IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 10, Issue 5 (Mar. - Apr. 2013), PP 105-110
www.iosrjournals.org

Comparative Study of RBFS & ARBFS Algorithm

Disha Sharma[1], Sanjay Kumar Dubey[2]
[1](*Information Technology, Hindustan College of Science & Technology, Farah, Mathura, India)*
[2](*Assistant Professor, Amity University, Noida , India)*

Abstract : *RBFS is a best-first search that runs in space that is linear with respect to the maximum search depth, regardless of the cost function used. This algorithm allows the use of all available memory. One major flaw of this algorithm is that it can visit the same node several times. This makes the computational cost may be unaffordable for some accuracy. This problem can be much solved by using various extension of Heuristic Search. By using anytime approach in our simple RBFS will improve solution by saving the current solution & continues the search.*

Anytime RBFS algorithm will optimize the memory & time resources and are considered best for RBFS algorithm. When the time available to solve a search problem is limited or uncertain, this creates an anytime heuristic search algorithm that allows a flexible substitution between search time and solution quality. Some applications like Real Time Strategy (RTS) games have applied these algorithms to find a fast solution that will help the algorithm prune some paths during the subsequent computation.

Keywords *-AI Algorithm, Anytime, Best-first search, Computational time, Heuristic Search*

I. Introduction

Recursive Best-First Search or RBFS, is an Artificial Intelligence Algorithm that belongs to heuristic search algorithm [1]. It expands fronteir nodes in best-first order. It uses the problem specific information about the enviroment to determine the preference of one node over the other [2].

RBFS is similar to a recursive implementation of depth-first search, with the difference that it uses a special condition for backtracking that ensures that nodes are expanded in best-first order [3]. It works by maintaing on the recursion stack the complete path to the current node being expanded, as well as all immediate siblings of nodes on that path, along with cost of the best node in the subtree explored below each sibling [4]. Whenever the cost of the current node exceeds that of some other node in the previously expanded portion of the tree, the algorithm backs up to their deepest common ancestor, and continues the search down the new path [2].

RBFS explores the search space by considering it as a tree. An example of the search space with cost equal to depth is shown in Fig. 1.

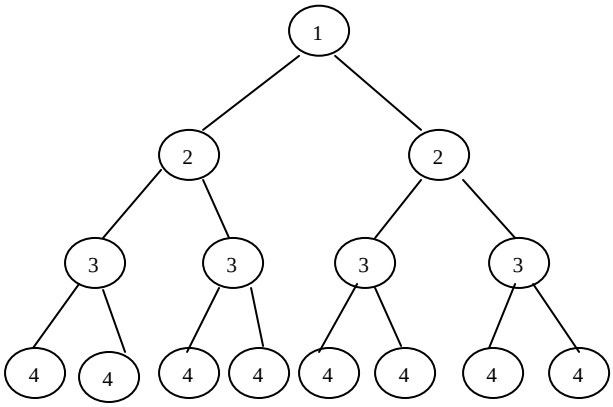

figure 1 : example of search space with costs equal to depth

Fig. 2 shows the sequence of steps RBFS makes while exploring this tree. The Fig. 2a shows the call on the root and generation of its children. As both children are of equal quality, the left (the first one) is examined first, with the bound 2 i.e. minimum of its parent and its best sibling (Fig. 2b). As both of its descendants exceed this bound, the algorithm backtracks and starts examining the right subtree, but it stores the cost of the minimum bound breaking node (3) in the left subtree (Fig. 2c). Fig. 2d and 2e show an expansion of the right subtree, while Fig. 2f gives details when algorithm returns from the right subtree, updates its stored value to 4, and starts regenerating the left subtree. Since the left subtree has already been explored up to the cost of 3, it is now safe to

move up to that bound in depth-first manner (Fig. 2g) and proceed from there in a best-first manner again (Fig. 2h and 2i).

