COMPARISON OF VARIOUS HEURISTIC SEARCH TECHNIQUES FOR FINDING SHORTEST PATH

Mr. Girish P Potdar[1], Dr.R C Thool[2].

[1]Associate Professor, Computer Engineering Department, P.I.C.T, Pune,
[2]Professor, Department of Information Technology, SGGS IE&T, Nanded,

ABSTRACT

Couple of decades back, there was a tremendous development in the field of algorithms, which were aimed at finding efficient solutions for widespread applications. The benefits of these algorithms were observed in their optimality and simplicity with speed. Many of the algorithms were readdressed to solve the problem of finding shortest path. Heuristic search techniques make use of problem specific knowledge to find efficient solutions. Most of these techniques determine the next best possible state leading towards the goal state by using evaluation function. This paper shows the practical performance of the following algorithms, to find the shortest path:Hill Climbing, Steepest-ascent, and Best-First and A. While implementing these algorithms, we used the data structures which were indicated in the original papers.In this paper we present an alternative data structure multi-level link list and apply the heuristic technique to solve shortest path problem. This was tested for class of heuristic search family-- A* and Best First Search approaches. The results indicate that use of this type of data structure helps in improving the performance of algorithms drastically.*

Keywords:

Multilevel link list, Informed search techniques, Heuristic function, Shortest path algorithm.

1.INTRODUCTION

Heuristic search algorithms have exponential time and space complexities as they store complete information of the path including the explored intermediate nodes. Hence many applications involving heuristic search techniquesto find optimal solutions tend to be expensive. Despite of these, the researchers have strived to find optimal solution in best possible time. In this paper we have considered major algorithms which are applied to find the shortest path: hill – climbing, steepest –ascent, best first and A* [1,2,4].

Hill climbing algorithms expand the most promising descendant of the most recently expanded node until they encounter the solution. Steepest – ascent hill climbing differs from hill climbing algorithm only the way in which the next node is selected. In this method it selects best successor node for expansion, unlike the first successor node for expansion, as done in hill climbing. Though this method tries to choose best possible path , but this method , like hill climbing method may fail to find a solution by reaching to a node from were no improvements can be done [5,8]. Best first search method selects the "best" node for further expansion by applying a

DOI : 10.5121/ijaia.2014.5405

heuristic function. It then generates the successor node in similar fashion till the goal node is reached. This technique tries to explore the advantages of breadth first and depth first search technique and provides better time bound solution. Best first algorithm involves OR graph, it avoids the node duplication and also works on the assumption that each node has parent link to give the best node from the node where it is derived and link to successors. A* algorithm is a slight modified version of best search algorithm. The difference is that in A* the estimate to the goal state is given by heuristic function and also it makes use of the cost of the path developed [2,3,6].

We will now discuss each of these methods for finding the shortest path.

2.HILL CLIMBING METHOD FOR SHORTEST PATH FINDING

Hill climbing algorithm expands one node at a time beginning with the initial node. Each time it expands only the best node reachable from current node. Thus this method does not involve complex computation and due to this reason cannot ensure the completeness of the solution. Hill climbing method does not give a solution as may terminate without reaching the goal state [12].Now let us look at algorithm of hill climbing for finding shortest path:

Procedure for hill climbing algorithm to find the shortest path:

```
hill_climb_sp (s, g, Q)
{
// s& g are start and goal nodes respectively.
// Q is queue which stores the successor
  //  nodes.
// let curr_node indicate current working
//   node.
// path _cost gives the cost of the path.

initialiseQ;
curr_node = s;
path_cost=0;
while (1)
{
if (curr_node is goal node) then
terminate the process with SUCCESS;
else
{
find successor node of curr_node;
addthis node in Q ;
}

if(Q is empty)then
terminate the process with FAILURE;
else
{
temp_node = first node of Q ;
```

```
path_cost = path_cost  +
        edge_cost [curr_node][temp_node];
curr_node = temp_node;
delete first node from Q ;
}
}
```

One may notice that there can be failure state when algorithm may fail to reach the goal node. This will happen especially when the processing has reached to a node from where no new best nodes are available for further expansion. This will happen especially when the processing has reached to a node from where no new best nodes are available for further expansion.

