International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 2 Issue 7, July - 2013

Applications of Depth First Search: A Survey

Gaurav Rathi, Dr. Shivani Goel
Thapar University, Patiala (Punjab)

Abstract

In this paper, various applications of depth first serach algorithms (DFS) are surveyed. The value of DFS or "Backtracking" as a technique for solving problem is illustrated by many applications such as cycle detection, strongly connected components, topological sort, find articulation point in a graph. The time complexity in different applications of DFS are also summarized.

Keywords

Depth first search, articulation point, strongly connected component, detecting cycle, graph, topological sort, railway rescheduling.

Introduction

There are many techniques for searching a graph. The DFS algorithm extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path. Given a graph G = (V, E) where V stand for set of vertices and E stands for set of edges. A vertex u ϵ V,where we want to explore each vertex in graph. Let n = |V| and m = |E|. Basically a graph can be of two types: directed and undirected. Graph can be represented by two techniques : 1) by matrix, 2) by linked list. Now we assume graph is represented by a linked list. The advantages of such representation are:
i)It requires $\Theta(n+m)$ space to store the vertices and there corresponding list, as opposed to $\Theta(n^2)$ for the adjacency matrix

ii)It makes it possible to go through the neighbours of a vertex u in O(|adj[u]|) time, linear in the number of neighbours.

Graphs form a suitable abstraction for problems in many areas like chemistry, electrical engineering, sociology and many more. Thus it is important to have the most economical algorithmsfor answering graph-theoretical questions.
DFS will process the vertices first deep and then wide. After processing a vertex, it recursively processes all of its descendants. Backtracking and depth first search is a technique which has been widely used to finding a solution of combinatorial theory and artificial intelligence [6].
Suppose G is a graph and we want to explore it. Initially all the vertices are unexplored than we start from a random vertex of graph G and now follow adjacent edge, traversing edge and visit new vertex we select single adjacent vertex and we continue in this way. At each step, we select an unexplored edge leading from a vertex already visited and we traverse this edge. The edge leads to some vertex, either new or already visited. Whenever we run out of edges leading from old vertices, we choose some unvisited vertex, if any exists, and begin a new exploration from this point. Eventually we will traverse all the edges of G, each exactly once[7].

Overall Strategy of DFS Algorithm

Depth-first search selects a source vertex *s* in the graph and paint it as "visited." Now the vertex *s* becomes our current vertex. Then, we traverse the graph by considering an arbitrary edge (*u*, *v*) from the current vertex *u*. If the edge (*u*, *v*) takes us to a painted vertex *v*, then we back down to the vertex *u*. On the other hand, if edge (u, v) takes us to an unpainted vertex, then we paint the vertex *v* and make it our current vertex, and repeat the above computation. Sooner or later, we will get to a "dead end," meaning all the edges from our current vertex *u* takes us to painted vertices. This is a deadlock. To get out of this, we back down along the edge that brought us here to vertex *u* and go back to a previously painted vertex *v*. We again make the vertex *v* our current vertex and start repeating the above computation for any edge that we missed earlier. If all of *v*'s edges take us to painted vertices, then we again back down to the vertex we came from to get to vertex *v*, and repeat the computation at that vertex. Thus, we continue to back down the path that we have traced so far until we find a vertex that has yet unexplored edges, at which point we take one such edge and continue the traversal. When the depth-first search has backtracked all the way back to the original source vertex, s, it has built a DFS tree of all vertices reachable from that source. If there still undiscovered vertices in the graph, then it selects one of them as the source for another DFS tree. The result is a forest of DFS-trees. Note that the edges lead to new vertices are called discovery or tree edges and the edges lead to already visited (painted) vertices are called back edges[9].

In order to keep track of progress, depth-first-search colours each vertex. Each vertex of the graph is in one of three states:

1. Undiscovered,
2. Discovered but not finished (not done exploring from it), and
3. Finished (have found everything reachable from it) i.e. fully explored.

The state of a vertex, *u*, is stored in a color variable as follows:

1. color[*u*] = White - for the "undiscovered" state,
2. color[*u*] = Grey - for the "discovered but not finished" state, and
3. color[*u*] = Black - for the "finished" state.

Like BFS, depth-first search uses $\pi[v]$ to record the parent of vertex v. We have $\pi[v]$ = NIL if and only if vertex v is the root of a depth-first tree.

