How effective are graph path searching and traversing algorithms compared to tree ones?

Word Count: 3603

Subject: Computer Science

May 2020 Session

Candidate Number:hrs196

Table of Contents

1. Introduction

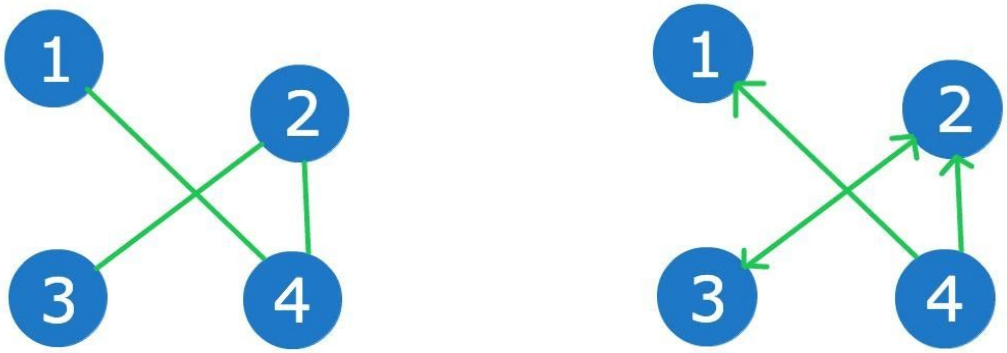
Searching is one of the most pivotal aspects of not only programming but also our daily lives. Hence, it must come as no surprise that the most popular website is a search engine. However, searching is an arduous task as sometimes there can be millions or even billions of pieces of data. To simplify and speed up the process of searching, various algorithms have been made. In this essay, we will explore two main categories of algorithms graphs and trees. Graphs are networks of nodes connected in with specific rules. Trees, on the other hand, arranges it's nodes in sorted order. We will compare Graph Traversal Algorithms, Greedy Best-First Search, A* search, Binary Search trees, and B Trees. Python will be used to run these algorithms, and the same samples of data will be given to all algorithms. The comparison will focus on memory usage and execution time.

2. Theory

2.1 Graphs

In computer science, Graphs are abstract data structures, meaning they are governed by a set of operations. Graphs are made up of vertices and edges. Vertices store a value of a certain type. Theoretically, there can be a finite or infinite number of vertices. Edges are the pathways between two vertices. If two vertices are connected by edges they are *adjacent*.

Graphs are divided into a myriad of categories, in this essay, however, we will only take a look at *directed* and *undirected* graphs as well as weighted ones. In undirected graphs, all edges are *bidirectional*[1], meaning that if A has an edge to B, then B is automatically connected with A. Directed graphs, however, edges have directions. So A can have an edge directed to B, but that doesn't necessarily mean that B has an edge towards B.



A **B**

Figure 2.1.1 ***A.****Example of an undirected graph.* ***B.*** *Example of a directed graph*

In the examples provided above in figure 2.1.1, we can see both kinds of graphs. We can show the graphs as a set in the following form:

$$V = \{1, 2, 3, 4\}$$

The vertex set will be the same for both graphs since they have the same vertices.

$$\varepsilon_A = \{(1, 4), (2, 3)(2, 4)\}$$

$$\varepsilon_B = \{(4, 1), (2, 3)(3, 2)(4, 2)\}$$

[1] Nykamp DQ, "Undirected graph definition." From *Math Insight*. http://mathinsight.org/definition/undirected_graph

As seen clearly, the set of edges is completely different. Since the “adjacency relation is symmetric” in undirected graphs, the number of elements in the set corresponds to the number of edges on the graph. In directed graphs, however, the order of pairs is important. If the set has (A, B), that means that the edge leads from A to B. That’s why in our set we have (2,3) and (3,2) since the edge between the nodes is bidirectional.

In an undirected graph, a *degree* is the number of edges leading to the graph. In a directed graph, in-degree is the number of edges leading into the node and outdegree is the number of edges leading out of a node.

The task of programming the graphs is a sufficiently easy one. Code below shows how we can represent the graphs above, using only one dictionary:

A

```
Graph = {
        1: 4,
        2: [3,4],
}
```

B

```
Graph = {
        1: 0,
        2: 3,
        3: 2,
        4: [1,2],
}
```

Figure 2.1.2 ***A.*** *Python code of an undirected graph.* ***B.*** *Python code of a directed graph*

In the dictionaries above, keys are the nodes of the graph and their values are the nodes they are connected to. In an undirected graph, we don’t need to specify all the nodes since the edges are bidirectional. Zero indicates that a node doesn’t have an outdegree.

However, in the real world, most graphs are weighted. The meaning of the weight changes depending on the context, it can mean cost, difficulty, or length of an edge. Below is an example of a weighted graph.

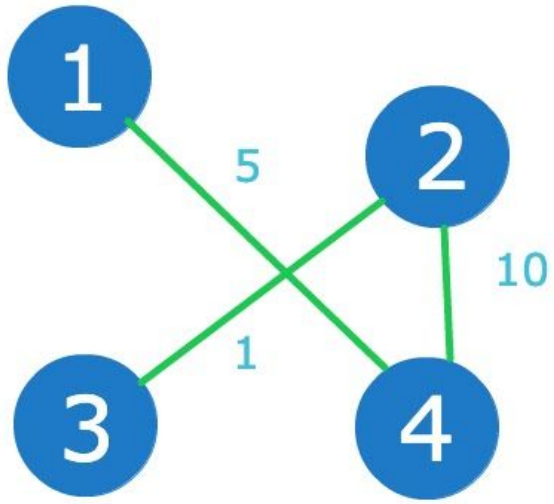


Figure 2.1.3 Example of a weighted graph

2.1.1 Graph Traversal Algorithms

Graph Traversal Algorithms are similar to linear search. The process is commenced from a source and it keeps searching until the desired node is found. There are two Graph Traversal Algorithms Depth-First Search(DFS) and Breadth-First Search(BFS).