3. STEEPEST ASCENT HILL CLIMBING METHOD FOR SHORTEST PATH FINDING

This method is a result of variation in hill climbing. Here, instead of moving the immediate best node, all the reachable nodes from current node are considered and among these the best one is chosen. In case of simple hill climbing, the first successor node which is better, is selected, due to this we may omit the best one. On the contrary steepest ascent hill climbing method not only reaches to the better state but also climbs up the steepest slope.

The variation in algorithm will be only in finding the best successors node from all the possible successor nodes from all possible successor, and not just the first best node [2,12,15].

The algorithm is given below:

```
steep_asc_hll( s, g , Q )
{   //  s& g are stait and goal nodes respectively.
  //  Q is queue which stores the successor
  //  nodes.
//   let curr_node indicate current working        //   node.
// path _cost gives the cost of the path.

initialise Q;
curr_node = s;
path_cost=0;

while (1)
{

if (curr_node is goal node) then
terminate the process with SUCCESS;
else
  {
find all the reachable node from curr_node;
determine the cost of reaching to these nodes
fromcurr_node;
according their cost add them in Q.
 }
if (Q is empty) then
```

```
terminated the process with FAILURE;
else
{    temp_node = first node of Q ;
path_cost = path_cost  +
          edge_cost [curr_node][temp_node];
curr_node = temp_node;
delete first node from Q ;
 }

}
```

One can notice that hill climbing and steepest – hill climbing may fail to find a solution. Either algorithm may not reach goal node as it may reach to a node where we may not find better nodes. In such cases we may need to back-track as use more rules before choosing the next node. However this process will be time consuming.

Both the methods discussed, may terminate not by finding a goal node but may reach node from where no better nodes can be generated.

This will happen if the processing has reached to one of the following situations:

i) A node might have been selected which may be better that its neighbors, however there may be few better nodesavailable which are step away. This situation is termed as local maxima.

ii) A node might have been selected, whose neighbors may have the same value and hence choosing next best node is difficult. This is known as plateau.

iii) A ridge is a special kind of local maximum, though the path selected so far may be the best, yet making further moves difficult.

The next algorithms described here try to overcome these problems.

4. BEST FIRST METHOD FOR SHORTEST PATH FINDING

Best first search is a type of graph search algorithm. Here the nodes are expanded one at time by choosing lowest evaluation value. This evaluation value is a result of heuristic function giving a measure of distance to the goal node. For typical applications such as shortest path problems, the evaluation function will be accurate as it accounts for distance or an absolute value [14,19].

Best first search is a combination of breadth and depth first search. Depth first search has an advantage of arriving at solution without computing all nodes, whereas breadth first arriving at solution without search ensured that the process does not get trapped. Best-first search, being combination of these two, permits switching between paths. At every stage the nodes among the generated ones, the best suitable node is selected for further expansion, may be this node belong to the same level or different, thus can toggle between depth-first and breadth-first. This method involves OR graph, avoids node duplication, and also requires two separate lists for processing. OPEN list keeps the nodes whose heuristic values are determined, but yet to be expanded. CLOSE list have the nodes which have been already checked, further these nodes are kept in this list to ensure no duplications. It implies that the OPEN list has the nodes which need to be considered for further processing and the entries in CLOSE list indicate the nodes which may not be re-required in further steps [6,7].

Let us look at the best first search algorithm for finding shortest path:

```
bfs_sp (s, g)
{
        // s is start node &  g is goal node
        // let OPEN and CLOSE be the two lists.
        // let current_w indicates current working //node.
        // path_cost indicates cost of reaching to a    // node x.

        path_cost=0;
        OPEN=NULL;
        CLOSE=NULL;

        do
        {
                add_node (OPEN, s);  //add S to
                                         //OPEN list;
                current_w= first element of OPEN;
                determine f(n) for successor nodes of current_w;
                add these new nodes to OPEN based on their f (n) values;
                movecurrent_w to CLOSE;
                current_w= first node of OPEN;
                path_cost=path_cost + f(n) of current_w;
        }
                while (current_w is not g and OPEN is not empty);
                If (current_w=g) then
                printpath_cost;
                else
                print failure;

}
```

In this case f (n) a heuristic function is an actual edge cost function.