DFS time-stamps each vertex when its color is changed.

1. When vertex v is changed from white to grey the time is recorded in d[v].
2. When vertex v is changed from grey to black the time is recorded in f[v].

The discovery and the finish times are unique integers, where for each vertex the finish time is always after the discovery time. That is, each time-stamp is anunique integer in the range of 1 to 2|V| and for each vertex v, d[v] < f[v]. In other words, the following inequalities hold:

Algorithm Depth-First Search

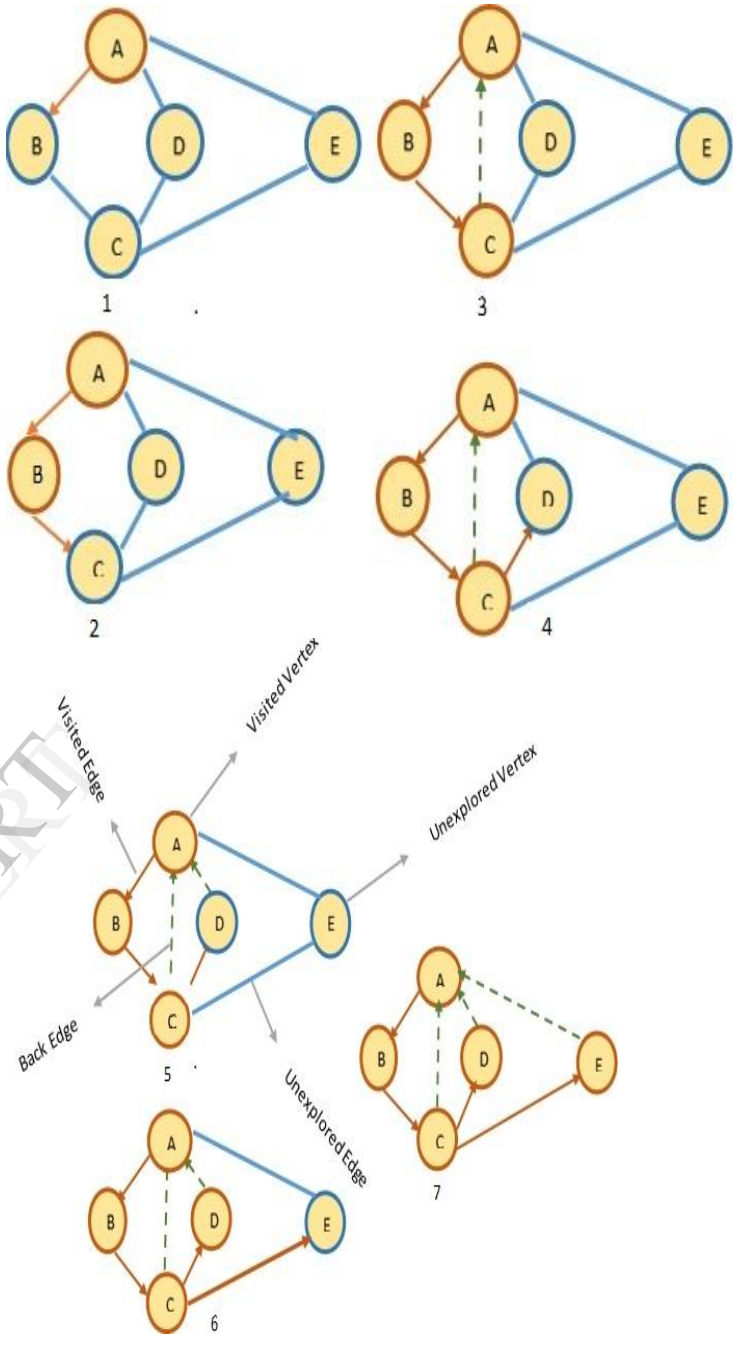
The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when edge (u, v) is explored. The following pseudo code for DFS uses a global timestamp time[8].