DFS can be implemented in two ways - iterative and recursive. The recursive function relies on *backtracking* - moving backwards when there are no more nodes to check. The algorithm of a recursive DFS goes as follows:

1. Pick a source node to start from and add that node to a stack.
2. Move to the adjacent node and add that node to the stack.
3. If the node doesn't have any unchecked adjacent nodes, remove the node from the stack and go back.
4. Repeat until the target node is found.

The iterative function is very similar to the recursive one, except it relies on a while loop rather than recursion.

```
DFS-iterative (G, s):                                    //Where G is graph and s is source vertex
  let S be stack
  S.push( s )              //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
      //Pop a vertex from stack to visit next
      v  =  S.top( )
     S.pop( )
     //Push all the neighbours of v in stack that are not visited
    for all neighbours w of v in Graph G:
        if w is not visited :
                  S.push( w )
                 mark w as visited


DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

Figure 2.1.4 Pseudocode for DFS[2]

BFS, on the other hand, uses queues for storing the vertices. A queue follows the First-In-First-Out method when traversing the graph. The algorithm is as follows:

1. Select a source node and add the node to a queue.
2. Move to the adjacent nodes and add them to the queue and remove the source node from the queue.

[2] Depth First Search Tutorials & Notes: Algorithms. (n.d.). Retrieved from https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

3. Repeat the process until the queue is empty

```
BFS (G, s)                    //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

      mark s as visited.
      while ( Q is not empty)
            //Removing that vertex from queue,whose neighbour will be visited now
            v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                         Q.enqueue( w )              //Stores w in Q to further visit its neighbour
                         mark w as visited.
```

Figure 2.1.5 Pseudocode for BFS[3]

In both pseudocodes above, we can see that the nodes are always marked as "visited". This is done to ensure that the same node is not processed several times and that the function is not stuck in a loop if there is an edge that is connecting a node to itself.

However, unfortunately, BFS and DFS are simple algorithms and can only work with unweighted graphs. An altered version of BFS known as Djikstra's algorithm is used to find the shortest distance in a **weighted** graph. After we determine the source and the goal node, the steps are:

1. Set the distance to source node equal to 0 and to all other nodes equal to infinity.
2. Create a set of "visited nodes", initially the value is only the source vertex.
3. Create a queue of all the unvisited nodes, initially all nodes except the source.

[3]Breadth First Search Tutorials & Notes: Algorithms. (n.d.). Retrieved from https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

4. For each node, calculate the distance to its neighbours - distance to current node + distance to the neighbour node.
5. If the newly calculated distance is less then their previous value, replace it with the new one.
6. Repeat for all nodes, until the goal node is reached.

```
dist[s] ←o                                  (distance to source vertex is zero)
for all v ∈ V–{s}
     do dist[v] ←∞                          (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all vertices)
while Q ≠∅                                  (while the queue is not empty)
do  u ← mindistance(Q,dist)                 (select the element of Q with the min. distance)
    S←S∪{u}                                 (add u to list of visited vertices)
     for all v ∈ neighbors[u]
          do if  dist[v] > dist[u] + w(u, v)        (if new shortest path found)
                 then    d[v] ←d[u] + w(u, v)       (set new value of shortest path)
                                            (if desired, add traceback code)
return dist
```

Figure 2.1.6 Pseudocode for Dijkstra's algorithm[4]

2.1.2 Greedy Best-First Search

Best-First Search technique is heuristic, meaning, it sacrifices “optimality, accuracy, precision, or completeness for speed”[5]. Unlike the traversal algorithms, Best-First Search algorithms focus more on the weight of the edge and decide where to move based on which node is the most promising one. It uses an *f(n)* function to evaluate the

[4] Depth First Search Tutorials & Notes: Algorithms. (n.d.). Retrieved from https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/
[5] Vincent Kenny, Matthew Nathal, and Spencer Saldana (ChE 345 Spring 2014) https://optimization.mccormick.northwestern.edu/index.php/Heuristic_algorithms

adjacent nodes. The algorithm itself is pretty similar to BFS, but instead of a queue here we will use a special type of queues - a *priority queue(PQ)*. The difference between queues and priority queues is that a PQ has a priority associated with each value in the queue. In our case, values will be the vertices and the priority will be the weights of the edges. Usually PQs are implemented with **Insert(value,priority)**,**DeleteMax()** and **DeleteMin()** functions, which insert, delete the element with maximum and minimum priority respectively.

```
Best-First-Search( Maze m )
    Insert( m.StartNode )
    Until PriorityQueue is empty
        c <- PriorityQueue.DeleteMin
        If c is the goal
            Exit
        Else
            Foreach neighbor n of c
                If n "Unvisited"
                    Mark n "Visited"
                    Insert( n )
            Mark c "Examined"
End procedure
```

Figure 2.1.7 Pseudocode for Greedy Best-First Search[6]

2.1.3 A* Search

The final graph traversal algorithm we are going to analyze is the A* Search. To optimize speed and memory while being executed, A* star combines some aspects of

[6](n.d.). Retrieved from https://courses.cs.washington.edu/courses/cse326/03su/homework/hw3/bestfirstsearch.html

BFS and Greedy Best-First search algorithm. Just like the GBFS, A* is a heuristic algorithm and it uses *f(n)* function to evaluate the first node that should be explored. The difference between A* and greedy BFS is the *value*[7] of f(n) function. In A*, f(n)=g(n)+h(n), where:

- n is the previous node.
- g(n) is the cost of the path from the source node to the n.
- h(n) is a heuristic estimate of the cheapest cost from n to the target node.

While in the greedy BFS, *f(n)=h(n)*. There are several ways to estimate the h(n), but for the sake of this essay, we will have the same predefined estimates for all graphs. It should also be noticed that A* is a *complete* algorithm, meaning it will explore all the nodes, which might cause us a problem, but more on that later.