5. A* ALGORITHM FOR SHORTEST PATH FINDING

We know that the various search techniques are designed, tested and are being used for various purposes whatever it is for system software or application software. But the base for this is however mainly because of the problems in planning domain. Classical approaches to heuristic search algorithm work on assumption of the existence of deterministic model of sequential decision making leading to the solution. The research work focused on solving planning problems under uncertainty [1]. Heuristic algorithms have given a new looked into the problems belonging to this domain [6,10].

The shortest path problem can be solved by A* algorithm. The heuristic function needs to evaluate two costs, g and h. Let g(n), in shortest path problem, represent cost of choosing the path from starting node to node n; and h(n) represents optimal cost of node n to the goal node. Now the cost of node n is given by: $f^*(n) = g(n) + h^*(n)$. However the value of $h^*(n)$ will be unknown in most of the situations, which results in unknown value of $f^*(n)$. A* algorithm, however makes a best approximation for $h^*(n)$[16,17].

The A* algorithm to solve the shortest path problem can be written as: [10]

Step 1: Start from the start node; place it in OPEN list. This will be current working node.

Step 2: Explore all the nodes adjacent to the one in OPEN list.

Step 3: Determine the cost function for all the nodes obtained in step 2; and place them in OPEN list in increasing order of cost function values.

Step 4: Move current working node, from OPEN list to CLOSE list.

Step 5: Now the first node in OPEN List will be the current working node (which is having least cost function due to insertion criteria in step 3).

Step 6: If this current working node is not the goal state (final node), then repeat step 2 to step 5.

Step 7: The CLOSE list gives the shortest path and the value of last cost function obtained gives the optimal cost.

6. EXPERIMENTAL RESULTS

All the algorithms discussed in previous sections were implemented in C++ and run on 2.4 GHz Intel C2D system with 2GB RAM. The random data sets were created for varying number of input nodes and saved in separate files. While testing these algorithms stored data was given as input data and processed. The algorithms were tested for the number nodes and edges explored/visited were compared. The Number of nodes and edges considered during the process for various algorithms are given in Table 1 and Table 2 respectively.

Note: HC –Hill Climbing, ST_AC --Steepest Ascent
Hill Climbing,BFS—Best First Search and A*.

Table 1: Number of nodes considered.

Nodes	HC	ST_AC	BFS	A*
1000	1439	915	351	183
1100	2171	1006	782	426
1200	2391	1106	1806	1029
1300	2533	1214	762	401
1400	2787	1310	1871	1057
1500	2923	1394	2360	1346
1600	3023	1510	363	188
1700	3339	1598	2696	1536
1800	3373	1722	2876	1634
1900	3749	1820	219	111
2000	3875	1921	2767	1567
2100	4063	2015	1952	1065
2200	4277	2106	3683	2137
2300	3557	2198	3713	2132
2400	4751	2301	1850	1017
2500	4959	2388	2683	1479
2600	4995	2502	3641	2074
2700	5275	2608	1885	1039
2800	5257	2704	2163	1190
2900	5705	2805	1712	922
3000	5945	2900	1637	876

Table-1 shows that there is significant amount of improvement on number of nodes being considered in A* algorithm compared to the rest of the methods.

Table 2: Number of edges considered

Nodes	HC	ST_AC	BFS	A*
1000	919881	492103	318260	165347
1100	1207619	595531	698507	376975
1200	1437589	709383	1273917	702929
1300	1686345	835800	839059	439800
1400	1957171	970824	1663677	920304
1500	2245595	1112690	2003375	1112315
1600	2549145	1267835	545470	281622
1700	2885731	1432627	2585858	1430620
1800	3223745	1611151	2905999	1605239
1900	3605601	1795683	402089	202895
2000	3992219	1990342	3443484	1905039
2100	4401245	2193714	3072621	1667820
2200	4831941	2408930	4339274	2416884
2300	5014481	2633403	4721867	2629654
2400	5754649	2867561	3522853	1921764
2500	6244621	3112370	4758366	2602019
2600	6744499	3368084	5818135	3240099
2700	7280819	3631545	4146417	2263359
2800	7805331	3907490	4805723	2621745
2900	8402039	4191736	4195443	2246319
3000	8993299	4487278	4208595	2241750

Table-2, as a consequence of nodes expanded or considered results in varying number of edges. In BFS and A* the edge count reduces, as we make proper heuristic estimation. Where as in Hill climbing or in Steepest Ascent, we try to choose the immediate best node, which will ultimately result in exploring more number of edges.