DFS (V, E)

```
1.  for each vertex u in V[G]
2.    do color[u] ← WHITE
3.       π[u] ← NIL
4.  time ← 0
5.  for each vertex u in V[G]
6.    do if color[u] ← WHITE
7.          then DFS-Visit(u)  ▷ build a new DFS-tree from u
```

DFS-Visit(u)

```
1.  color[u] ← GRAY              ▷ discover u
2.  time ← time + 1
3.  d[u] ← time
4.  for each vertex v adjacent to u   ▷ explore (u, v)
5.     do if color[v] ← WHITE
6.           then π[v] ← u
7.                DFS-Visit(v)
8.  color[u] ← BLACK
9.  time ← time + 1
10. f[u] ← time                  ▷ we are done with u
```



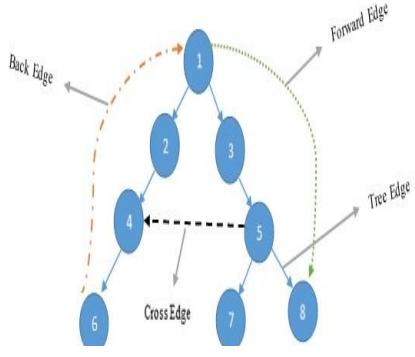
Applications of DFS

There are a large number of applications which use depth first search algorithm. Some of these are discussed here:

1)Detecting cycles

For this application first we should know about the types of edges in DFS algorithm. In DFS algorithm edges classify in four types .

- **Tree Edges** edge (*u; v*) is a tree edge if *v* was first discovered by exploring edge (*u; v*). The tree edges form a spanning forest of *G* (no cycles) .
- **Back Edge:** an edge (*u; v*) is a back edge if it connects vertex *u* to an ancestor *v* in a DFS tree.
- **Forward edge:** an edge (*u; v*) is a forward edge if it connects a vertex *u* to a descendant *v* in a DFS tree.
- **Cross edges:** all other edges.



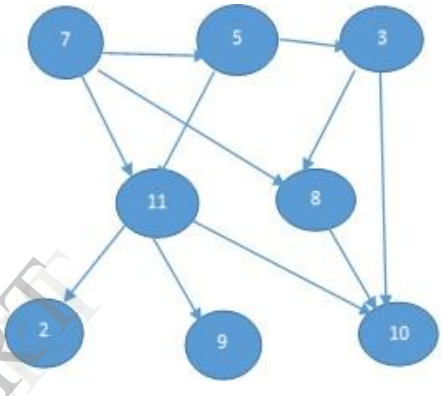
We already know that all edges of *G* will be classified as either tree edges or back edges. In addition, if *G* is acyclic, then only tree edges will occur. This is because if *G* is acyclic, then it is a forest (disconnected trees). Therefore, we have the following result: An undirected graph is acyclic if DFS yields no back edges. Detecting cycle can be done in O(n + m) time by running DFS. We use a stack s to keep track of the path between the start vertex and current vertex. As soon as a back edge is encountered, we return the cycle as the portion of the stack from the top to vertex w[7] .

Algorithm

```
1.   CycleDFS( G, v, z)
2.  SetLabel(v, VISITED)
3.  S.push(v)
4.for all e ϵ incidentEdges(v)
5.     ifgetLabel(e) = UNEXPLORED
6.        W ← opposite(v,e)
7.        S.push(e)
8.         IfgetLabel(w) = UNEXPLORED
9.              setLabel(e, DISCOVERY)
10.               pathDFS(G, w, z)
11.S.pop()
12.          else
13.            C ← new empty stack
14.repeat
15.o←S.pop()
16.C.push(o)
17.               until o = w
18.                returnc.elements()
19.  S.pop()
```

2)Topological Sorting

A **topological sort** (sometimes abbreviated **topsort** or **toposort**) or **topological ordering** of a directed graph is a linear ordering of its verticessuch that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another. In this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph(DAG). Any DAG has at least one topological ordering. The algorithms are known for constructing a topological ordering of any DAG in linear time[10].



Before we can carry out the tasks we have to arrange them in an order that respects all the dependencies. This is what we call a topological order of the dependency graph.
Let G = (V, E) be a directed graph. A topological order of G is a total order ‹ of the vertex set V such that for all edges (v ,w) ϵ E we have v ‹ w.
Topological order of the graph may be different or more than one. For example, in this graph is

7 ‹ 5‹ 3 ‹ 11 ‹ 8 ‹ 2 ‹ 9 ‹ 10
7 ‹ 5 ‹ 3 ‹ 8 ‹ 11 ‹ 10 ‹ 2 ‹ 9
7 ‹ 5 ‹ 3 ‹ 11 ‹ 8 ‹ 9 ‹ 10 ‹ 2 etc.