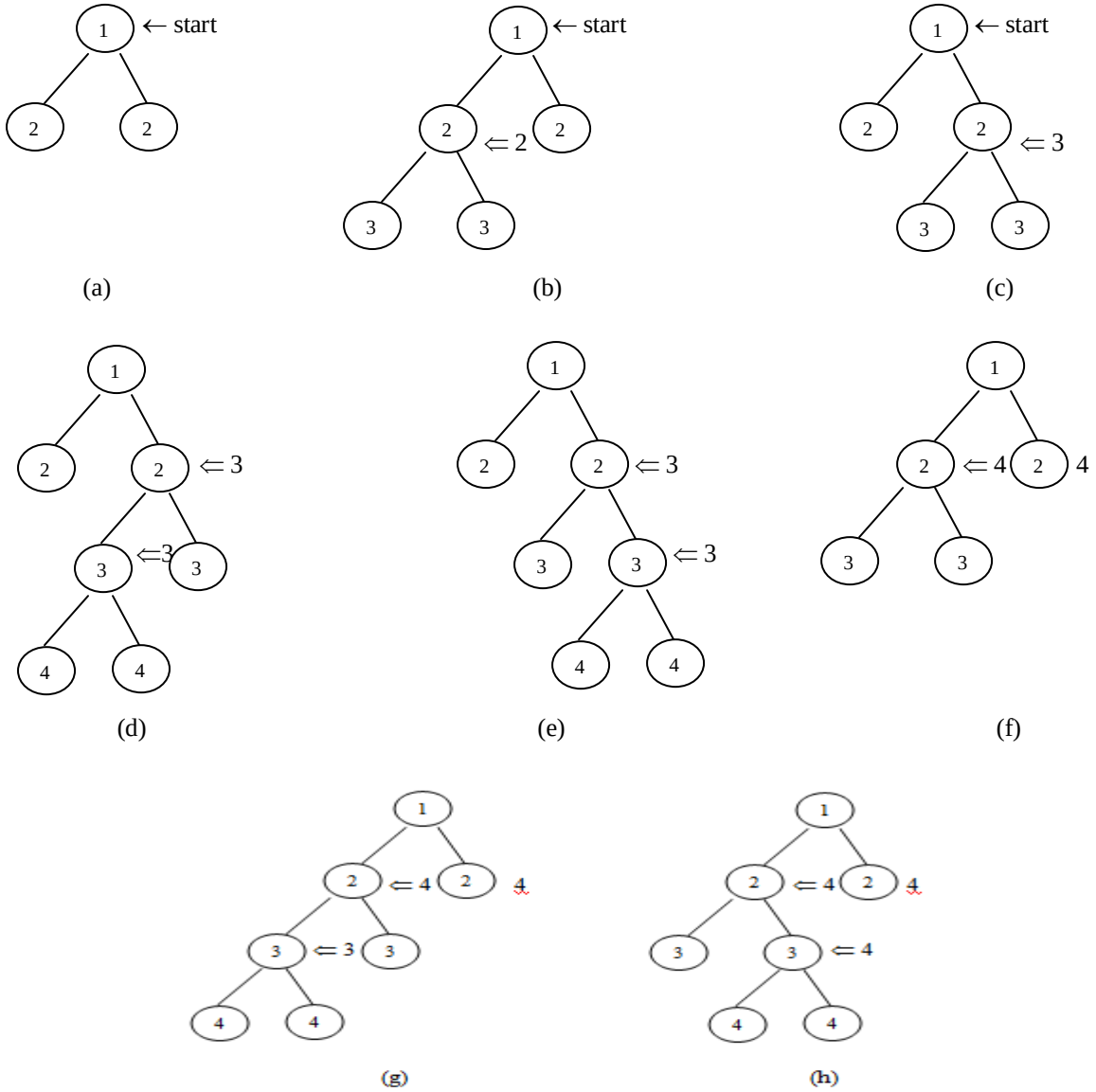

figure 2 : The sequence of steps RBFS makes when explring a tree with cost equal to depth

RBFS is called with two arguments : a node to expand and an upper bound on the cost of the node's subtree. The RBFS algorithm works by keeping track of an upper bound. Thgis upper bound allows the algorithm to choose better paths rathe tha continuing indefinitely down the current path [4].