```
A* Search(s,g){  //s is the source node and g is the goal node
    let Visited be PriorityQueue
    let Unvisited be PriorityQueue
    Visited.push(s,0)
    while(Visited not empty){
        let q be node with least f(n) from Unvisited //f(n)=g(n)+h(n)
        remove q from Unvisited
        if q is the goal
            return q.f // the f(n) of the q
        else
            Foreach neighbour n of q
                let n.g  = q.g + distance between n and q
                let n.h  = heuristicEstimate() // we will give a simple value for the sake of the essay
                let n.f2 = n.g + n.h;
                if n is in Visited // if the node has already been explored
                   if n.f>n.f2  // if the new distance is lower than the old then change the old value to the new
                     n.f = n.f2;
                else
                   visited.push(n,n.f2)

    }
}
```

Figure 2.1.8 Pseudocode for A Search*

[7]What are the differences between A* and greedy best-first search? (2018, November 10). Retrieved from https://ai.stackexchange.com/questions/8902/what-are-the-differences-between-a-and-greedy-best-first-search

That was all the necessary knowledge of graphs. Now we will move on to a different type of data structure - Trees.

2.2 Trees

Just like graphs, trees are abstract data types. It's named a tree because it's hierarchical structure reminds us of a tree, though of an inverted one. The top node, the root of the tree, is one value that can be a *parent node* to two *children*. The node that has no children is the *leaf node*. The *height* of a tree is the longest path from the root to the leaf and *depth* of a node is its distance to the root. Below is a representation of a general tree(a tree without any constraints).

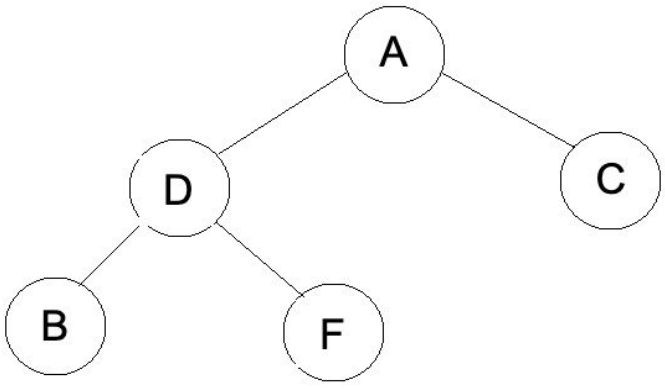


Figure 2.2.1 A General Tree

The nodes can have any values: numbers, letters or names. Defining general trees is pretty simple. The code below is for defining a tree structure in C++.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

Figure 2.2.2 Defining a general tree

Nevertheless, there are many more types of trees with distinctive properties and operations. In this essay, we will investigate Binary-Search Trees and B-Trees.

2.2.1 Binary Search Trees(BST)

The structure of binary search trees is just like the one of a general tree, the difference is in the values of nodes and its order. The features of BST are:

- Nodes have to have a numerical value.
- A parent can have at most 2 child nodes.
- Left-child is a node with a lesser value.
- Right-child is a node with a bigger value.

The fact that nodes are sorted in this order makes functions, such as searching, inserting and deleting, very swift. However, sometimes the trees can be *unbalanced*, the height of the left side is much bigger or smaller than the height of the right side, making the structure inefficient. This can decrease the efficiency of the tree from O(log n) to O(n).

Below we can see an example of a balanced Binary-Search Tree:

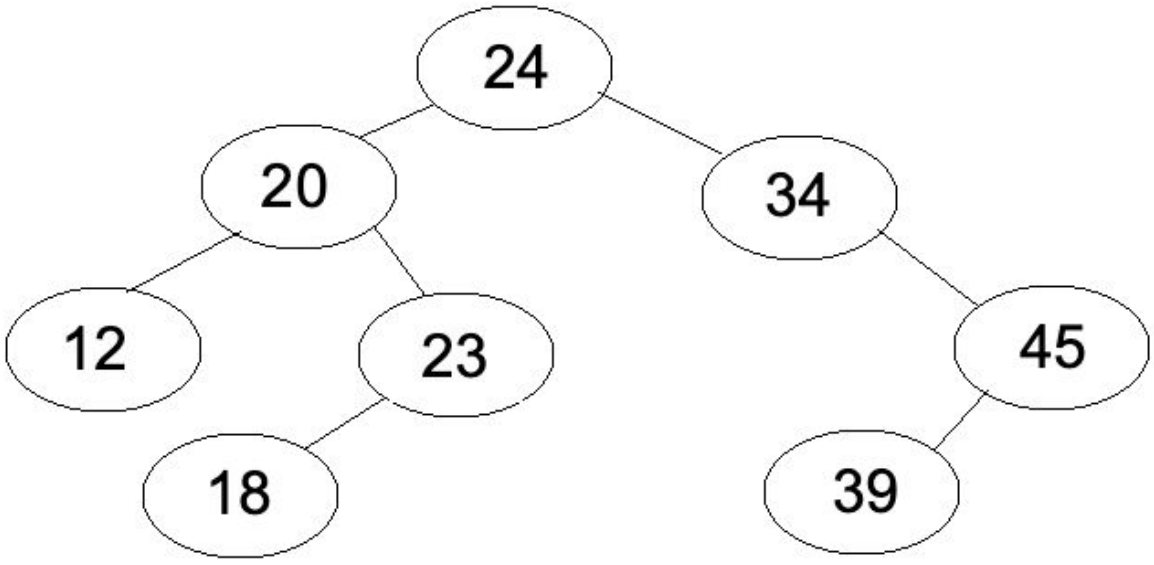


Figure 2.2.3 Example of a balanced BST

When it comes to traversing a BST, there are three techniques, all of which yield the same result, but in different orders:

A. *Preorder*

   1. The root
   2. The left subtree(recursively until the leaf node)
   3. The right subtree(recursively until the leaf node)

B. *Inorder*

   1. The left subtree(recursively until the leaf node)
   2. The root
   3. The right subtree(recursively until the leaf node)

C. *Postorder*

1. The left subtree(recursively until the leaf node)
2. The right subtree(recursively until the leaf node)
3. The root

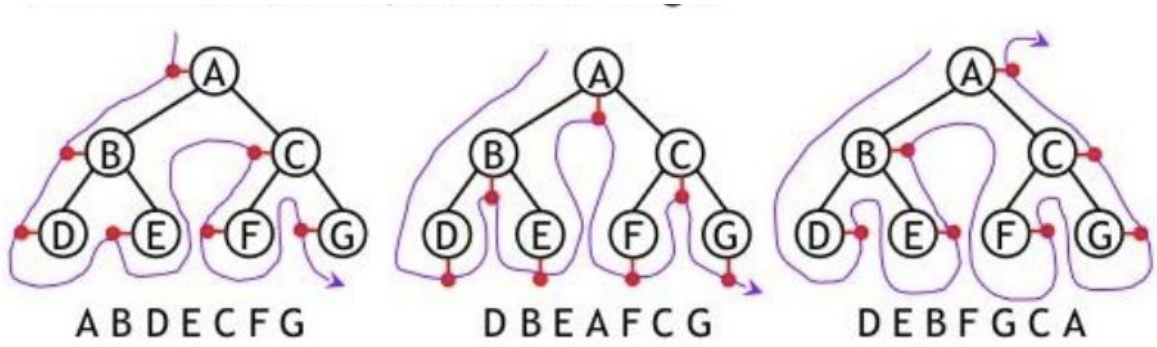


Figure 2.2.4 Visualization of Preorder, Inorder and Postorder[8]

Algorithm for searching for a value in the tree is relatively simple:

1. If the node is equal to the queried value, return the node
2. If the value of the node is bigger than the queried value, explore left
3. Otherwise, explore right.

[8] (n.d). Retrieved from https://ib.compscihub.net/wp-content/uploads/2018/07/5.1.16.pdf