A directed graph has a topological order if, and only if, it is a DAG. Our algorithm is based on DFS. Let G be a directed graph. Consider the execution of dfs(G). We define an order ‹ of the vertices of G by saying that v ‹ w if w becomes black before v (i.e., vertices that finish later are smaller in the order).If we find, during the execution of dfsFromVertex(G , v) for some vertex v, an edge from v to a grey vertex w, then we know that G contains a cycle.

We can now modify our basic DFS algorithm in order to get an algorithm for computing the order ‹ and printing the vertices in this order. If the input graph G is not a DAG, our algorithm will simply print .G has a cycle. The algorithm adds all vertices to the front of a linked list when they become black. Thus vertices becoming black earlier appear later in the list, which means that the list is in order ‹ . If during the execution of sortFromVertex(G , v) for some vertex v, an edge from v to a

grey vertex w is found, then there must be a cycle, and the algorithm reports this and stops.

Algorithm

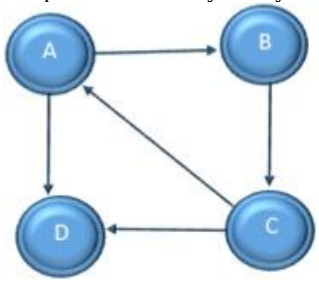
topSort(G)
1. Initialise array state by setting all entries to white.
2. Initialise linked list L
3. **for** all v ϵV do
4. **if**state[v] = white then
5. sortFromVertex(G , v)
6. print all vertices in L in the order in which they appear

sortFromVertex(G , v)
1. state[v] ←grey
2. for all w adjacent to v do
3. ifstate[w] = white then
4. sortFromVertex(G , w)
5. else if state[w] = grey then
6. print.Ghas a cycle.
7. halt
8. state[v] ←black
9. L.insertFirst(v)

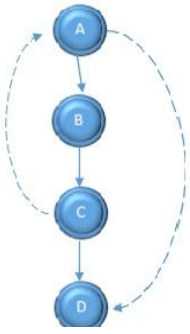
3)Strongly Connected Component

Decomposing a directed graph into its strongly connected components is a classic application of depth-first search. The problem of finding connected components is at the heart of many graph application. Generally speaking, the connected components of the graph correspond to different classes of objects. The first linear-time algorithm for strongly connected components is due to Tarjan (1972).

Given a digraph or directed graph G = (V, E), a strongly connected component (SCC) of G is a maximal set of vertices C subset of V, such that for all u, v in C, both $u \rightarrow v$ and $v \rightarrow u$, that is, both u and v are reachable from each other. In other words, two vertices of directed graph are in the same component if and only if they are reachable from each other.

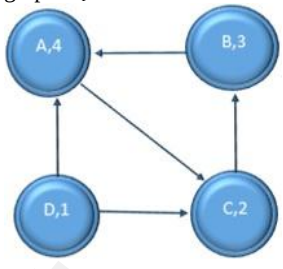


Now, as per the algorithm, perform DFS on this graph G produce below diagram. The dotted line From C to A indicates a Back edge



Now, performing post order traversal on this tree gives: D , C , B , and A.

Now reverse the given graph, G and call it as G_r and at the same time assign post order numbers to the vertices. The reverse graph G_r will look like:



Let step is, perform DFS on this reversed graph G_r. While doing DFS we need to consider the vertex which has largest DFS number. So, first start at A and with DFS we go to C and then B. At B, we cannot move further. This says that (A, B, C) is strongly connected component. Now the only remaining element is D and we end our second DFS at D itself. So the connected component are (A, B, C) and (D).

implementation based on this discussion can be given as:

