2	0	6	3	7	0	0	4	0	5	0
P_3	2	0	0	9	7	2	0	0	6	5
P_4	4	3	2	0	1	2	3	2	0	1
P_5	8	7	6	2	5	5	6	3	2	4
P_6	5	2	1	8	9	4	2	1	4	7
P_7	3	6	9	0	0	3	6	2	0	0
P_8	0	4	8	2	5	0	3	3	2	5
P_9	0	0	9	7	8	0	0	3	7	3

Table 1 shows current resource allocation and maximum resource requirements of each process. At this point we need to calculate the remaining resource needed for each process to complete its execution by subtracting *Current_allocation* from *Maximum_need*. *Maximum_Needed_allocation* is the maximum number of any resource type in *Need_allocation*. Table 2 shows the *Maximum_Needed_allocation* and *Need_allocation* of each process.

TABLE II: *Need_allocation* and *Maximum_Needed_allocation* vector of processes.

Process No	*Needed_allocation*					*Maximum_Needed_allocation*
	A	B	C	D	E	
P_0	4	0	3	0	1	4
P_1	1	7	3	0	0	7
P_2	0	2	3	2	0	3
P_3	0	0	0	3	2	3
P_4	2	0	0	0	0	2
P_5	3	1	3	0	1	3
P_6	1	0	0	4	2	4
P_7	0	0	7	0	0	7
P_8	0	1	5	0	0	5
P_9	0	0	6	0	5	6

Now all the processes will be sorted based on the *Maximum_Needed_allocation* value in increasing order. Table 3 represent the order of processes after sorting.

TABLE III: Sorted list of processes based on maximum need

Process No	*Needed_allocation*					*Maximum_Needed_allocation*
	A	B	C	D	E	
P_4	2	0	0	0	0	2
P_2	0	2	3	2	0	3
P_3	0	0	0	3	2	3
P_5	3	1	3	0	1	3
P_0	4	0	3	0	1	4
P_6	1	0	0	4	2	4
P_8	0	1	5	0	0	5
P_9	0	0	6	0	5	6
P_1	1	7	3	0	0	7
P_7	0	0	7	0	5	7

The available resource for each resource types are [A B C D E] is [4 5 2 4 7]. To perform the safety check operation we have to sort the processes in increasing order of *Maximum_needed_allocation* showed in Table II. In Table III we give the sorted *Maximum_Need* vector. Followings are explanations for each process in Table III and Table IV:

1. *For P_4:* The available resources are [A B C D E]=[4 5 2 4 7] and minimum available is 2. The need vector of the process is$\{$(A 2)$\}$ and *Maximum_Needed_allocation* is also 2. So we get *Maximum_Needed_allocation*$\leq$*Minimum_Available* is true and the process is completed in one comparison. Now the available resources are [A B C D E]=[6 8 4 4 8].

2. *For P_2:* The minimum available is 4. The need vector of the process is $\{$(B 2),(C 3),(D 2)$\}$ and *Maximum_Needed_allocation* is 3 and satisfied the condition. Now the available resources are [A B C D E]=[6 12 4 9 8].

3. *For P_3:* The minimum available is 4. The need vector of the process is $\{$(D 3), (E 2)$\}$ and *Maximum_Needed_allocation* is 3. So the condition *Maximum_Needed_allocation*$\leq$*Minimum_Available* is true and the process is completed in one comparison. Now the available resources are [A B C D E] = [8 12 4 15 13].

Similar method is used for each process in the list. The complete operation is given in Table IV. Finally the safe sequence of the processes obtained by our algorithm is as $<$$P_4$, P_2, P_3, P_5, P_0, P_6, P_8, P_9, P_1, and $P_7$$>$.

From the previous example we can conclude that, if the maximum number of instance of any resource type needed of a process is less than the minimum available instance of all resource type, then the resource required for the process can be allocated without checking all the resources. Therefore, in this example we need only 10 comparisons to check safety. However, if use the original banker's algorithm was used, that would take at least 50 comparisons for the same operation.