```
BST-Search(x, k)
1: y ← x
2: while y ≠ nil do
3:     if key[y] = k then return y
4:     else if key[y] < k then y ← right[y]
5:     else y ← left[y]
6: return ("NOT FOUND")
```

Figure 2.2.5 Pseudocode for searching a BST[9]

A similar approach is used when finding the shortest distance between two nodes. To understand the steps, we first need to define the term *Lowest Common Ancestor(LCA)* of two nodes. The formal definition of LCA is "the shared ancestor of n1 and n2 that is located farthest from the root"[10]. Now, we can take a look at the steps for finding the shortest path between n1 and n2 in BST[11]:

1. Start from the root.
2. Move right if both n1 and n2 are greater than the current node.
3. Move left if both n1 and n2 are less than the current node.
4. If one is smaller and the other is bigger, the current node is an LCA and the distance is the sum of the distances from n1 to root and n2 to root.

[9] Retrieved from https://www.cs.rochester.edu/~gildea/csc282/slides/C12-bst.pdf
[10]Lowest Common Ancestor in a Binary Tree: Set 1. (2019, July 22). Retrieved from https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/
[11] Shortest distance between two nodes in BST. (2019, June 21). Retrieved from https://www.geeksforgeeks.org/shortest-distance-between-two-nodes-in-bst/

2.2.2 B-Trees

B-Trees are another type of tree data structures, that are very dissimilar to Binary Trees. B-Trees are self-balancing, meaning they arrange the nodes in a way to keep the height as small as possible. This makes the efficiency of the algorithm O(log n) no matter the number of nodes, while in BST, the efficiency can be as low as O(n) if the tree is unbalanced. That's why B-Trees are very useful when handling large databases. But just like the Binary Search Trees, the values of nodes in B-Trees, known as *keys*, can only be numerical.

B-Trees are defined by a *minimum degree* or *order* **k**. A node can have at most k-1 keys and k children. It's also vital that the root has at least 1 key and two child nodes. And every non-leaf node has to have at least $\lceil k/2 \rceil$ children. Finally, all leaves are on the same level.

There are many rules surrounding B-Trees relative to other data structures, but this all ensures maximum efficiency when inserting, deleting, traversing and searching the B-Trees.

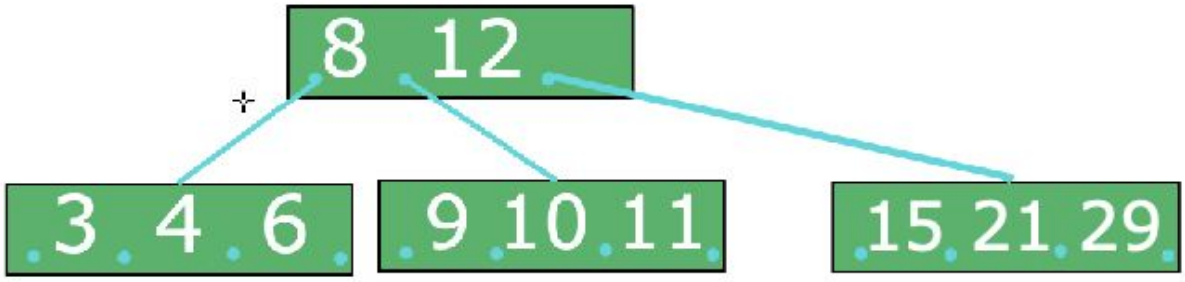


Figure 2.2.6 A simple B-Tree with an order of 4

The figure above is a simple example of a B-Tree of order 3. The root has only 2 keys, hence 3 children. The first child - 3,4,6 - are values that are less than 8. The second one - 9,10,11 - are the values between 8 and 12. And the last one is a compilation of keys more than 12.

The traversal algorithm for B-Trees is similar to the aforementioned Inorder algorithm for BST. We start from the leftmost leaf and return all the keys in the node. By repeating the function recursively, we reach the root and ultimately the rightmost child