Vertex	Post Order Number
A	4
B	3
C	2
D	1

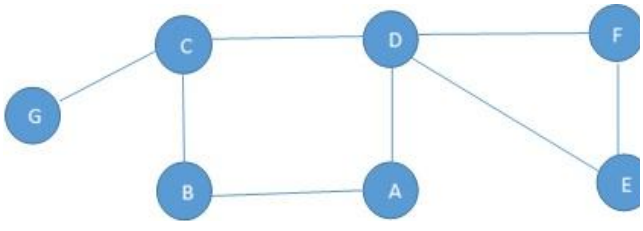
Algorithm

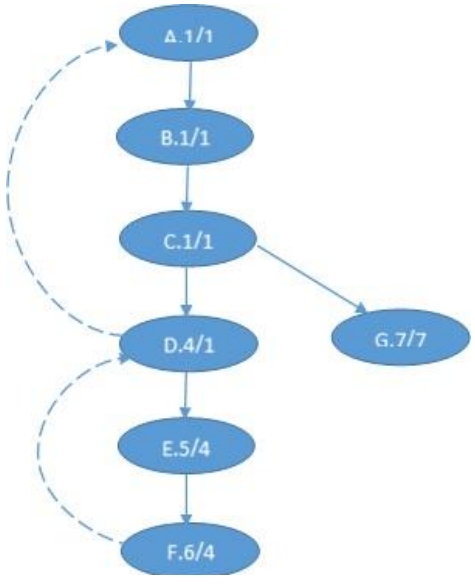
```
1.  intadjMatrix[256][256], table[256]
2.  Vector <int>st
3.  int counter = 0
4.  intdfsnum[256], num = 0, low[256]
5.  Void stronglyConnectedComponent(int u)
6.    low[u] = dfsnum[u] = num++
7.          Push(st, u);
8.    for(int v = 0 ; v<256 ; ++v)
9.       if(dfsnum[v] == -1)
10.         stronglyConnectedComponent(v)
11.      low[u] = min(low[u],low[v]);
12.   if(low[u]==dfsnum[u])
13.      while(table[u]=counter)
```

```
14.          table[st.back()]=counter
15.          Push(st)
16.       ++counter
```

4)Cut Vertex or Articulation Point

In an undirected graph, a cut vertex is a vertex and if we remove it, then the graph splits into two disconnected components. As an example, consider the following figure. Removal of "D" vertex divides the graph into two connected components {E , F}and {A , B , C , G }. Similarly, removal C vertex divides the graph into {G} and {A, B, C, D, E, F} . For this graph, A and C are the cut vertices.





Note: A connected, undirected graph is called bi-connected if the graph is still connected after removing any vertex.
DFS provides a linear-time algorithm (O(n)) to find all cut vertices in a connected graph. Starting at any vertex, call a DFS and number the nodes as they are visited. For each vertex, we call this DFS number dfsnum(v).

The tree generated with DFS traversal is called DFS spanning tree. Then, for every vertex v in the DFS spanning tree, we compute the lowest-numbered vertex, which we call low(v), that is reachable from v by taking zero or more tree edges and then possibly one back edge.
Based on the above discussion, we need the following information for this algorithm. The dfsnum of each vertex in the DFS tree (once it gets visited), and for each vertex v, the lowest depth of neighbours of all descendants of v in the DFS tree, called the low. The dfsnum can be computed during DFS. The low of v can be computed after visiting all descendants of v (i.e. just before v gets popped off the DFS stack) as the minimum of the dfsnum of all neighbours of v (other than the parents of v in the DFS tree) and the low of all children of v in the DFS tree.
The root vertex is a cut vertex if and only if it has at least two children. A non-root vertex u is a cut vertex if and only if there is a son v of u such that low (v) ≥ dfsnum(u). This property can be tested once the DFS return from every child of u (that means, just before u gets popped off the DFS stack), and if true, u separates the graph into differentbi-connected Component

This can be represented by computing one bi-connected component out of every such v (a component which contains v will contain the sub-tree of v, plus u), and then erasing the sub-tree of v from the tree.