The resulting graphs of the two algorithms are given in Fig 1 and Fig 2.

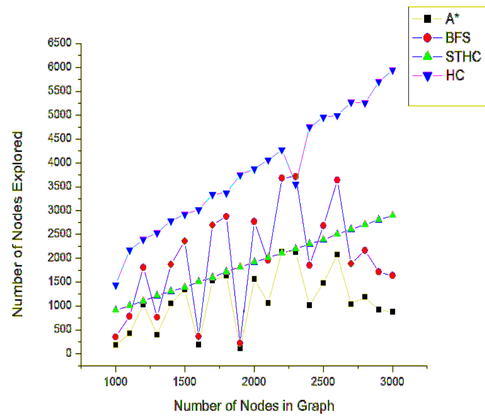

Figure 1: Comparison of number nodes considered against total nodes in graph.

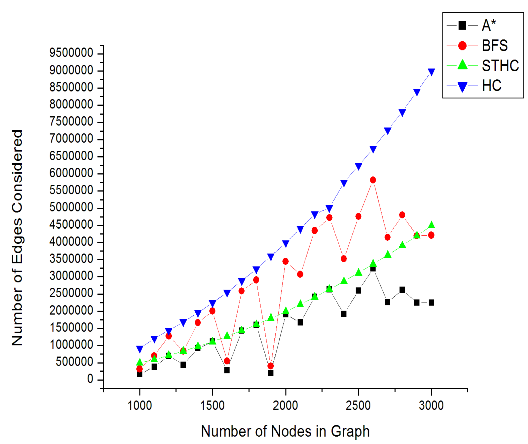

Figure 2: Comparison of number edges considered against total nodes in graph

One may also observe here that certain unexpected variations in the values. This is mainly due to the fact that these algorithms were executed till they find the solution and were not run for fixed number of iterations.

7.BEST FIRST SEARCH USING MLL AS DATA STRUCTURE

Now let us look at the variations to the algorithms presented in section 4 and 5. Here let us make use of multi-level linked list as data structure for implementation [8,18,20].

In section 4, we discussed the conventional approach of Best first search for finding shortest path. In this section we present slightly modified approach for solving the problem of shortest path finding. We store the current working node in parent list of MLL and all the adjacent nodes in its successor list. The best node as determined by f(n), will be chosen for further expansion.
f(n) = min(cost(n,i)), ∀i, where i is an adjacent node of n.
The skeleton of the algorithm is given below:

Best First method with multi-level linked list:

```
bfs_mll_sp (s, g)
{
  // s is start node &  g is goal node
  // begin the process from s; this will be the
 //first node in MLL, let it be current //working  node call it  current_w
     //path_cost indicates cost of reaching to a
   // node x.
        path_cost=0;
        do
        {
                determine f(n) for successor nodes of current_w; add these new nodes to
                successor link S   based on their f (n) values for the current parent node;
                current_w= first node of S;
                path_cost=path_cost + f(n) of current_w;
        }while (current_w is not g and S
        of current parent node is not empty);
                If (current_w=g) then
                printpath_cost;
                else
                print failure;
}
```

In this case f (n) a heuristic function is an actual edge cost function.

8. A* ALGORITHM USING MLL AS DATA STRUCTURE

As stated earlier, shortest path problem aims at finding minimum cost path between pair of nodes cumulatively and then find the final path between start and goal nodes[9,21]. A* algorithm with MLL, resultin pruningthe search space [8,17]. The approach which has been followed in our work makes use of an exact accurate function. The evolution function f(n) is given as:

f(n)=g(n) +h(n); where g(n) is the cost of an edge between the currently explored node or current working node and the node n being examined, h(n) is the best edge cost value from the set of edge costs going out from the node n to the all possible adjacent nodes.

Let us look at the algorithm.

Step 1: Start from start state; this will become the first node in MLL, call it as current working node. Since this is the first node this will be the first node of parent list.