```
Traverse(node){
    for i from 0 to n // n is the degree of a tree
        if node is not a leaf
            Traverse(node.children);
        print(node.keys);
}
```

Figure 2.2.7 Pseudocode for traversing a B-tree

The search algorithm for B-Trees is relatively simple. Just like the traversal algorithm, it's a recursive one. Start from the root, and go through all keys in all nodes until the desired value is found.

3. Practice

3.1 Procedure

In this part of the essay, we will finally implement all the algorithms we talked about in part 2. As mentioned earlier, the same graph maps will be used to test the algorithms, and the main focus will be on the execution time and memory used. To get more precise measurements, we will have several graph maps. The sizes of graphs will vary from several nodes to a few thousand and all graphs will be weighted and directed. This will give us a clearer picture of the effectiveness and the practicality of various algorithms. The values of the nodes in the graphs will then be imported into BST and B-Trees, to test the tree algorithms. To conclude, we will calculate the average execution time and memory of each algorithm, and compare them.

3.2 **Small Graphs**

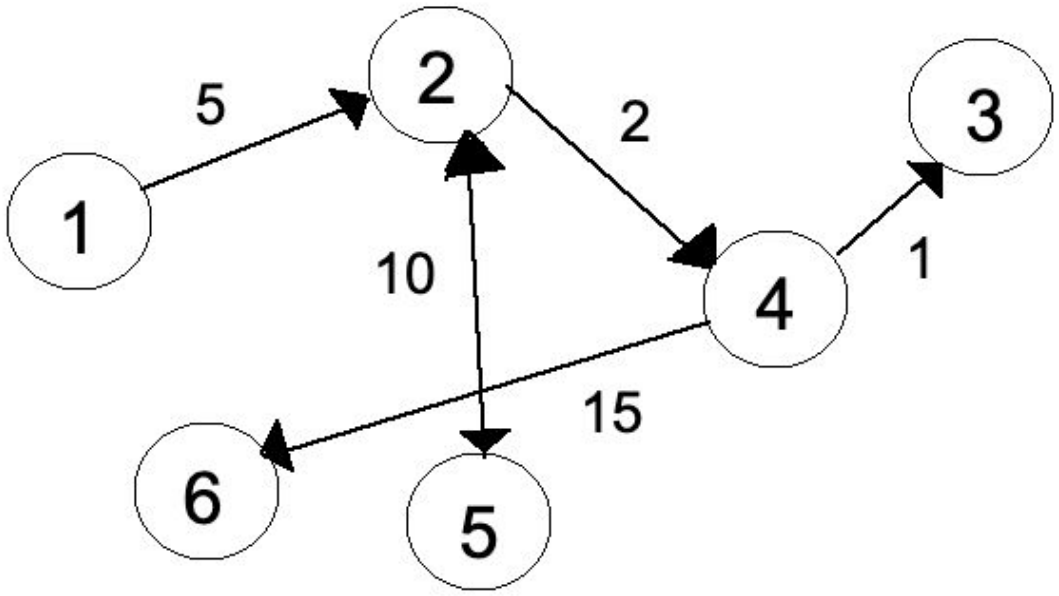


Figure 3.2.1 First graph map

The first graph to be analyzed is shown above. As mentioned, it's a directed graph with the respective weights of paths next to them. In Python, the graph will be defined as:

