

An Appropriate Search Algorithm for Finding Grid Resources

Olusegun O. A.¹, Babatunde A. N.², Omotehinwa T. O.³, Aremu D. R.⁴, Balogun B. F.⁵

^{1,4}Department of Computer Science University of Ilorin, Ilorin, Nigeria

^{2,5} Departments of Computer, Library and Information Sciences, Kwara State University, Malete, Nigeria

³ Department of Mathematical Sciences, Kogi State University, Anyigba, Nigeria

Abstract: *A grid is a collection of computing resources that perform tasks. Searching within a grid resource is very challenging considering the potential size of the grid and wide range of resources that are represented. The aim of this study is to find the search algorithm that is appropriate for a search problem in a grid environment. Breadth first search, Depth first search and Depth first iterative deepening search algorithms were studied and analyzed to find the one with minimal complexity. The algorithms were implemented using graph with a collection of nodes. They were analyzed based on their completeness and optimality in terms of running time. Octave was used as the analysis tool to measure the time complexity of these search algorithms. From the analysis of these algorithms, Depth First Iterative deepening search algorithm was adopted as a search strategy for grid resources because of its completeness and optimality in terms of running time. The research shows that Depth First Iterative Deepening search algorithm is asymptotically optimal in terms of cost of solution, running time, and space required for uninformed searches. Therefore, DFID search is the best search strategy suitable for grid resources among the three search strategy studied in this work.*

Keywords: Grid, Breadth first search, Depth first search, Depth first iterative deepening search.

1. Introduction

A grid is a collection of computing resources that perform tasks. It can also be defined as a distributed system that involves a large number of files. Searching within a grid resource is very challenging considering the potential size of the grid and the wide range of resources that are represented. Search algorithm is an algorithm for finding an item with specified properties among a collection of item. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure. It can also be defined as a procedure or set of steps for solving a particular problem by using the problem as input and returning a solution, usually after evaluating a number of possible solutions [1].

Analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to develop, maintain and execute them. Algorithm's efficiency is a function of the amount of computational resources it requires, measured typically as execution time and the amount of space or memory that the algorithm uses [2]. It is a measure of the resources that must be expended in developing, implementing and maintaining

an algorithm. Well-designed algorithm exhibits a minimum of unnecessary complexity.

Algorithmic complexity is concerned about how fast or slow particular algorithm performs. Complexity is a measure of the performance of an algorithm. It measures the internal and external factors. The internal factors include the time and space while the external factors includes the size of input to the algorithm, the speed of the computer and the quality of the algorithm [3]. Unmanaged complexity leads to algorithm difficult to use, maintain and modify. It caused increased development costs and overrun schedules. Algorithmic complexity is concerned about how fast or slow a particular algorithm performs.

2. SEARCH ALGORITHM

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem [4] Search algorithm is an algorithm for looking for a particular item with specified properties within a collection of item. Search algorithm can be classified into informed and uninformed searches. Informed search is an artificial intelligent search that uses information about the problem to guide the search, usually guesses the distance to a goal, but the search may not be always possible. Uninformed search algorithm is a search strategy that does not use the information about the problem to guide the search, i.e. it does not use any intelligent to guide the search.

In this study, the uninformed types of search algorithms are considered. This section describes and distinguishes between the various search algorithms. Each search engine uses algorithms that are slightly different, that is, why a search for one word or phrase would yield different results from different search engines. Some searches involve looking for something in a database like looking up your record in the database. Many types of searches are possible; some of the most common types of search algorithms are described in this study.

2.1 Criteria for evaluating search strategies

- **Completeness:** Is the algorithm guaranteed to find a solution when there is one? Hopefully it doesn't get tired and give up.
- **Optimality:** Does the strategy find the highest quality solution with least cost when there are several solutions?
- **Time Complexity:** How long does it take to find a solution?

- Space Complexity: How much memory is needed to perform the search? Memory conservation is not as important as it once was.

Time and space complexity are measured in terms of

- a) b: maximum branching factor of the search tree i.e. the number of children the tree has.
- b) d: depth of the least cost solution.
- c) m: maximum depth of the state space.

2.2 Informed Search

Informed search is an artificial intelligent search that uses information about the problem to guide the search, usually guesses the distance to a goal state, but the search may not be always possible. It uses problem-specific knowledge beyond the definition of the problem itself. This search tries to reduce the amount of search that must be done by making intelligent choices for the nodes that are selected for expansion. This implies the existence of some way of evaluating the likelihood that a given node is on the solution path. Informed search uses heuristic for its moves.