IV. Complexity Analysis of Proposed algorithm

A. Time Complexity

In the proposed algorithm in subsection III.A, step 1 executes for *n* times one for each process. Step 2 takes O(nd) to check for each resource type of each process. Step 3 is the sorting method which takes O(nd) where *n* is the number of processes and *d* is the number of maximum resource requirement. Step 4 performs the main operation in the algorithm. It takes either O(*n*) times in the best case or O(nd) in the worst case based on the specified condition. Therefore, the overall time complexity is O($n + nd + nd + nd$) or O(nd). In best case the time complexity of the main procedure of safety checking will be only O(n) one for each process.Therefore, the time complexity in our algorithm is significantly reduced than the time complexity of the original banker's algorithm.

B. Space Complexity

The space complexity of the original banker's algorithm [1] is O(nd) and for the algorithm proposed by Youming Li [6] is O(nd^2). The space complexity of the modified algorithm [6] is higher than the original algorithm due to the storage requirement for the permutation matrix. In our proposed algorithm, for simplicity we have shown array data structure for *Maximum_need[]*, *Current_allocation[]* and *Needed_allocation[]*. In application we suggest to use linked list for these high dimensional data structure. Because linked list is dynamic data structure which is very efficient for memory utilization, access time is very fast and no need pre-allocated memory. Therefore, the space requirement of this algorithm is less than the original banker's algorithm.

V. Performance Comparison

There is some difference between our new safety detection algorithm and the original banker's algorithm [1] and also with the modified banker's algorithm defined in [6]. In the original banker's algorithm, the processes are evaluated without any sorting and it consider all resource types. Our algorithm has used the sorting method for processes and the data structure *Minimum_Available* contains the resources of each process.The modified algorithm proposed in paper [6] is also little more different from our algorithm. In that algorithm the authors have used a large dimensional permutation and to sort the process they also proposed to use counting sort which has a high time complexity. In our proposed algorithm we do not use any permutation which reduces the time complexity and also we use the radix sort method to sort the processes based on maximum resource requirement. The modified algorithm [6] also uses the greedy approach to check all resource type, but we know that greedy method do not always provide the best result and also increase the complexity. In the new algorithm we have used the linked list to store the resource requirement. To check the availability of required resources we have used a new procedure in step 4. These are the main differences with both

TABLE IV: Representation of complete operation of our algorithm.

Process No	*Needed_allocation*					*Maximum_Needed_allocation*	*Available*					*Minimum_available*	*Maximum_Needed_allocation* ≤*Minimum_available*
	A	B	C	D	E		A	B	C	D	E		
P_4	2	0	0	0	0	2	4	5	2	4	7	2	True
P_2	0	2	3	2	0	3	6	8	4	4	8	4	True
P_3	0	0	0	3	2	3	6	12	4	9	8	4	True
P_5	3	1	3	0	1	3	8	12	4	15	13	4	True
P_0	4	0	3	0	1	4	13	18	7	17	17	7	True
P_6	1	0	0	4	2	4	16	18	10	17	20	10	True
P_8	0	1	5	0	0	5	20	20	11	21	27	11	True
P_9	0	0	6	0	5	6	20	23	14	23	32	14	True
P_1	1	7	3	0	0	7	20	23	17	30	35	17	True
P_7	0	0	7	0	0	5	22	24	18	30	35	18	True
							25	30	20	30	35		

of the previous algorithms. Table V presents a comparison of time complexity of the three safety detection algorithms. From that table it can be seen that our proposed algorithm achieved the best time complexity among the three algorithms.

TABLE V: Time complexity of safety algorithm.

Algorithm	**Time Complexity of Safety Algorithm**
Original Banker's Algorithm	$O(n^2d)$
Modified Banker's Algorithm [6]	O(nd^2+d$\sum_{1\leq j\leq d} M[j]$)
Proposed New Algorithm	O(nd)

VI. Conclusion

In this paper we have proposed a new algorithm based on the well-known banker's algorithm. The experiments and analysis show that our proposed algorithm obtained an improved result with more efficient time and space complexities over the original banker's algorithm.In some cases, our approach also performs better than some contemporary modified banker's algorithm, for example the algorithm proposed in [6]. Our new algorithm is *n* times faster than the original banker's algorithm and this is also *d* times faster than the modified algorithm of [6]. In our algorithm we have eliminated the use of complex and large dimensional permutation matrix that is used in the modified algorithm of [6]. Our algorithm uses dynamic memory allocation for data structures that reduces the memory requirement. In the best case condition, the algorithm needs only *n* comparisons to detect the safe state of any system, whereas the original banker's algorithm would need $O(n^2d)$. Therefore, when the number of resource types is increased then the result in our algorithm is more promising. We conducted experiment and analysis onlyfor safety detection. In future, we plan to apply the proposed new algorithmfor the purpose of deadlock avoidance and conduct more comprehensive analysis to validate our method with some undesired processes.