Algorithm

For the above graph, the DFS tree with dfsnum / low can be given as show in above figure.

```
1.  adjMatrix [256][256]
2.  dfsnum[256], num = 0, low[256]
3.  void cutVertices(int u)
4.    low[u] = dfsnum[u] = num++
5.    For(int v = 0 ; v <  256 ; ++v)
6.       If(adjMatrix[u][v] &&dfsnum[v]= = -1)
7.          CutVertices(v)
8.          If(low[v] >dfsnum[u])
9.             Print  "u"
10.         low[u] = min (low[u] , low [v] )
11.      else     // (u,v) is a back edge
12.         low[u] = min(low[u], dfsnum[v])
```

5)Train Rescheduling During Traffic Disturbances

Railways are an important part of the infrastructure in most countries. The railway network consists of station and line sections, tracks, blocks, and events. Each station and line section can have one or more parallel tracks. All tracks are bidirectional, i.e. the track can be used for traffic in both directions depending on the schedule. A train uses exactly one track on a station or line section, but which specific track to use is often not predefined and therefore part of the re-scheduling problem. Each track is composed of one or several blocks connected serially and separated by signals. Each block can only be used by at most one train at a time due to the safety restriction imposed by line blocking[4]. A track with two

blocks can in theory hold two trains in the same direction, but not two trains in opposite direction due to the lack of a meeting point.

Each train has an individual, fixed route (i.e., the sequence of sections to occupy) which is represented as a sequence of train events to execute. A train event is when a certain train occupies a certain section. A train event has certain static properties such as minimum running time but also some dynamic properties, e.g., track allocation and start and end times on the section.

The railway traffic delay management and re-scheduling problem has been considered an important and difficult problem since quite some time[4]. It has been studied from different perspectives such as capacity, robustness, as well as passenger delay and dissatisfaction.

The capacitated delay management problem is a special case of the job shop scheduling (JSS) problem, where train trips are jobs which are scheduled on tracks that are considered as resources. A branch and bound (B&B) procedure is proposed for resource constrained project scheduling formulation by incorporating an exact lower bound rule and a beam search heuristic is used for tight upper bound.

In the train re-scheduling problem we have a disturbance in the railway traffic network forcing us to modify the predefined timetable in line with certain objective(s) and constraints. We have a set of n trains, $T = \{t_1, t_2, \ldots, t_n\}$ on a set of m sections, $S = \{s_1, s_2, \ldots, s_m\}$ where each section $s_j \epsilon$ {station, line} have a number of tracks $p \epsilon \{1, \ldots, p_j\}$. A station is called symmetric if the choice of track to occupy has no, or negligible, effect on the result.

Each train i has a set of events, K_i, and the set of all train events is denoted as $K = \{K_1, K_2, \ldots, K_n\}$ and its cardinality is: $C = P|T| \; i=1 \; |K_i|$. Each train event k has a predefined start time t start k and end time tend k in line with the timetable and which needs to be modified based on the minimum running time dk. It also belongs to a specific section $s_k \epsilon \{s_1, s_2, \ldots, s_m\}$. Each event is executed on exactly one track of its section.

The objective is to minimize the sum of the final delay suffered by each train at its final destination within the problem instance. The quality of the solution is thus given by this objective value, where a lower value indicates an improvement.

The main objective of the sequential greedy algorithm is to quickly find a feasible and good-enough solution, and therefore it performs a depth-first search (DFS)[5]. It uses an evaluation function to prioritize when conflicts arise and branches according to a set of criteria.

Notation used when describing the parallel DFS algorithm .

The search tree is built iteratively by selecting the earliest event of each train, collecting them into a candidate list and executing

Symbol	Definition
NC	C1,C2, ...,Cn where NC is the candidate list
PS	partial solution branch
T_0	the time when the disturbance occurs
ET_{Limit}	execution time limit (30 sec in our experiments)
C_i	candidate index to start with
T_c	total number of candidates
BV_w	branching value
GBV	global best value communicated via the white board
CV_w	cost estimation value of the current node
W	total number of workers
W	worker index

the best event in this list and adding it to the tree[4]. An event represents a train movement, i.e., a train running on a certain section with a start time, a minimum running time, a preferred track to occupy, and an end time. Each node in the search tree represents either an active event (i.e. a track has been allocated and a new start time has been set), or a terminated event (i.e., the train has left the assigned track and the corresponding event has been assigned an end time)[3]. In each node, an optimistic cost estimation is made of the solution which this branch at its best could generate.

processing, (ii)depth-first search, and (iii)solution improvement using backtracking and branching on potential nodes. In the pre-processing phase, all events which were active at the disturbance time T0 (see Figure 2) are executed by allocating a start time and a track. A candidate list which holds the next event of each train is created and sorted w.r.t the earliest starting time of the events.