Heuristic is a function that ranks alternatives in various search algorithms at each branching step, based on the available information in order to make a decision about which branch to follow during a search. For example, for shortest path problems, an heuristic is a function, $h(n)$ defined on the nodes of a search tree, which serves as an estimate of the cost of the cheapest path from that node to the goal node. Heuristics are used by informed search algorithms to choose the best node to explore.

2.3 Uninformed Search

Uninformed search can also be referred to as Brute-force, blind, exhaustive or tree search. Brute force search is a general problem solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem statement. These search algorithms does not require any domain-specific knowledge i.e. it does not take into account the specific nature of the problem. All that is required of a brute-force search is a state description, a set of legal operators, an initial state, and a description of the goal state [5]. It uses no intelligence to determine the best path to the solution and therefore may not be very efficient. However, they typically perform a lot of comparisons before they find the solution.

Brute force search is simple to implement and will find a solution if it exists. However, its cost is proportional to the number of candidate solutions, which in many practical problems, tends to grow very quickly as the size of the problem increases. Therefore Brute force search is typically used when the problem size is limited, or when there are problem specific heuristics that can be used to reduce the set of candidate solutions to a manageable size. It is also used when the simplicity of implementation is more important than speed.

For this research, 3 types of uninformed search were studied. These are Breadth first search, Depth first search and Depth first iterative deepening search.

2.4 Breadth First Search

Breadth first search (BFS), searches breadth-wise in the problem space. It is like traversing a tree where each node is a state, which may be a potential candidate for solution [6]. BFS is a simple search strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded [7]. In this strategy, no viable solution is omitted and therefore guarantees that optimal solution is found but it is often not feasible when the search space is large.

BFS can be implemented using a list of unexpanded nodes and manages the list as a first-in first-out queue. BFS starts by expanding the root to see if the solution is generated. If not, the search continues to expand each child of the root until all of the children are expanded. After all of the children are expanded, the search expands each of the root's children. This continues until the solution is found or all nodes are searched. In other words, breadth first search expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of node. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue. The time and space complexity of BFS is $O(bd)$. The diagram for the Breadth First Search progress is shown on figure 1 below.

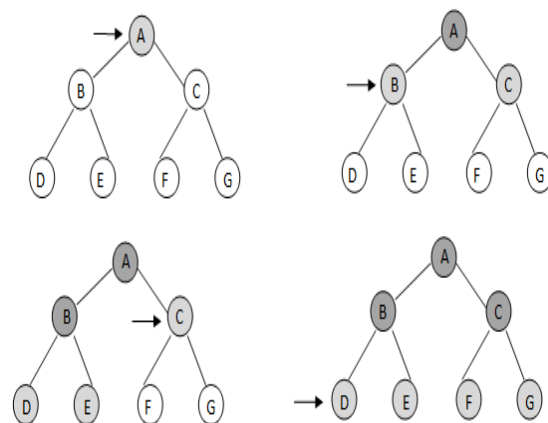


Figure 1 Breadth First Search progress

2.4.1 Breadth First Search Algorithm

1. Create a variable called LIST and set it to the initial state.
2. Loop until the goal state is found or LIST is empty.

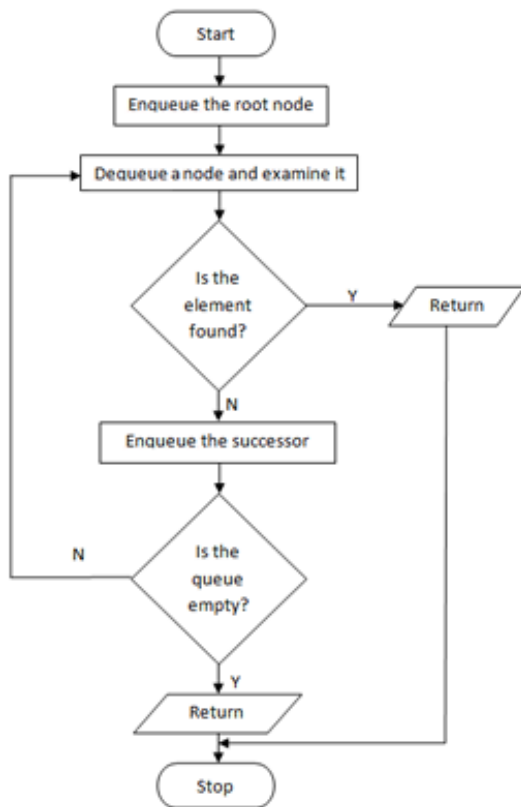


Figure 2 Breadth First Search Flowchart

- a. Remove the first element, say A, from the LIST. If LIST was empty then quit.
- b. For each way that each rule can match the state described in A do:
 - i. Apply the rule to generate a new state.
 - ii. If the new state is the goal state, quit and return this state.
 - iii. Otherwise add this state to the end of LIST