Step 2: Explore all the nodes adjacent to the one in the current parent node list.
Step 3: Determine g value of the current working node.
Step 4: Obtain the h values for all the nodes obtained in step 2.
Step 5: Find the f values for the expanded nodes; keep them according to their values in successor list for the current parent node (this will result in the list maintained in increasing order of the f values, which will be the cost function).
Step 6: Pick up the first node from the successor list obtained in step 5 which will be the next working node.
Step 7: If this current working node is not the goal state, then attach this node to parent list and repeat step 2 and step 6.
Step 8: The set of nodes belonging to parent list gives the shortest path and the cost function determined in the last step will be the optimal cost.

9. EXPERIMENTAL RESULTS OF BFS AND A* WITH MLL:

The BFS and A* algorithms discussed in previous sections were implemented in C++ and run on 2.4 GHz Intel C2D system with 2GB RAM. The random data sets were created for varying number of input nodes and saved in separate files. While testing these algorithms stored data was given as input data and processed.

The algorithms were tested for the number nodes and edges explored/visited were compared. The Number of nodes and edges considered during the process for various algorithms are given in Table 3 and Table 4 respectively.

Table 3: Number of nodes considered.
[Existing approach indicates the conventional approach that is being implemented and in MLL we used new method]

Nodes Considered	BFS		A*	
	Existing Approach	Using MLL	Existing Approach	Using MLL
1000	351	20	183	17
1100	782	21	426	22
1200	1806	30	1029	20
1300	762	28	401	27
1400	1871	53	1057	21
1500	2360	37	1346	23
1600	363	47	188	22
1700	2696	64	1536	25
1800	2876	32	1634	25
1900	219	33	111	28
2000	2767	56	1567	30
2100	1952	82	1065	27
2200	3683	27	2137	31
2300	3713	52	2132	30
2400	1850	33	1017	32
2500	2683	51	1479	32
2600	3641	77	2074	30
2700	1885	33	1039	33
2800	2163	56	1190	32
2900	1712	40	922	37
3000	1637	40	876	34

Table 4: Number of edges considered.
[Existing approach indicates the conventional approach that is being implemented and in MLL we used new method]

Nodes Considered	BFS		A*	
	Existing method	Using MLL	Existing	Using MLL
1000	318260	8795	165347	7543
1100	698507	10394	376975	9248
1200	1273917	12667	702929	11022
1300	839059	14001	439800	13123
1400	1663677	17652	920304	15005
1500	2003375	18267	1112315	17141
1600	545470	22183	281622	19209
1700	2585858	27306	1430620	21460
1800	2905999	24276	1605239	23694
1900	402089	26541	202895	26087
2000	3443484	30619	1905039	28535
2100	3072621	37144	1667820	31004
2200	4339274	33394	2416884	33731
2300	4721867	37251	2629654	36516
2400	3522853	38437	1921764	39132
2500	4758366	42381	2602019	41959
2600	5818135	48610	3240099	44601
2700	4146417	46383	2263359	47632
2800	4805723	50453	2621745	50368
2900	4195443	52588	2246319	53422
3000	4208595	55831	2241750	56672

10.CONCLUSION

We have presented major class of heuristic algorithms. The comparison shows that though all these algorithms can be applied to find the shortest path, but should not be used unless there is a real- time, event driven actions are anticipated. The comparison gives us clear idea that best-first search and A* algorithms are very well suitable when goal node cannot be reached from all nodes. However there may be interesting scenarios that may come out when these algorithms are applied with different data structures.

The results clearly indicate that hill climbing or steepest ascent hill climbing algorithms are not suitable for problems such as shortest path finding. This is due to the fact that there is no assurance of getting final optimal solution for all the cases. Best first and A* algorithms on the other hand ensure optimal solution for limited graph size. For larger number of nodes these algorithms not only tend to take more time but the optimality factor may be of concern.

There are number of factors for using different data structure approach, in heuristic algorithm – as special case study we implemented Best First Search and A* algorithm with multilevel linked list as data structure is the main reason of the increased speed in determining the shortest path. One can easily figure out the fact that both these algorithms with multilevel lined list results in reduced area that is to be searched, which eventually gives us the better way of handling the nodes at runtime. Since the number of nodes or edges considered in the process is less, the time taken to find the optimal solution will also be less which are shown in the results section.