In the second phase, feasible (i.e., deadlock free, without conflicts, etc.) candidate events are executed. The candidate list is updated accordingly with the next event of the train that just executed an event (if it has any events left to execute). This is repeated until a feasible solution is found, i.e., all events are executed.

The third phase starts as soon as the first feasible solution has been found and it aims to improve the best solution found so far by backtracking to a potential node. A potential node is a node that has an estimated cost that is lower than the currently best solution, and another branch from this node is then explored. The improvement process continues until the time limit is reached or a feasible solution with an objective value as low as the lower bound is found.

Our parallel algorithm is based on the sequential greedy algorithm. The B&B procedure is improved by sharing improved solutions among workers using a synchronized white board. We use a master-slave parallelization strategy. Initially, only the master is active and the workers (slaves) are waiting to get the initial unexplored subspaces. Using the notations in

Table, we outline the parallel algorithm starting with the master thread.

Let NC and PS be empty, and the disturbance occurs at time T0. As in the sequential algorithm, identify the events that are active at T0, execute them, and put them into PS. Populate the NC with the next event to execute ***of e***ach train[2], sorted w.r.t the earliest starting time, and compute the theoretical lower bound. Determine the values of $T_c$and W where $W = T_c$ in these experiments. A unique copy of the problem along with ET_{Limit}, C_i and PS are sent to each worker.
The outline of the worker threads is as follows:
Candidate selection: First execute the candidate C_i, determine the new NC and get a suitable candidate based on the depth-first search node selection rule.
Stopping criteria: If the bounds in term of execution time limit ET_{Limit} is exceeded or the lower bound is reached, then terminate and output the best result. If the candidate to execute, i.e., C_i, is not suitable then stop execution and return.
Read white board: Read the white board for availability of improved solutions found by any other worker; if available, then update BV_w in line with GBV.
B&B process: When $CV_w$is greater than or equal to BV_w, discard the node, backtrack and try other alternatives, and discard new branches from symmetric stations.
Feasible solution: If all of the train events are terminated and an improved solution is found, then update GBV on the white board with the improved solution. With an updated value of BV_w, backtrack and start branching from the node with a value less than BV_w.
Deadlock handling: In case of no track is available due to deadlock, backtrack and start branching where the wrong decision was made.
Results: After termination, send back the solution S(w) to master.

Conclusion

VLSI, web applications, network, and data mining are represented by using a massive graph. These applications need graph search, checking all nodes in a graph with the goal of finding a specific node with a given property. DFS is a technique which helps to search a specific node in a huge graph and with some modification we can convert the DFS algorithm in a memory efficient or time efficient algorithm for i.e. DFS_h , Iterative deepening , parallel DFS etc. we can use DFS in different Projects where graph is used i.e. Road Networks , SPIHT, Railways Re-scheduling etc.

The idea of systematically exploring a graph lets us learn the structure of a graph and thereby solve a wide variety of graph problems. One particular powerful variant of graph exploration is Depth first search.

References

[1]. Broder, A., R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata and A. Tomkins, J. Wiener, 'Graph Structure in the Web,' *Computer Networks*, Vol. 33, pp. 309–320, June 2000.

[2]. Aggarwal, A., R.J. Anderson, 'A Random NC Algorithm for Depth First Search,' *Cominatorica*, 8(1), pp. 1–12,1988.

[3]. F. Corman. Real-time Railway Traffic Management, Dispatching in complex, large and busy railway networks. Ph.D. thesis, TechnischeUniversiteitDelft,The Netherlands, December 2010. 90-5584-133-1.

[4]. A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. IEEE Trans. on Knowledge and Data Engineering, 11(1):28–35, 2002.

[5]. X. Zhou and M. Zhong. Single-track train timetabling with guaranteed optimality: Branch-and-bound algorithms with enhanced lower bounds. Transportation Research Part B: Methodological, 41(3):320 – 341, 2007.

[6]. International journal on Artificial Intelligence Tools, vol .6, no. 2 (1997) 255-271world scientific comapany

[7]. Jon Kleinberg and ÉvaTardos, *Algorithm Design*. Pearson Education, 2006.

[8]. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*.The MIT Press, September 2001.

[9]. M. Goodrich and R. Tamassia, *Algorithm Design*. John-Wiley and Sons. 2002.

[10]. CS2 Algorithm and data structure note 10 CS2BH 31 January 2005