This upper bound keeps track of the f-value of the best alternative path available form any ancestor of the current node[5]. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of the best leaf [4].

To achieve the described behaviour, besides static heurictic estimate f(n), RBFS maintains also backed-up value $F_b(n)$ for each node [6]. $F_b(n)$ is initialized with f(n) when the node is generated for the first time. When the algorithm explores the whole subtree of the node up to the given bound and returns, this value is updated to the minimum cost of the fronteir nodes in this subtree, that is with the cost of the node which minimally exceeds the given bound. In this way the essential information on the quality of the subtree is preserved for possible subsequent regenerations. The updated value $F_b(n)$ of the node serves also as an indicator if the node has been already explored. If $F_b(n) > f(n)$ then the node n has been explored previously and the F_b values for the children of n can be set to the maximum of their parent updated value $F_b(n)$ and their respective static values f. In this way we are allowedr to search already explored space in the depth-first manner, without unnecessary backtracking the best first behaviour would probably require [7].

II. Comparison of Algorithms

The memory limitation of the heuristic path algorithm can be overcome simply by replacing the best-first search IDA* using the same weighted evaluation function [9]. However, with u ≥ ½, IDA* is no londer a best-first search, since the total cost of a child can be less than that of its parent, and thus nodes are not necessarily expanded in best-first order. An alternative algorithm is Recursive Best First Search(RBFS).

RBFS is a best-first search that runs ina space that is linear with respect to the maximum search depth, regardless of the cost function used [1]. Even with an admissible cost function, RBFS generates fewer nodes than IDA*, and is generally superior to IDA*, except for a small increase in the cost per node generation.

We present a novel, RBFS based algorithm which uses all available memory. This is achieved by keeping the generated nodes in memory and pruning some of them away when the memory runs out. This new RBFS algorithm is implememted using Anytime Algorithm and will be called as Anytime Recursive best first search.

2.1 Existing Study

Recursive best-first search, or RBFS is a general heuristic search algorithm that expands frontier nodes in best-first order, but saves memory by determining the next node to expand using stack-based backtracking instead of by selecting from an Open list [8].

Table 2.1 below is the pseudo code of the RBFS algorithm that we have implemented. The RBFS algorithm is called with the root and the goal node as its parameters plus with the limit parameter which provides us with an upper limit on the f-costs of the nodes in the selected path [3]. In the RBFS algorithm, the root node is first compared with the goal node and if root node matches with the goal node then the algorithm returns with the goal success. However if no match found then the root node is expanded using the *ExpandNode* function. The basic functionality of the *ExpandNode* function alongside its pseudo code is also explained below in Table 2.2. The *ExpandNode* function returns the successor nodes of the root node with all the calculations of their f-costs.

Then we check whether the least f-cost of these successor nodes is less then the limit or not? If yes, then these successor nodes are added in the Open list and sorted with their increasing f-costs. Also the root node is added in the Close list. The successor node with the least f-cost is then taken as a best node and the successor node with the second least f-cost is taken as an alternative node. We recall the RBFS algorithm with the best and the goal node plus the alternative node's f-cost as a limit, and this process continues recursively until we find the goal node.

