Graph Search

Parallel Programming

October 4, 2016

Graphs

A graph is a finite set of nodes with edges between nodes.
Formally, a graph G is a structure (V,E) consisting of

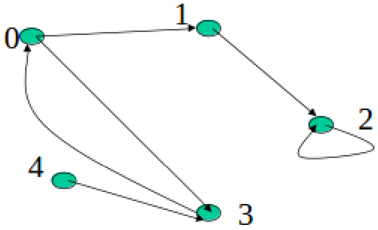
- a finite set V called the set of nodes, and
- a set E that is a subset of VxV. That is, E is a set of pairs of the form (x,y) where x and y are nodes in V

Adjacency Matrix Representation

- In this representation, each graph of n nodes is represented by an n x n matrix A, that is, a two-dimensional array A
- The nodes are (re)-labeled 1,2,,n
- A[i][j] $=$ 1 if (i,j) is an edge
- A[i][j] $=$ 0 if (i,j) is not an edge

Example of Adjacency Matrix Representation

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \end{bmatrix}$$



Graph Traversal

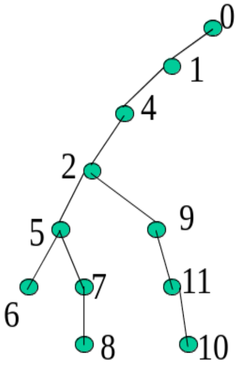
Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited.

- There are two standard graph traversal techniques:
  1. Depth-First Search (DFS)
  2. Breadth-First Search (BFS)

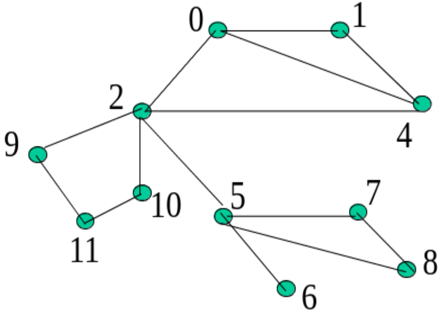
Depth-First Search

- DFS follows the following rules:
  1. Select an unvisited node x, visit it, and treat as the current node
  2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
  3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
  4. Repeat steps 3 and 4 until no more nodes can be visited.
  5. If there are still unvisited nodes, repeat from step 1.

Illustration of DFS



DFS Tree



Graph G

DFS Algorithm

Depth First Search algorithm(DFS) traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

It employs following rules.

1. Rule 1 - Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
2. Rule 2 - If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
3. Rule 3 - Repeat Rule 1 and Rule 2 until stack is empty.

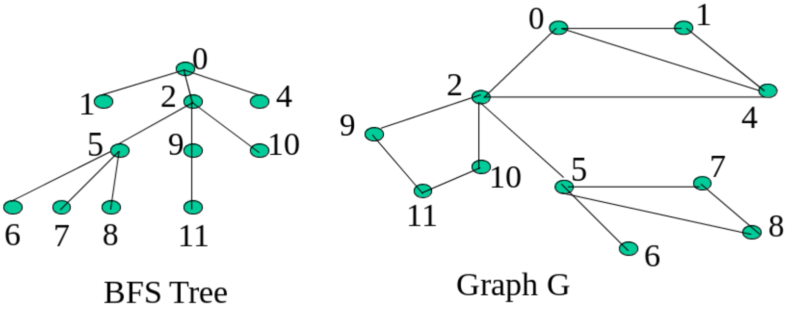
Pseudo-Code

```
DFS(s,n)
{
push(s);
vis[s]=1 //set it as visited
k=pop();
if(k not equal to 0)
print k;
while(k not equal to 0)
{
for(i=1 to n)
if(a[k][i]not equal to 0 and node is not visited)
{
push(i);
vis[i]=1;// set it as visited
}
k=pop();
if(k not equal to 0)
print k;
}
for(i=1 to n)
if(node is not visited)
dfs(i,n);
}
```

Breadth-First Search

- BFS follows the following rules:
  1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
  2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
  3. Repeat step 2 until no more nodes can be visited.
  4. If there are still unvisited nodes, repeat from Step 1.