References

[1] Silberschatz A., Galvin P. B. and Gagne G., *Operating System Principles*, 6th ed. John Wiley & Sons, 1982.

[2] Tanenbaum A. S., *Modern operating system*, 2nd ed. Prentice Hall, Inc, 1992.

[3] Hao Y. and Hesuan H., *Robust Deadlock Control Using Shared-Resources for Production Systems with Unreliable Workstations*, 2013 IEEE International Conference on Automation Science and Engineering (CASE), 2013.

[4] Youming L., Ardian G., and James H., *On Dijkstra's algorithm for deadlock detection*, Advanced techniques in computing sciences and software engineering. Springer, Heidelberg, pp. 385–387, 2010.

[5] Youming L. and Robert C. :A new algorithm and asymptotical properties for deadlock de-tection problem for computer systems with reusable resource types. Advances and innova-tions in systems, computing sciences and software engineering. Springer, Heidelberg, pp. 509–512, (2007).

[6] Youming L. :A Modified Banker's Algorithm. Innovations and Advances in Computer, Information, Systems Sciences, and Engineering Lecture Notes in Electrical Engineering, vol. 152, pp 277-281, (2013).

[7] Yuan X., Xu H. and Qiao S.: Research of Improved Banker Algorithm. International Con-ference on Graphic and Image Processing (ICGIP 2012), edited by Zeng Zhu, Proc. of SPIE, vol. 8768, (2012).

[8] Louise E. Moser P. M. Melliar-S.: The World Banker's Algorithm, Journal of Parallel and Distributed Computing,vol. 9, no. 4, pp. 369-373, (1990).

[9] Mark L., Spyros R., Placid F.: The Application and Evaluation of Banker's Algorithm for Deadlock-Free Buffer Space Allocation in Flexible Manufacturing Systems, International Journal of Flexible Manufacturing Systems, vol. 10, no. 4, pp 73-100, (1998).

[10] Lang S.-D.: An extended banker's algorithm for deadlock avoidance, IEEE Transactions on Software Engineering, vol. 25, no. 3, (1999).

[11] Chang W. K., Tanchoco J. M. A., and Pyung-H. K. :Deadlock Prevention in Manufacturing Systems with AGV Systems: Banker's Algorithm Approach, J. Manuf. Sci. Eng 119(4B), pp. 849-854, (1997).

[12] Gang X. and Wu Zhiming: Deadlock Avoidance Based on Banker's Algorithm for FMS, InternationalJournal of Advanced Manufacturing. Technology, 1998, 16(1).

[13] Dhamdhere D.M.: Operating Systems: A Concept-based Approach, Tata McGraw-Hill Education, (3rd ed.), (2006).

[14] G. Maione and F. DiCesare "A Petri Net and Digraph-Theoretic Approachfor Deadlock Avoidance in Flexible Manufacturing Systems", 1998 IEEE International Conference on Systems, Man, and cybernetics,International Journal of Engineering and Applied Computer Science (IJEACS) Volume: 02, Issue: 02, February 2017

[15] E. Emeka, S. Ghosh, M. Iqbal, and T. Asos Dagiuklas "Improved Deadlock Prevention Algorithms in Distributed Systems Mahboobeh Abdoos Reliable Resource Provisioning Using Bankers' Deadlock Avoidance Algorithm in MEC for Industrial IoT", IEEE Access, 2018.

[16] Chao Gu, Zhiwu Li, and Abdulrahman Al-Ahmari, "A Multistep Look-Ahead Deadlock Avoidance Policy for Automated Manufacturing Systems",Discrete Dynamics in Nature and Society, (2017), Article ID 8687035 .

[17] A W Salim, F Wiranata, C M Mahidin, R F Waruwu, Vikram, S Wardani, A.Dharma, "Implementation Resource Request Alghoritm In Simulation of Deadlock Avoidance", Journal of Physics, (2019).

[18] S. Pang, H.ongqi Chen, H.ao Liu, J. Yao, and M. Wang, "A deadlock resolution strategy based on spiking neural P systems", Journal of Ambient Intelligence and Humanized Computing, (2019)