Breadth-first search can be implemented using queue as shown below:

1. Enqueue the root node.
 2. Dequeue a node and examine it.
 - a. If the element sought is found in this node, quit the search and return a result.
 - b. Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
 3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
 4. If the queue is not empty, repeat from Step 2.
- The flowchart is shown in Figure 2 above.

2.4.2 Properties of Breadth First Search

- Complete? Yes, If b is finite
- Optimal? Yes, only if all step cost are identical i.e if cost = 1 per step.
- Time complexity: $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d
- Space complexity: $O(b^{d+1})$ – Keeps every node in memory

2.5 Depth First Search

Depth first search (DFS) searches deeper into the problem space [8]. Depth first search uses last-in first-out stack for keeping the unexpanded nodes. It is a search strategy that extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path [3]. More commonly, depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack. DFS starts by expanding the root or the initial node and generating its successors to see if the solution is generated. In each subsequent step, DFS expands one of the most recently generated nodes. This continues until a leaf is encountered. A leaf is a node without any children. After a leaf is encountered, and no success exists, the search backtracks to the parent and explores an alternate child. This continues until all nodes are expanded or the solution is found. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

DFS remedies the space limitation of breadth-first search by always generating next a child of the deepest unexpanded node [5]. Both algorithms can be implemented using a list of unexpanded nodes; with the difference that breadth-first search manages the list as a first-in first-out queue. The time complexity of DFS is $O(bm)$ and the space complexity is $O(bm)$. Diagram shown on figure 3 below;

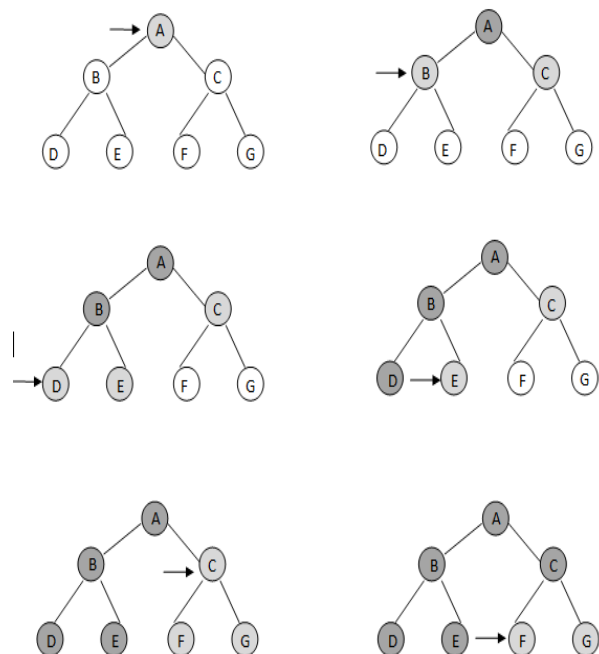


Figure 3 Depth First Search progress

2.5.1 Depth First Search Algorithm

1. If the initial state is a goal state, quit and return success.
2. Otherwise, loop until success or failure is signaled.

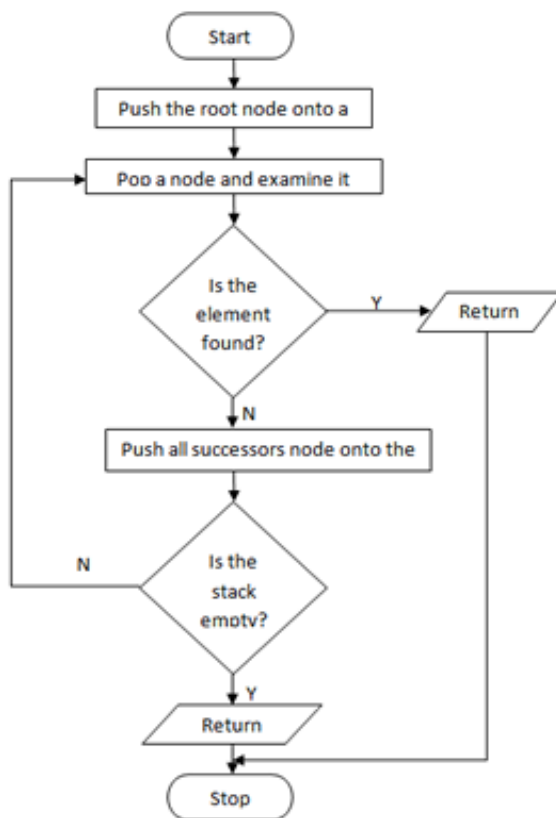


Figure 4 Depth First Search Flowchart

a. Generate a state, say E, and let it be the successor of the initial state. If there is no successor then signal failure.