One may even work on eliminating the already explored nodes in subsequent levels, which may further reduce the space requirement.

REFERENCES

[1] BlaiBonet and Eric A. Hansen, (2010)“Heuristic Search for Planning under Uncertainty”, Chapter in Heuristics, Probability and Causality: A Tribute to Judea Pearl College Publications. pp 3-22

[2] Eric A Hansen, Rong Zhou, (2007) “Anytime Heuristic Search”, Journal of Artificial Intelligence Research 28, pp 267-297

[3] G.Cornuejols and G L Nemauser, (1978) “Tight bounds for christofides” travelling salesman heuristic* Short Communication Mathematical Programming, Vol. 14, Issue 1, pp 116-121

[4] Anne L. Gardner, (Sept 1980) “Search: An Overview”, AI magazine, Vol. 2, Number 1

[5] R. Korf, (1990) “Real time heuristic search”, Artificial Intelligence ACM Digital Library, Vol. 42, pp189-211

[6] RinaDechter and Judia Pearl, (July 1985) “Generalized Best-First Search Strategies and the Optimality of A*.”, Journal of the Association for Computing Machinery, Vol. 32, No. 3, pp 505-536

[7] L. Fu, D. Sun and L. R. Rilett, (2006) “Heuristic shortest path algorithms for transportation applications: State of the art”, Elsevier Computer and Operations research 33, pp 3324-3343

[8] Girish P. Potdar andDr.R.C.Thool, (2013) “An Alternate way of implementing Heuristic Searching Technique” International Journal of Research in Computer andCommunication Technology, Vol. 2, No 9, pp-793-795

[9] Hen-Yong Pang, Alicia Tang Y.C., (2006) “A Route Advisory System (RAS) For Travelling Salesman Problem”, Journal of Applied Sciences Research 2(1), pp 34-38

[10] C.H. Peng, J.S. Wangand R.C.T. Lee, (1994)“Recognizing Shortest Path Trees in Linear Time”, Information Processing Letters, Vol. 57, pp 77-85

[11] R.C.T. Lee, S.S. Tseng, R.C. Chang, Y.T. Tsai., (2012) “Introduction to Design and analysis of algorithms –A strategic approach”, Tata McGraw Hill edition 2012

[12] P.P.Chakrabarti,S. Ghose, A. Acharya and S.C. de Sarkar, (1989) “Heuristic search in restricted memory”, Artificial Intelligence, 41(2), pp 197-221,

[13] Herman Keindl, Angelika Leeb and Harald Smetana,(1994) “Improvements on linear space search algorithms”, in proceedings ECAI-94, pp 155-159,

[14] Richard E. Koff, (1993) “Linear space best-first search”, Artificial Intelligence, 62, pp 41-78

[15] A.Martelli, (1977) “On the search complexity of admissible search algorithms”, Al, Vol. 8, pp 1-13

[16] D. Dreyfus, (1967) “An appraisal of some shortest path algorithms”, Journal of the Operations Research Society of America,Vol. 17 Issue 3, pp 395-412

[17] A.V. Goldberg, (2001) "A simple shortest path algorithm with linear average time", In proceeding 9th ESA, Lecture notes in computer science LNSC 2161, pp 230-261

[18] B.V.Charkassy,A.V. Goldberg, T.Radzik, (1996) “Shortest Path Algorithms: theory and experimental evaluation”, Mathematical Programming, 73(2) pp 129-74.

[19] J.W.Lark, C.C.White III, K. Syverson., (1995) “A best first search algorithm guided by a set- valued heuristic”,IEEE Transactions on Systems, Man, and Cybernetics, Vol. 25, pp 1097-1101

[20] R.K.Ahuja, K. Mehlhorn, J.B.Orlin and R.E.Tarjan, (April 1990,) “Faster algorithms for shortest path algorithms”, Journal of the Association for Computing Machinery,Vol. 37, No. 2, pp 213-223

[21] D.P.Bertekas, (1991) “The auction algorithms for shortest paths”, SIAM J. Opt, Vol. 1, pp 425-447