ADVANCED DATA STRUCTURES - COP 5536 - SPRING 2014

Dijkstra Algorithm using Fibonacci Heap

Name - Nitin Kosuri
Email - nikosuri@ufl.edu
UFID - 29694624

Implementation of Dijkstra algorithm in an unordered graph , using a simple data structure – Array, as well as Fibonacci heap , and measure the relative performance of the two implementations.

1. Associated Files

Dijkstra.cpp : This file has main function and it initiates the execution.

Usage:

Random Mode:

1. *g++ dijkstra.cpp*
2. *./a.out –r n d x*

where ‘n’ is the number of vertices, ‘d’ is the density of the graph and ‘x’ being the source vertex.

User-Input Mode:

I. *./a.out –s/f file-name*

where ‘–f’ denotes Fibonacci Mode, ‘-s’ denotes Simple Array Mode and file-name is the input file containing undirected graph representation. It has the number of vertices, edges and all combinations of vertices which form edges and their corresponding weights.

2. Compiler Description

The project has been compiled and tested under the following platform.

Platform/OS	**Compiler**
Linux (Ubuntu)	*g++*

Steps to execute the project in UNIX Environment using g++:

I. Run the command g++ dijkstra.cpp
II. Run ./a.out -r n d x for random mode.

3. Structure of the program using function prototypes

✓ *Fibonacci Heap functions*

a. **void fib_heap_insert(fibonacci_heap *fib_heap_obj, fibonacci_node *new_node, int key);**

Overview: This function inserts a new node into the fibonacci heap. Depending on the key mentioned, heap min pointer is changed appropriately and degree incremented by one.

Parameters: fib_heap_obj → Fibonacci heap pointer
New_node →new Fibonacci node pointer. Its values are initialized inside this function.
Key → key of the new node.

b. **Void fib_heap_existing_to_root(fibonacci_heap *fib_heap_obj, fibonacci_node *new_node);**

Overview: Add a node already existing in the heap into the root's doubly linked list and its pointers changed correspondingly. Mark value of the new node is set to false.

Parameters: fib_heap_obj → Fibonacci heap pointer
New_node → Pointer to the node which is truncated from its parent and added to the root list.

c. **fibonacci_heap *fib_heap_make()**

Overview: This function creates an empty heap and initializes its elements. It is used only when we are creating a new f-heap

d. **void fib_heap_add_child(fibonacci_node *parent_node, fibonacci_node *new_child_node)**

Overview: This function adds the new_child_node to the parent_node child list. Degree of the parent in incremented by one as well to reflect to the node.

Parameters: parent_node → parent_node pointer.
New_child_node → new child pointer which is added to the children linked list of the parent.

e. **void fib_heap_remove_node_from_root(fibonacci_node *node)**

Overview: This function is used to truncate a child node from a parent. Its parent's children linked list is updated to reflect the removed node.

Parameters: node → Node which has to be deleted from its siblings list.

f. **void fib_heap_link(fibonacci_heap *heap_inst, fibonacci_node *high_node, fibonacci_node *low_node)**

Overview: This function links two nodes as part of the consolidation operation. First the larger node is sliced from its sibling list and is then added as a child to the smaller node.

Parameters: heap_inst → the main heap pointer
High_node → node having the higher key which has to be linked to the lower key node
Low_node → node with the lower node

g. **void fib_heap_consolidate(fibonacci_heap *heap_inst)**

Overview: This method is the crucial method for fibonacci heaps where all the actual time is spent in pairwise merging of all the trees, to ensure at the end of the operation there are no two trees with the same degree. We scan the root list with the help of min pointer in the heap and combine trees with same degree into a single tree. We do this with the help of a auxillary table to see if there is a tree with degree already existing in our heap.

Parameters: heap_inst → Heap pointer

h. **fibonacci_node *fib_heap_extract_min(fibonacci_heap *heap_inst)**

Overview: This function extracts the minimum value from the heap based on the min_node pointer that each heap structure maintains. In order to verify the correctness of the fibonacci implementation with the array one, I am writing out each extracted min node to a file for making the comparision easier.

Parameters: heap_inst → the heap pointer

i. **void fib_heap_cut(fibonacci_heap *heap_inst, fibonacci_node *node, fibonacci_node *node_parent)**

Overview: This method removes the child node from its parent in the heap.