```
Graph = {
    1: [2],
    2: [4,5],
    3: [],
    4: [3,6],
    5: [2]
    6: [0];
}
```

Figure 3.2.1 Defining a graph in Python

The second graph will be a slightly larger one, with 20 nodes. In total, there will be graphs with 6, 20, 50 and 100 nodes in the "small section". For each algorithm, we will run the trial 2 times to get more precise results. The results for small graphs are below.

Test number	Algorithm	Time(s)	Memor(MB)
1	DFS	$1.365*10^{-7}$	5.131
2	DFS	$1.051*10^{-7}$	5.131
3	BFS	$5.319*10^{-5}$	5.332
4	BFS	$4.502*10^{-5}$	5.332
5	BST	$5.853*10^{-7}$	5.130
6	BST	$8.722*10^{-7}$	5.130
7	B-Tree	$2.532*10^{-6}$	5.132
8	B-Tree	$3.536*10^{-6}$	5.132

Test number	Algorithm	Time(s)	Memory(MB)
1	BFS	$4.538*10^{-5}$	5.331
2	BFS	$1.578*10^{-4}$	5.331
3	DFS	$2.521*10^{-7}$	5.130
4	DFS	$2.602*10^{-7}$	5.130
5	BST	$1.373*10^{-6}$	5.131
6	BST	$1.257*10^{-6}$	5.131
7	B-Tree	$1.670*10^{-6}$	5.132
8	B-Tree	$1.708*10^{-6}$	5.132

Test number	Algorithm	Time(s)	Memory(MB)
1	BFS	$1.802*10^{-4}$	5.331
2	BFS	$1.281*10^{-4}$	5.331
3	DFS	$5.498*10^{-7}$	5.130
4	DFS	$5.501*10^{-7}$	5.130
5	BST	$6.092*10^{-6}$	5.131
6	BST	$6.081*10^{-6}$	5.131
7	B-Tree	$8.086*10^{-6}$	5.132
8	B-Tree	$6.181*10^{-6}$	5.132

Test number	Algorithm	Time(s)	Memory(MB)
1	BFS	$4.432*10^{-4}$	5.332
2	BFS	$4.654*10^{-4}$	5.332
3	DFS	$1.479*10^{-7}$	5.132
4	DFS	$1.475*10^{-7}$	5.132
5	BST	$3.032*10^{-5}$	5.132
6	BST	$3.001*10^{-5}$	5.132
7	B-Tree	$5.585*10^{-5}$	5.132
8	B-Tree	$5.473*10^{-6}$	5.132

Figure 3.2.2 Tables of results for traversing graphs with 6, 20, 50 and 100 nodes, respectively

Test number	Algorithm	Time(s)	Memor(MB)
1	Djikstra	$7.226*10^{-7}$	5.131
2	Djikstra	$9.672*10^{-7}$	5.131
3	Greedy BFS	$6.052*10^{-7}$	5.431
4	Greedy BFS	$6.429*10^{-7}$	5.431
5	A*	$1.283*10^{-6}$	5.134
6	A*	$1.386*10^{-6}$	5.134
7	BST	$8.575*10^{-7}$	5.132
8	BST	$7.233*10^{-7}$	5.132
9	B-Trees	$1.536*10^{-6}$	5.132
10	B-Trees	$1.912*10^{-6}$	5.132

Test number	Algorithm	Time(s)	Memor(MB)
1	Djikstra	$1.019*10^{-6}$	5.131
2	Djikstra	$1.049*10^{-6}$	5.131
3	Greedy BFS	$5.432*10^{-7}$	5.431
4	Greedy BFS	$5.319*10^{-7}$	5.431
5	A*	$1.253*10^{-6}$	5.134
6	A*	$1.398*10^{-6}$	5.134
7	BST	$1.700*10^{-7}$	5.131
8	BST	$1.444*10^{-7}$	5.131
9	B-Trees	$1.820*10^{-6}$	5.132
10	B-Trees	$1.782*10^{-6}$	5.132

Test number	Algorithm	Time(s)	Memor(MB)
1	Djikstra	$1.614*10^{-6}$	5.131
2	Djikstra	$1.852*10^{-6}$	5.131
3	Greedy BFS	$3.332*10^{-6}$	5.431
4	Greedy BFS	$2.548*10^{-6}$	5.431
5	A*	$2.825*10^{-6}$	5.134
6	A*	$2.928*10^{-6}$	5.134
7	BST	$7.531*10^{-6}$	5.131
8	BST	$5.823*10^{-6}$	5.131
9	B-Trees	$3.240*10^{-6}$	5.132
10	B-Trees	$3.587*10^{-6}$	5.132

Test number	Algorithm	Time(s)	Memor(MB)
1	Djikstra	$2.978*10^{-6}$	5.133
2	Djikstra	$3.190*10^{-6}$	5.133
3	Greedy BFS	$1.325*10^{-6}$	5.433
4	Greedy BFS	$2.248*10^{-6}$	5.433
5	A*	$1.399*10^{-6}$	5.134
6	A*	$1.531*10^{-6}$	5.134
7	BST	$2.353*10^{-5}$	5.131
8	BST	$2.305*10^{-5}$	5.131
9	B-Trees	$2.446*10^{-5}$	5.132
10	B-Trees	$2.484*10^{-5}$	5.132

Figure 3.2.3 Tables of results for path-searching graphs with 6, 20, 50 and 100 nodes, respectively

3.3 Larger Graph

In the real world, it's more common to come across graphs with several thousand, if not millions, of nodes. An example of this would be Google Maps, where roads are edges and intersections are vertices. Implementing the abovementioned algorithms on larger graphs will help us get a clearer picture of the efficiency of each algorithm and a

more vivid view of the correlation between the number of nodes and the execution time as well as the memory consumption.

In this section, we will implement a graph map of 1000,5000 and finally 10000 nodes.

Trial Number	Algorithm	Time(s)	Memory(MB)
1	BFS	2.553E-04	5.399
2	BFS	2.423E-04	5.399
3	DFS	1.220E-03	5.203
4	DFS	1.244E-03	5.203
5	BST	3.211E-03	5.131
6	BST	3.091E-03	5.131
7	B-Tree	5.650E-05	5.132
8	B-Tree	5.962E-05	5.132

Trial Number	Algorithm	Time(s)	Memory(MB)
1	BFS	2.598E-04	5.670
2	BFS	2.482E-04	5.670
3	DFS	1.360E-03	5.470
4	DFS	1.384E-03	5.470
5	BST	9.390E-02	5.132
6	BST	1.075E-01	5.132
7	B-Tree	3.490E-04	5.133
8	B-Tree	3.580E-04	5.133

Trial Number	Algorithm	Time(s)	Memory(MB)
1	BFS	1.181E-03	7.028
2	BFS	1.033E-03	7.028
3	DFS	0.32	6.827
4	DFS	0.31	6.827
5	BST	0.39	5.132
6	BST	0.40	5.132
7	B-Tree	5.764E-04	5.132
8	B-Tree	5.769E-04	5.132

Figure 3.3.1 Tables of results for traversing graphs with 1000,5000 and 10000 nodes, respectively

Trial Number	Algorithm	Time(s)	Memory(MB)
1	Djikstra	9.609E-05	5.244
2	Djikstra	1.047E-04	5.244
3	Greedy BFS	2.348E-05	5.244
4	Greedy BFS	5.372E-05	5.244
5	A*	2.515E-03	5.265
6	A*	2.580E-03	5.265
7	BST	3.100E-03	5.131
8	BST	3.091E-03	5.131
9	B-Trees	5.332E-05	5.132
10	B-Trees	5.327E-05	5.132

Trial Number	Algorithm	Time(s)	Memory(MB)
1	Djikstra	1.297E-04	5.698
2	Djikstra	1.260E-04	5.698
3	Greedy BFS	4.350E-05	5.698
4	Greedy BFS	1.130E-05	5.698
5	A*	2.541E-03	5.790
6	A*	2.549E-03	5.790
7	BST	8.970E-02	5.131
8	BST	9.653E-02	5.131
9	B-Trees	3.406E-04	5.132
10	B-Trees	3.829E-04	5.132

Trial Number	Algorithm	Time(s)	Memory(MB)
1	Djikstra	2.178E-03	7.755
2	Djikstra	2.287E-03	7.755
3	Greedy BFS	6.390E-04	7.754
4	Greedy BFS	3.430E-04	7.754
5	A*	0.44	8.239
6	A*	0.36	8.239
7	BST	0.38	5.132
8	BST	0.39	5.132
9	B-Trees	5.75E-04	5.133
10	B-Trees	5.62E-04	5.133

Figure 3.3.2 Tables of results for path-searching with 1000, 5000, and 10000 respectively

4. Analysis of results

4.1 Small Graphs

Starting from the traversing algorithms, we have only Depth-First Search, Breadth-First Search, Binary Search Tree and B-Trees, since other algorithms are not used for traversing. Out of the four algorithms, DFS has proven to be the fastest with all trials having an order of magnitude of -7. BFS, on the other hand, was asserted as the slowest one with the fastest result being $4.502*10^{-5}$. BST showed the strongest correlation between the number of nodes and the time of execution, with 0.7 microseconds taken to traverse a 6 node graph and around 30 microseconds for a 100 node graph. As for the B-Trees, the results ranged from 3 microseconds to 24, also showing a standard correlation. The difference in memory used by each algorithm was negligible. In some cases, I was surprised to see that even when I added more nodes to the graph, the execution time stayed almost the same or the difference was negligible. One explanation for this is that the difference of 90 nodes for a computer is not that much to make significant changes in the execution time.

When coming to path-searching algorithms, a different set of algorithms was used - Djikstra's, Greedy BFS, A*, BST and B-Trees. Graph algorithms - Djikstra, GBFS and A*- were faster in this case and showed a weaker link between the number of nodes and the time. Djikstra had the fastest result of around 0.7 microseconds and the slowest one of 3 microseconds. For A*, the order of magnitude stayed the same - -6. Tree algorithms, contrarily, were slower and showed a strong relationship between the number of nodes and the execution time. BST had the best result of 0.72 microseconds

in the 6-node-graph and the worst result of 23 microseconds in the 100-node-graph. The difference in memory was in several kilobytes, which is negligible.

4.2 Larger Graphs

As more nodes were added to the graph, a significant change was observed in both execution time and memory usage. With larger graphs, B-Trees was proven to be the fastest even with 10000 nodes, it took the B-Trees algorithm 376 microseconds to fully traverse the graphs. Furthermore, BFS performed the worst when it comes to memory usage as it used 7MB compared to 6.8 of DFS and 5.1 of BST and B-Trees. It should also be mentioned that BFS algorithms showed a strong link between the number of nodes and memory complexity, while the tree algorithms didn't show any throughout the whole procedure. In this section, the fastest result was obtained by B-Trees 56.5 microseconds and the slowest result belongs to DFS - 0.31 seconds(310000 microseconds).

In regards to the traversing larger graphs, a similarly large difference between execution time in each subsequent increase in the number of nodes was remarked. In this case, Greedy BFS and B-Trees have been proven to be equally effective time-wise, however, B-Trees has an upper hand memory-wise. Yet again, we witness how graph algorithms face a significant increase in memory complexity, while the memory usage of tree algorithms stays the same. The worst performing algorithms were A*, 2500 microseconds at 1000 nodes and 0.40 seconds(400000 microseconds) at 10000, and Binary-Search Tree, 3100 and 380000 microseconds at 1000 and 10000 nodes respectively.

5. Conclusion

The purpose of this essay is to compare different traversing and path-searching algorithms, Graph Traversal Algorithms, Greedy Best-First Search, A* search, Binary Search trees, and B Trees, based on the time it takes for the program to reach its goal. Tables and pseudocodes for better visualization were also provided.

Some of the results were unexpected, such as BFS being faster than DFS when the number of nodes grew, or B-Trees being the fastest algorithm for traversing larger graphs.

We also found out that while memory complexity difference is negligible for smaller graphs, the difference increases exponentially with the number of nodes for graph algorithms, but stays the same for tree ones.

Even though most abovementioned algorithms follow a somewhat similar pattern and logic, there were several reasons that might have caused the difference in execution time - some algorithms being recursive, different methods of inserting nodes and importing libraries in some cases. When implementing BST and B-Trees for larger graphs, a separate function had to be implemented just to increase the recursion limit.

All algorithms discussed in this essay are used in a myriad of industries. Each has its own strength and weakness that discussed them above in section 4. It's important to analyze all the features of an algorithm before electing it. I hope my essay was useful in projecting the advantages and drawbacks of some algorithms.

Bibliography

Nykamp DQ, “Undirected graph definition.” From *Math Insight*. http://mathinsight.org/definition/undirected_graph

Depth First Search Tutorials & Notes: Algorithms. (n.d.). Retrieved from https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

Breadth First Search Tutorials & Notes: Algorithms. (n.d.). Retrieved from https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

Depth First Search Tutorials & Notes: Algorithms. (n.d.). Retrieved from https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

Vincent Kenny, Matthew Nathal, and Spencer Saldana (ChE 345 Spring 2014) https://optimization.mccormick.northwestern.edu/index.php/Heuristic_algorithms

(n.d.). Retrieved from https://courses.cs.washington.edu/courses/cse326/03su/homework/hw3/bestfirstsearch.html

What are the differences between A* and greedy best-first search? (2018, November 10). Retrieved from https://ai.stackexchange.com/questions/8902/what-are-the-differences-between-a-and-greedy-best-first-search

(n.d). Retrieved from https://ib.compscihub.net/wp-content/uploads/2018/07/5.1.16.pdf

Retrieved from https://www.cs.rochester.edu/~gildea/csc282/slides/C12-bst.pdf

Lowest Common Ancestor in a Binary Tree: Set 1. (2019, July 22). Retrieved from https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/

Shortest distance between two nodes in BST. (2019, June 21). Retrieved from https://www.geeksforgeeks.org/shortest-distance-between-two-nodes-in-bst/

Appendices

1. BFS(Python)