However if the least f-cost of the successor nodes is greater then the limit, then that means that we already know a path with lower f-cost then this one, thus the RBFS algorithm backtracks and updating the node with the least f-cost of its successors.

```
public double RBFS(Node Root, Node Goal, double limit)
{
if(Root==Goal)
{
goalSucc=true;
return Root.fCost;
}
else
{
Node [ ] successors=ExpandNode(Root);
successors.SortNodes(); /*sorts successors nodes by increasing g fCost*/
if(successors[firstNode].fCost>limit)
return successors[firstNode].fCost;
else
{
closeList.Insert(Root);
foreach(Node s in successors)
{
if(s!=closeList[item])
openList.Insert(s);
}
openList.Sort();
Node bestNode=openList.RemoveFirstNode();
Node alternativeNode=openList.RemoveFirstNode();
```

```
while(goalSucc==false)
{
bestNode=RBFS(bestNode,Goal,Math.Min(limit, alternativeNode.fCost));
openList.Insert(bestNode);
list.Sort( );
bestNode=openList.RemoveFirstNode();
alternativeNode=openList.RemoveFirstNode();
}
}
}
}
```

Table 2.1 : Pseudo code of the RBFS function of the RBFS algorithm

```
public Node[] ExpandNode(Node Root)
{
Node[8] succ;
foreach (Node s in succ)
{
s.state=a neighbouring point of the root;
s.parent=root;
s.gCost=Root.gCost+step cost from Root to s;
s.fCost=s.gCost+Heuristic(s, Goal);
}
}
```

Table 2.2 : Pseudo code of the ExpandNode function of the RBFS algorithm

2.2 Proposed Study

The process of making the RBFS algorithm anytime, is simple. RBFS algorithm uses the control manager class (*ctr_manager class*) to convert into ARBFS i.e Anytime Recursive Best-First Search. This *ctr_manager class* not only adds the time limit to the RBFS algorithm but also controls the stopping and the restarting of the RBFS algorithm. After the anytime algorithm, it finds a solution, and each time it finds an improved solution, it saves the solution and continues the search.

In the pseudo code given in Table 2.3, *ctr_manager class* implements a thread to add the time limit to the RBFS algorithm. The most important variable in this *ctr_manager class* is that of *new_Solution* which holds the best solution found so far by the RBFS algorithm. The *ctr_manager class* can be called using the *thread_ARBFS* function which takes time limit, root node, goal node and the upper limit as its parameters. The *thread_ARBFS* function then starts the thread with the time limit and the RBFS algorithm with root node, goal node and the upper limit as its parameter. When the RBFS algorithm starts, it keeps on checking for more improved solutions and placed it in the *new_Solution.* When the time limit is up the thread will set the *goalSucc* variable of the RBFS algorithm and the algorithm stops.

However if now we want to restart the algorithm the *ctr_manager class* will check whether the RBFS algorithm finds an improved solution when run previously or not? If yes then it will clear the Open and the Close lists and restart the thread with the new time limit, and also restart the RBFS algorithm with new improved solution as the root node. However if the RBFS algorithm didn't find any improved solution previously then *ctr_manager class* will not clear the Open and the Close lists, allowing the RBFS algorithm with more time to find the improved solution using these lists.

```
public class ctr_manager
{
bool is_First=true;
Node new_Solution=null;
public void run_rbfs()
{
thread.sleep(sleep_Time);
goalSucc=true;
}
public void thread_ARBFS(int sleep_Time, Node r N, Node g N, double limit)
{
thread_r=new thread();
```

```
if(is_First)
{
new_Solution=rN;
thread.Start(run_rbfs);
RBFS(rN, gN, limit);
}
elseif(rN==new_Solution)
{
// keep all the states intact because algorithm needs more time
// to find a better solution
thread.Start(run_rbfs);
RBFS(rN, gN, limit);
}
elseif(rN!=new_Solution
{
// algorithm finds an improve solution, clears all the previous states
// to allow algorithm a fresh start
closeList.RemoveAll();
openList.RemoveAll();
thread.Start(run_rbfs);
RBFS(new_Solution, gN, limit);
}
}
}
```

Table 2.3 : Pseudo code of the ctr_manager class of the anytime RBFS algorithm

III. Analysis

The memory complexity of RBFS is O(db), where d is the depth of the search and b is the branching factor [9]. Instead of continuing down the current path as far as possible, as in ordinary depth-first search, RBFS keeps track of the *f*-cost of the best alternative path available from any ancestor of the current node, which is passed as an argument to the recursive function. RBFS expands the nodes in best-first order even when the evaluation function is nonmonotonic.