Parameters: heap_inst → heap pointer
Node →Pointer to the node that is being cut from its parent
Node_parent → parent of the truncated node

j. **void fib_heap_cascading_cut(fibonacci_heap *heap_inst, fibonacci_node *node)**

Overview: This recursive function removes the nodes from heap if the child mark values are true. It goes from child to parent until it sees a parent whose child mark value is false. It sets the child mark of last parent as true since it just lost a child.

Parameters: heap_inst → heap pointer
Node → pointer to the node on which recursive cut is being called

k. **void fib_heap_decrease_key(fibonacci_heap *heap_inst, fibonacci_node *node_inst, int new_key)**

Overview: This method sets the key field of node to amount specified as argument. Based on the new value it may be removed from the parent and called for cut and cascade methods.

Parameters: node_inst → pointer to the node whose key us being updated to 'new_key' value

✓ *Graph functions*

a. **graph* create_graph(int graph_size)**

Overview: This function creates a graph with the number of vertices specified by the user.

Parameters: graph_size → Number of vertices which is specified via user arguments.

b. **void add_edge(graph* graph_obj, int src, int dest, int edge_len)**

Overview: Adds an edge to an undirected graph.

Parameters: graph_obj →Pointer to the initialized graph
Src → Index of the source node
Dest → index of the destination node
Edge_len → weight of the edge between source and the destination

c. **void run_DFS(graph *my_graph, int node_index, bool *discovered)**

Overview: This recursive function runs the Depth-First-Search to find if the graph is completely connected.

Parameters: node_index → index of the node on which the DFS is being run.
Discovered → list of Boolean values that calculate if the node is connected to the graph.

d. **bool check_connected(graph *my_graph)**

Overview: This utility function checks if the graph is connected by running Depth-Frist-Search on the graph.

e. **graph* generate_random_graph(int vertices, int density)**

Overview: Method which generated a random graph with input parameters vertices and density. Total edges are calculated.

Parameters: vertices → total number of vertices as specified by the user

Density → Density of the edges

✓ *Dijkstra implementations*

a. **void dijkstra_normal(graph* graph_instance, int src)**

Overview: This function implements the Dijkstra algorithm using the routine array way for calculating the next minimum distant vertex from the source node.

Parameters: graph_instance → pointer to the created graph
Src → source vertex from which shortest paths have to be calculated

b. **void dijkstra_fibanocci(graph* graph_instance, int src)**

Overview: This function implements the Dijkstra algorithm using the fibonacci heap. Source vertex is specified by the user as input.

Parameters: graph_instance → pointer to the created graph
Src → source vertex from which shortest paths have to be calculated

c. **int get_min_distant_unmarked_node(graph* graph_obj, int *distance_to_dest, bool *marked)**

Overview: This function calculates the min distant unmarked node using the normal array implementation.

Parameters: distance_to_dest → list of distances of the nodes from the source node
Marked → list of marked Boolean values that denote if the minimum is reached for a particular node.

d. **int get_min_distant_unmarked_node_fib_heap(graph* graph_obj, fibonacci_heap *heap, fibonacci_node **node_array, bool *marked)**

Overview: This function calculates the minimum distant node using the fibonacci heap.

Parameters: node_array → list of node pointers to all the destination nodes
Marked → list of marked Boolean values that denote if the minimum is reached for a particular node.

4. Results

✓ *Expectations*

If we conduct the dijkstra algorithm using Fibonacci heap the expected complexity is **nlogn+e** and array implementation it is **n^2**. Fibonacci heap is better when compared to the Simple Scheme.

EXECUTION TIMES FOR SIMPLE AND FIBONACCI SCHEMES IN SECONDS

Vertices/Density	100	75	50	25	1
1000	1.03	0.78	0.52	0.28	0.1
	0.84	0.65	0.40	0.25	0.1
3000	10.63	7.85	5.22	2.26	0.62
	9.38	6.64	4.34	1.89	0.33
5000	35.19	24.64	16.31	6.47	0.9
	32	22.3	16.43	5.39	0.68

5. Conclusion

For small number of vertices the asymptotic behavior of fibanocci heap implementation is almost similar. For very large values of n the performance of fibanocci heap will be better asymptotically.