Illustration of BFS



BFS Algorithm

Breadth First Search algorithm(BFS) traverses a graph in a breadthwards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.
It employs following rules.

1. Rule 1 Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
2. Rule 2 If no adjacent vertex found, remove the first vertex from queue.
3. Rule 3 Repeat Rule 1 and Rule 2 until queue is empty.

Pseudo-Code

```
BFS(s,n)
{
Set all nodes to "not visited";
q.enqueue(initial node);
vis[s]=1//set it as visited
p=q.dequeue();
if(p is not zero)
print p;
while(p is not zero)
{
for(i=1 to n)
if(a[p][i] is not equal to 0 and node is not visited)
{
q.enqueue(i) //node i
vis[i]=1 //set it as visited
}
p=q.dequeue();
if(p is not equal to 0)
print p;
}
for(i=1 to n)
if (i is not visited)
BFS(i,n)
}
```

Complexity

Algorithm	Time	Space
BFS	$O(\|V\| + \|E\|)$	$O(\|V\|)$
DFS	$O(\|V\| + \|E\|)$	$O(\|V\|)$

Table : Time and Space Complexity

Application Scalability : Scalability is the capability of the system to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth.

Vertices	Execution time		
	bfs	dfs	total
10	0.000011	0.00001	0.000021
20	0.000021	0.000019	0.00004
30	0.000033	0.000032	0.000065
40	0.000035	0.000033	0.000068
100	0.000248	0.000242	0.00049
500	0.002436	0.002379	0.004815
1000	0.009886	0.009677	0.019563

Table : Execution time for different graph size

Analysis : Scalability

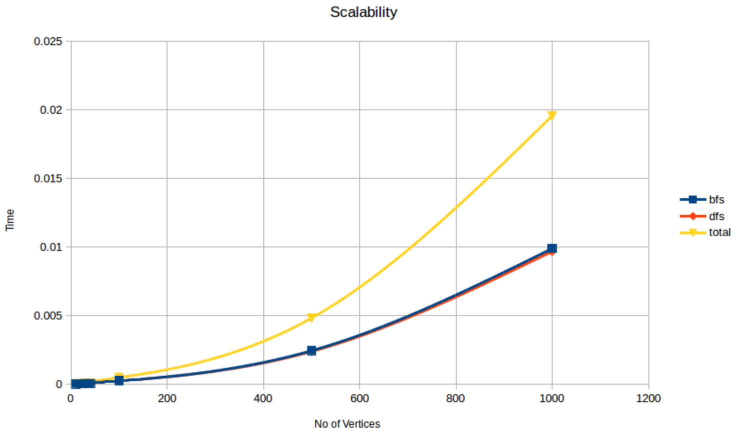


Figure : No. of Vertices Vs Time

CPU and Memory requirement :

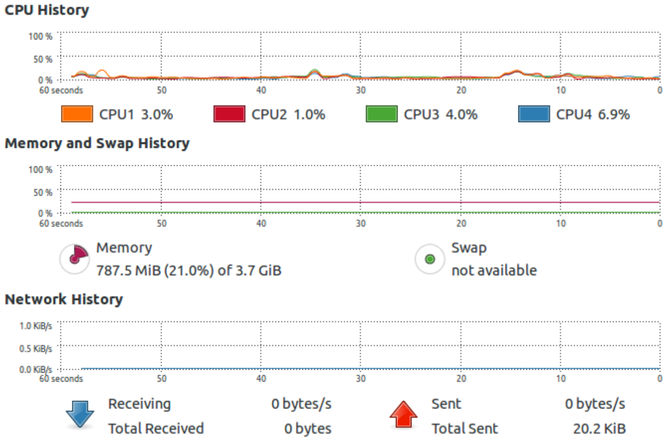


Figure : gnome-system-monitor

Analysis : Cache Misses

```
==5015==
==5015== I   refs:      749,520
==5015== I1  misses:      1,122
==5015== LLi misses:      1,062
==5015== I1  miss rate:    0.14%
==5015== LLi miss rate:    0.14%
==5015==
==5015== D   refs:      306,859  (203,007 rd   + 103,852 wr)
==5015== D1  misses:      1,961  (  1,340 rd   +     621 wr)
==5015== LLd misses:      1,704  (  1,134 rd   +     570 wr)
==5015== D1  miss rate:     0.6% (    0.6%     +     0.5%   )
==5015== LLd miss rate:     0.5% (    0.5%     +     0.5%   )
==5015==
==5015== LL refs:         3,083  (  2,462 rd   +     621 wr)
==5015== LL misses:       2,766  (  2,196 rd   +     570 wr)
==5015== LL miss rate:      0.2% (    0.2%     +     0.5%   )
```