```
        from queue import Queue
q = Queue();
q.put(1);
visited = [];
visited.append(None);
for i in Graph:
        visited.append(False)
# visited.append(False)
visited[1] = True;
while not (q.empty()):
        v = q.get();
        # print(v);
        for i in Graph[v]:
        if not (visited[i]):
        visited[i] = True;
        q.put(i);
```

2. DFS(Python)

```
        visited = [];
        stack = []
        def DFS(n):
                stack.append(Graph[n]);
                visited.append(n);
                while(len(stack)):
                V = stack[0];
                del stack[0];
                for i in V:
                if i in visited:
                        continue;
                stack.append(Graph[i]);
                visited.append(i);
```

3. Greedy BFS(java, (n.d.). Retrieved from https://raw.githubusercontent.com/aliarjomandbigdeli/search-algorithms/master/src/SearchGreedyBFS.java)

```
import java.util.Comparator;

/**
 * greedy best first search algorithm
 * Greedy best-first search tries to expand the node that is closest to the goal,
 * on the grounds that this is likely to lead to a solution quickly.
 * Thus, it evaluates nodes by using just the heuristic function(h).
 *
 * @author Ali ArjomandBigdeli
 * @since 12.27.2018
 */
public class SearchGreedyBFS extends Search {
    public SearchGreedyBFS(boolean isGraph) {
        super(isGraph);
    }

    @Override
    public void execute() {
        search();
    }

    @Override
    public void search() {
        f.add(problem.getInitialState());
        nodeSeen++;
        while (!f.isEmpty()) {
            showLists();
            State s = f.remove();
            if (problem.goalTest(s)) {
                answer = s;
                createSolutionPath(s);
                return;
            }

            if (isGraph)
                e.add(s);
            nodeExpand++;
```

```
            for (Integer action : problem.actions(s)) {
                State child = problem.nextState(s, action);
                nodeSeen++;
                if (isGraph) {
                    if (!e.contains(child) && !f.contains(child)) {
                        f.add(child);
                    }
                } else {
                    f.add(child);
                }
            }
            f.sort(new Comparator<State>() {
                @Override
                public int compare(State s1, State s2) {
                    return ((Integer) (problem.h(s1))).compareTo(problem.h(s2));
                }
            });

            maxNodeKeptInMemory = Integer.max(maxNodeKeptInMemory, f.size() + e.size());