b. Call Depth-First Search with E as the initial state.

c. If success is returned, signal success. Otherwise continue in this loop.

Depth-first search can be implemented using stack as shown below:

1. Push the root node onto a stack.
 2. Pop a node from the stack and examine it.
 3. If the element sought is found in this node, quit the search and return a result.
 4. Otherwise push all its successors (child nodes) that have not yet been discovered onto the stack.
 5. If the stack is empty, every node in the tree has been examined – quit the search and return "not found".
 6. If the stack is not empty, repeat from Step 2.
- The flowchart is shown in figure 4 above.

2.5.2 Properties of Depth First Search

- Complete? No: fails in infinite-depth spaces, spaces with loops
- Optimal? No
- Time complexity: $O(bm)$: terrible if m is much larger than d but if solutions are dense, may be much faster than breadth-first search
- Space complexity: $O(bm)$ i.e. linear space

2.6 Depth First Iterative Deepening Search

DFID search combines the best features of breadth-first search and depth-first search. This search first performs a depth-first search to depth one, then starts over, executing a complete depth-first search to depth two, and continues to run depth-first searches to successively greater depths, until a solution is found [5]. Since it never generates a node until all shallower nodes have been generated, the first solution found by Depth First Iterative Deepening search is guaranteed to be along a shortest path. Moreover, since at any given point it is executing a depth-first search, saving only a stack of nodes, and the algorithm terminates when it finds a solution at depth d, the space complexity of DFID is only $O(bd)$.

Although it appears that Depth First Iterative Deepening (DFID) search wastes a great deal of time in the iterations prior to the one that finds a solution, this extra work is usually insignificant. To see this, note that the number of nodes at depth d is b^d , and each of these nodes are generated once, during the final iteration. The number of nodes at depth d-1 is b^{d-1} , and each of these is generated twice, once during the final iteration, and once during the penultimate iteration. In general, the number of nodes generated by DFID is $b^d + 2b^{d-1} + 3b^{d-2} + \dots + db$. This is asymptotically $O(b^d)$ if b is greater than one, since for larger values of d the lower order terms become insignificant. In other words, most of the work goes into the final iteration, and the cost of the previous iterations is relatively small. Depth first iterative deepening is asymptotically optimal in terms of time and space among all brute-force shortest path algorithms on a tree [5].

2.6.1 Depth First Iterative Deepening Algorithm

1. $L = 1$
2. While no solution, do
 - a. Perform DFS from initial state S_0 with cutoff L
 - b. If goal is found,
 - i. Stop and return solution,
 - ii. Else, increment depth limit L
3. Endif

2.6.2 Properties of Depth First Iterative Deepening Search

- Complete? Yes, if the branching factor is finite and there is a solution at some finite depth.
- Optimal? it will find the shortest solution first.
- Time complexity: $b + (b+b^2) + \dots + (b+\dots+b^d) = O(b^d)$ (i.e. asymptotically the same as BFS or DFS to limited depth d in the worst case)
- Space complexity: $O(bd)$ where b is the branching factor and d is the depth of the shallowest solution. (Since it's like DFS run different times, with maximum depth limit d)

Theorem: Depth-first iterative-deepening is asymptotically optimal among uninformed searches in terms of time, space, and length of solution.

Proof 1 (Length of solution): Since DFID generates all nodes at a given depth before expanding any nodes at a greater depth, it always finds a shortest path to the goal. Hence, it is optimal in terms of solution length.