Figure : Cache Misses

Analysis : Cache Misses

Vertices	D1			LL		
	Accesses	Misses		Accesses	Misses	
		Read	Write		Read	Write
10	82972	1300	546	2968	2171	496
20	168341	1314	568	3004	2180	518
30	306859	1340	621	3083	2196	570
40	498414	1386	698	3206	2218	643
100	2760300	2559	1308	4993	2231	1198
500	66608575	33636	16750	51509	2228	16582
1000	265688490	129721	64925	195773	62137	64734

Table : Cache Misses for different graph size

Analysis : Cache Misses

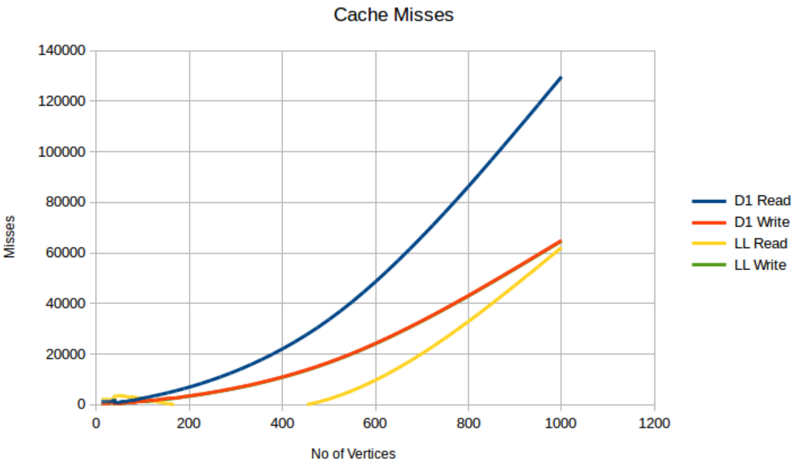


Figure : Cache Misses

Analysis : Memory Leaks

```
==5924== Memcheck, a memory error detector
==5924== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==5924== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==5924== Command: ./serial
==5924==
==5924==
==5924== HEAP SUMMARY:
==5924==     in use at exit: 0 bytes in 0 blocks
==5924==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==5924==
==5924== All heap blocks were freed -- no leaks are possible
==5924==
==5924== For counts of detected and suppressed errors, rerun with: -v
==5924== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure : Memory Leaks

Analysis : Function Costs

61.10	_IO_vfscanf	vfscanf.c
15.81	____strtol_l_internal	strtol_l.c
7.52	__isoc99_scanf	isoc99_scanf.c
4.30	_IO_sputbackc	genops.c
3.08	main	(unknown)
2.49	bfs	(unknown)
2.49	dfs	(unknown)

Instruction Fetch	60.91	60.91	Ir	
L1 Instr. Fetch Miss	6.95	6.95	I1mr	
LL Instr. Fetch Miss	6.42	6.42	ILmr	
Data Read Access	56.53	56.53	Dr	
L1 Data Read Miss	0.26	0.26	D1mr	
LL Data Read Miss	6.07	6.07	DLmr	
Data Write Access	65.88	65.88	Dw	
L1 Data Write Miss	95.61	95.61	D1mw	
LL Data Write Miss	96.49	96.49	DLmw	
L1 Miss Sum	31.41	31.41	L1m	= I1mr + D1mr + D1mw
Last-level Miss Sum	85.80	85.80	LLm	= ILmr + DLmr + DLmw
Cycle Estimation	61.10	61.10	CEst	= Ir + 10 L1m + 100 LLm

Thank You

Team Members

Rounak Jangir (201352028)
Anjul Tyagi (201351033)
Yash Choubey (201351006)