RBFS algorithm can be easily declare as Anytime RBFS algorithm only when it must satisfy all properties of Anytime algorithms[3]. There are few such properties on the basis of which we can judge are as follows:

i. Measurable quality: This property means that the quality of the solution that an anytime algorithm returns, should be measureable and represent able in some way. In our implemented anytime A* algorithm we measure this quality of the solution in terms of its straight line distance from the goal node.
ii. Mono-tonicity: This property means that the quality of the solution that an anytime algorithm returns, should increases with increase in computational time and quality of the input.
iii. Consistency: According to this property the quality of the solution of an anytime algorithm is connected with computational time it have and the quality of the input.
iv. Interrupt-ability: This property means that for an algorithm to be declared as anytime we should be able to stop it at any time and it should provides us with some solution. Our implemented anytime RBFS algorithm is stoppable at any time and it will return a solution whenever it stops.
v. Preempt-ability: According to this property an anytime algorithm can be stopped and can also be restarted again with minimal overhead.

IV. Conclusion

RBFS is a general heuristic search algorithm that expands frontier nodes in best-first order. The main problem with this algorithm is that it uses a special condition for backtracking that ensures that nodes are expanded(for the first time) in best-first order. We have presented a simple approach for converting a heuristic search algorithm RBFS into an anytime algorithm that offers a tradeoff between search time and solution quality. The approach uses a control manager class which takes care of the time limit and the stopping and restarting of the RBFS algorithm to find an initial, possibly suboptimal solution, and then continues to search for improved solutions until meeting to a provably optimal solution. It also bounds the sub-optimality of the currently available solution.

The simplicity of the approach makes it very easy to use. It is also widely applicable. It can be used with other search algorithms that explore nodes in best-first order. Thus, we conclude that anytime heuristic search provides an attractive approach to challenging search problems such as Real Time Strategy(RTS) games, especially when the time available to find a solution is limited or uncertain. Anytime variants of RBFS algorithm keep all the nice properties of the original algorithm, but have additional power when there is enough memory available. If the savings in node regeneration are big enough to balance larger time per node generation, it is easy to automatically switch back to ARBFS behavior.

References

[1]. Sree Kanth R. K., (2012) « Artificial Intelligence Algorithms », IOSR Journal of Computer Engineering(IOSRJCE), volume 6, issue 3, sep-oct, pp 1-8
[2]. Korf, R.E., (1993), « Linear space best first search », Artificial Intelligence, volume 62, issue 1, pp 41-78
[3]. Eric A. H., Rong Z., (2007), « Anytime Heuristic Search », Journal of Artificial Intelligence Research, volume 28, pp 267-297
[4]. Russell, S.J: Norvig, (2003), Artificial Intelligence: A Modern Approach, N,J: Prentice Hall, page 97-104, ISBN 0-13-790395-2
[5]. Keindl H., Kainz G., Leeb A., Smetana H., (1995)« How to use limited memory in heuristic search », in Proceedings IJCAI-95, ed., Cris S. Mellish, pp 236-242
[6]. Keindl H., Leeb A., Smetana H., (1994) »Improvements on linear space search algorithms » in Proceedings ECAI, volume 94, pp 155-159
[7]. Sen A. K. & Bagchi A., (1989) »Fast recursive formulations for best-first search that allows controlled use of memory », in Proceedings IJCA, volume 89, pp 297-302
[8]. Korf, R.E., (1995) « Space-efficient search algorithms », Computing Surveys, volume 27, issue 3, pp 337-339
[9]. Ghosh S., Mahanti A., Dana S.N., (1994) « ITS : An efficient limited- memory heuristic tree search algorithm », in Proc. Twelfth National Conference on Artificial Intelligence