Proof 2 (Running time): The nodes at depth d are generated once during the final iteration of the search. The nodes at depth $d - 1$ are generated twice, once during the final iteration at depth d , and once during the penultimate iteration at depth $d - 1$. Similarly, the nodes at depth $d - 2$ are generated three times, during iterations at depths d , $d - 1$, and $d - 2$, etc. Thus the total number of nodes generated in a DFID search to depth d is

$$bd + 2bd - 1 + 3bd - 2 + \dots + db \leq bd(1 + 2b - 1 + 3b - 2 + \dots + db - d) - (2)$$

letting $x = 1/b$ yields

$$bd(1 + 2x + 3x^2 + \dots + dxd - 1)$$

Since $(1 - x)^{-2}$, or $(1 - 1/b)^{-2}$, is a constant that is independent of d , if $b > 1$, then the running time of DFID is $O(bd)$.

Proof 3 (Space): Since DFID at any point is engaged in a depth-first search, it need to only store a stack of nodes which represents the branch of the tree it is expanding. Since it finds a solution of optimal length, the maximum depth of this stack is d , and hence the maximum amount of space is $O(d)$ [5].

3. METHODOLOGY

All the various search algorithms were studied and analyzed in order to choose the best search for grid resource. The algorithms were implemented using C programming language. Octave was used to simulate the running time performance of all the algorithms. The best algorithm based on the octave analysis was adopted for grid resources.

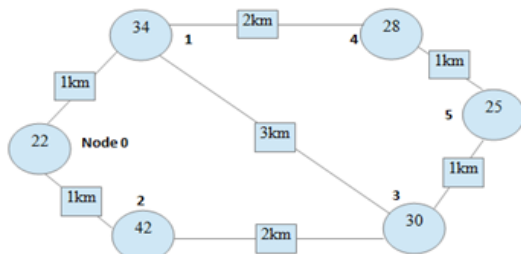


Figure 5 Simulated graph

4. RESULTS

4.1 Analysis of the algorithms

Table 1: Average time for the execution of the algorithms

Node Number	Depth First Iterative-Deepening Search (Seconds – System clocks)	Depth-First Search (Seconds – System clocks)	Breadth-First Search (Seconds – System clocks)
-1	0.025 - 25	0.030 - 30	0.055 - 55
0	0.055 - 55	0.055 - 55	0.060 - 60
1	0.030 - 30	0.055 - 55	0.050 - 50
2	0.060 - 60	0.055 - 55	0.030 - 30
3	0.060 - 60	0.055 - 55	0.025 - 25
4	0.055 - 55	0.060 - 60	0.055 - 55
5	0.055 - 55	0.055 - 55	0.055 - 55

-1	0.025 - 25	0.030 - 30	0.055 - 55
0	0.055 - 55	0.055 - 55	0.060 - 60
1	0.030 - 30	0.055 - 55	0.050 - 50
2	0.060 - 60	0.055 - 55	0.030 - 30
3	0.060 - 60	0.055 - 55	0.025 - 25
4	0.055 - 55	0.060 - 60	0.055 - 55
5	0.055 - 55	0.055 - 55	0.055 - 55

From the table above, it could be deduced that:

At node -1, the average running time of IDS is 0.025secs which is the shortest time

At node 0, IDS and DFS spent 0.055 while BFS spent 0.060

At node 1, IDS has d shortest time which is 0.030secs

At node 2, BFS has d shortest time which is 0.030secs

At node 3, BFS has d shortest time which is 0.025secs

At node 4, IDS and BFS have d shortest time which is 0.055secs

At node 5, they all have the same time, 0.055secs

Based on the output of the experiment carried out by this study, IDS has the shortest average running time BFS also has a short running time but since IDS has a linear storage capacity, it has an advantage over BFS. Therefore as said earlier, this study concludes that IDS is the best search strategy for grid resource.

4.1.1 Graph representation of the algorithms

The x-axis represents the number of nodes while the y-axis represents the time taken to reach each node otherwise known as time complexity. The graphs below were plotted using octave.