        }
    }
}
```

4. A*(Python, retrieved from https://stackabuse.com/basic-ai-concepts-a-search-algorithm/)

```
from collections import deque

class Graph:

        def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

        def get_neighbors(self, v):
        return self.adjacency_list[v]

        # heuristic function with equal values for all nodes
        def h(self, n):
        return 1

        def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
```

```
# haven't all been inspected, starts off with the start node
# closed_list is a list of nodes which have been visited
# and who's neighbors have been inspected
open_list = set([start_node])
closed_list = set([])

# g contains current distances from start_node to all other nodes
# the default value (if it's not found in the map) is +infinity
g = {}

g[start_node] = 0

# parents contains an adjacency map of all nodes
parents = {}
parents[start_node] = start_node

while len(open_list) > 0:
n = None

# find a node with the lowest value of f() - evaluation function
for v in open_list:
        if n == None or g[v] + self.h(v) < g[n] + self.h(n):
        n = v;

if n == None:
        print('Path does not exist!')
        return None

# if the current node is the stop_node
# then we begin reconstructin the path from it to the start_node
if n == stop_node:
        reconst_path = []

        while parents[n] != n:
        reconst_path.append(n)
        n = parents[n]

        reconst_path.append(start_node)

        reconst_path.reverse()

        print('Path found: {}'.format(reconst_path))
        return reconst_path
```

```
        # for all neighbors of the current node do
        for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                if g[m] > g[n] + weight:
                g[m] = g[n] + weight
                parents[m] = n

                if m in closed_list:
                        closed_list.remove(m)
                        open_list.add(m)

        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_list.remove(n)
        closed_list.add(n)

        print('Path does not exist!')
        return None


g = Graph(adjacency_list);
```

5. BST

```
        class Node:
                def __init__(self,val):
                self.val = val;
                self.leftChild  = None;
                self.rightChild = None;
                self.leaf = True;
                def __str__(self):
                return str(self.val);
```

```
class BST:
        def __init__(self,nodes=[]):
        self.nodes  = nodes;
        self.root = None if len(nodes)==0 else nodes[0];

        def insert(self,root,node):
        if self.root==None:
        self.root = node
        else:
        if(root.val<node.val):
                if root.rightChild is None:
                root.rightChild = node
                else:
                self.insert(root.rightChild, node)
        else:
                if root.leftChild is None:
                root.leftChild = node
                else:
                self.insert(root.leftChild, node)

        def inorder(self,root):
        if root:
        self.inorder(root.leftChild);
        print(root);
        self.inorder(root.rightChild);
        def pathFinding(self,root,find,path=[]):
        path.append(root.val);
        if(root==find):
        return path;
        if(((root.leftChild != None and self.pathFinding(root.leftChild, find, path)) or
        (root.rightChild != None and self.pathFinding(root.rightChild, find, path)))):
        return path;
        path.pop();
        return 0;
        def findLCA(self,root,n1,n2):
        path1 = [];
        path2 = [];
        if not self.pathFinding(root,n1,path1) or not self.pathFinding(root,n2,path2):
        return -1;
        i=0;
        while i<min(len(path1),len(path2)):
        if path1[i]!=path2[i]:
```

```
            break;
        i+=1;
        return path1[i-1];
```

6. B-Tree(Python, retrieved from https://gist.github.com/natekupp/1763661)

```
class BTreeNode(object):
        def __init__(self, leaf=False):
        self.leaf = leaf
        self.keys = []
        self.c    = []

class BTree(object):
        def __init__(self, t):
        self.root = BTreeNode(leaf=True)
        self.t    = t

        def search(self, k, x=None):
        if isinstance(x, BTreeNode):
        i = 0
        while i < len(x.keys) and k > x.keys[i]:          # look for index of k
                i += 1
        if i < len(x.keys) and k == x.keys[i]:    # found exact match
                return (x, i)
        elif x.leaf:                              # no match in keys, and is leaf ==> no match exists
                return None
        else:                                     # search children
                return self.search(k, x.c[i])
        else:                                     # no node provided, search root of tree
        return self.search(k, self.root)

        def traverse(self,root):
        for i in range(0,self.t):
        if not root.leaf:
                self.traverse(root.c[i]);
        if i in range(-len(root.keys),len(root.keys)):
                print(root.keys[i])

        def insert(self, k):
        r = self.root
        if len(r.keys) == (2*self.t) - 1:  # keys are full, so we must split
```

```
s       = BTreeNode()
self.root = s
s.c.insert(0, r)                    # former root is now 0th child of new root s
self._split_child(s, 0)
self._insert_nonfull(s, k)
else:
self._insert_nonfull(r, k)

def _insert_nonfull(self, x, k):
i = len(x.keys) - 1
if x.leaf:
# insert a key
x.keys.append(0)
while i >= 0 and k < x.keys[i]:
        x.keys[i+1] = x.keys[i]
        i -= 1
x.keys[i+1] = k
else:
# insert a child
while i >= 0 and k < x.keys[i]:
        i -= 1
i += 1
if len(x.c[i].keys) == (2*self.t) - 1:
        self._split_child(x, i)
        if k > x.keys[i]:
        i += 1
self._insert_nonfull(x.c[i], k)

def _split_child(self, x, i):
t = self.t
y = x.c[i]
z = BTreeNode(leaf=y.leaf)

# slide all children of x to the right and insert z at i+1.
x.c.insert(i+1, z)
x.keys.insert(i, y.keys[t-1])

# keys of z are t to 2t - 1,
# y is then 0 to t-2
z.keys = y.keys[t:(2*t - 1)]
y.keys = y.keys[0:(t-1)]

# children of z are t to 2t els of y.c
```

```
if not y.leaf:
z.c = y.c[t:(2*t)]
y.c = y.c[0:(t-1)]
```