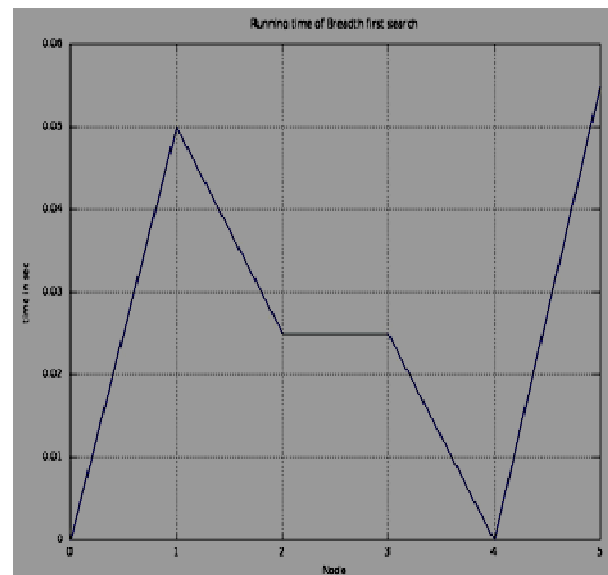


Figure 4 Breadth First Search running time

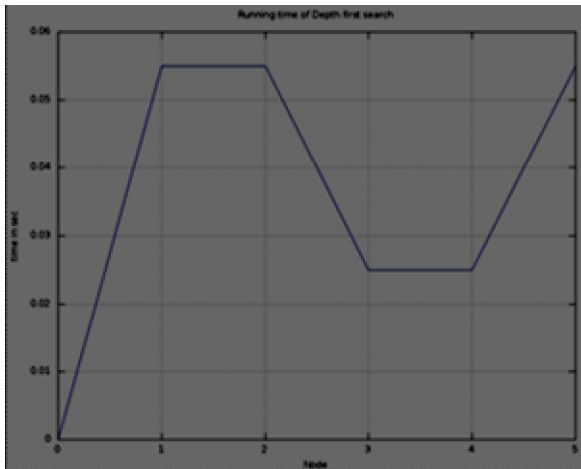


Figure 5 Depth First Search running time

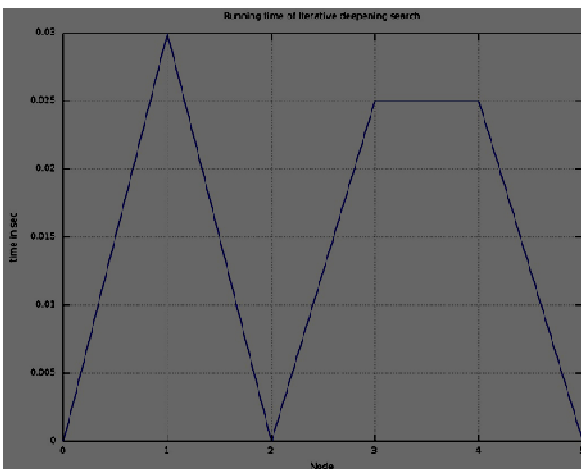


Figure 6 Depth First Iterative Deepening Search running time

5. CONCLUSION

The analysis done in this work shows that Breadth first search is complete, optimal based on some conditions but the time and space complexity is exponential, Depth first search space complexity is linear but it is neither complete nor optimal. Depth First Iterative Deepening Search is a good search strategy and is better than both Depth First search and Breadth First. This study concludes that most uninformed search algorithms have serious drawbacks; Breadth First Search uses too much space and Depth First Search uses too much time and is not guaranteed to find a shortest path to a solution. The Depth First Iterative Deepening search algorithm, however, is asymptotically optimal in terms of cost of solution, running time, and space required for uninformed searches. DFID combines both the best features of DFS and BFS and therefore, it is better than the DFS and BFS algorithms.

Therefore, DFID search is the best search strategy suitable for grid resources among the three search strategies studied in this work.

References

- [1] M. Jubran, "Search Algorithms" [Online]. 2011. Available: <http://www.elrona.com/02/16/search-algorithms/> [Accessed: Feb. 16, 2011].
- [2] D. M. Mount, "Design and Analysis of Computer Algorithms", Lecture Notes, 2004 CMSC 451, Department of Computer Science, University of Maryland, College Park, MD, 20742
- [3] R.C. Chakraborty, Problem Solving, Search & Control Strategies: AI Course Lecture 7-14, notes, slides. 2010
- [4] Z. Grunschlag, "Algorithms and Complexity" [Online]. 2002. Available: <http://www.coursehero.com/file/1661369/17/> [Accessed: Feb. 16, 2011].
- [5] R. E. Korf (1998), Artificial Intelligence search algorithms. Computer Science Department, University of California, Los Angeles, Los Angeles, Ca. 90095.
- [6] A. Robin, "Breadth First Search" [Online]. 2009a. Available: <http://intelligence.worldofcomputing.net/ai-search/breadth-first-search.html>. [Accessed: Dec. 16, 2009].
- [7] S. J. Russell, P. Norvig (2003). Artificial Intelligence: A Modern Approach (2nd ed.). Upper Saddle River, New Jersey: Prentice Hall.
- [8] A. Robin, "Depth First Search" [Online]. 2009b. Available: <http://intelligence.worldofcomputing.net/ai-search/depth-first-search.html>. [Accessed: Dec. 18, 2009].