Yisheng Zhou (Joseph)
Wayne B. Hayes
December 14th, 2015

Bio-Net Alignment Project Report

CS 199 Final Report, Fall 2015

Abstraction

In the previous version, dual expanding Dijkstra's algorithm uses max heap as the data structure saving prospective pairs of aligned nodes. After applying random seeds into the algorithm, the program needs to pick a random pair whose similarity is in the range [max similarity minus given threshold, max similarity]. The previous version just keeps popping out pairs with maximum similarity in the heap until the pop out pair has a similarity below a given value. Then, the program picks one pair randomly, and push every other pair back to the priority queue. Let k be the number of pairs whose similarities are in the given range. This progress takes T (k * log n + (k-1) * log n) time to push and pop, converting to O (k * log n). From a trail operation with a small threshold 0.05, this random seed progress takes 120 seconds, while the total time is 200 seconds. The random seed progress makes the program be too slow to run if the threshold is big enough. This report introduces a new data structure which can accelerate the progress of picking a random value in a given range from O (k log n) to O (log n).

Introduction

1. Using Other Heaps

The priority queue used in the previous dual expanding Dijkstra's algorithm is a binary heap, which is simple to program and debug, but is not efficient. The first idea is to change the binary heap to other heaps with faster running time.

Operation	Binary	Binomial	Fibonacci	Pairing	Brodal	Rank-pairing	Strict Fibonacci
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
insert	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
merge	$\Theta(m \log n)$[d]	$O(\log n)$[e]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

This is a form introducing the running time of all kinds of heaps. Based on this form, Fibonacci heap can be one the most efficient heaps in all operations. When the program inserts every pop out pairs, who are not chosen, back to the heap, the progress may only use O (k) time, while k is the pairs that need to be inserted. Before checking whether this time is applicable, there is another big problem which makes this idea does not work: the pop out time still keeps the same. After find the first pairs with the maximum similarity of the entire heap, the heap needs to use O (log n) time to update the heap. Then, the heap can output the next pair with the maximum similarity. Therefore, the time will still be O (k log n) for one randomly pick up.

2. Other Random Choice Methods

The previous random progress is to pick a random index. In order to get a random index, the program needs to know the number of pairs whose similarities are in the given range. If the program uses a heap and the number of acceptable pairs is k, the program needs to check at least k pairs. So, the running time cannot go below O (k). The aim of the random seeds progress is to let the program picks a random acceptable pair. It does not require an accurate and exact random probability distribution to pick a pair. Therefore, using other random pick method can be a possible choice to reduce the running time.

3. Implying Picking Randomly by Value

The program can get the maximum value in O (1) time, and calculate the domain [max similarity – given range, max similarity]. If the program picks a random value in this given range, it can find a pair with the nearest similarity of the random value. This method does not imply exact randomness into the program. For instance, if the domain is [0.8,0.9], and the pair similarity set is {0.8, 0.849, 0.8495, 0.85, 0.851, 0.9}, the pairs whose similarities are {0.8495, 0.85} almost have no chance to be chosen, though they should own one third probability distribution to be chosen.

Even with this problem, this method does imply the random seeds into the algorithm, runs faster, and can reduce running time into some acceptable length. If the program can find the nearest similarity in O (log n) time, the program can be faster, and programmers can use bigger threshold to test the program because the value k, which is the number of pairs whose similarities are in the given range, does not influence the total running time.

4. Structure

In order to make the idea works, there are several requirements. First of all, every value should be stored in sorted order. If the program uses heap like structures to save, it may fail to find the nearest value in O (1) time. So, the data structure should have a sorted array or a linked list. Secondly, the program still requires O (log n) time for inserting and removing. So, a tree or heap structure is still needed. Therefore, a possible design is to have a linked list at bottom saving all pairs with their similarities, ordering by their similarities. Above this linked list, there is a tree structure used for finding the value that the program is going to use in O (log n) time.

5. Skip List

A skip list is a data structure that allows fast search within an ordered sequence of elements. Fast search is made possible by maintaining a linked hierarchy of subsequences, each skipping over fewer elements. Its average searching time, inserting time and removing time are all O (log n), and it has a sorted list to save every value. Therefore, this data structure meets all the prerequisites mentioned before.

Design

The codes of the data structure can be found in skip_list.py. Here are some special designs used in this structure.

1. SkipNode

SkipNode is the basic element of the entire data structure. Every node has three variables. Value is a float, which is the value of the node used for sorting and searching. Info is other information related to this node. Its type is not defined. In this program, it is used to save yeast name and human name of the given similarity. The most important part is the Next. It is a list saving a set of pointers. Its length is the height of this node, and its element points to the next node in the rank this element belongs to. The first element, which is Next [0], is in the lowest rank, which also points to the next node in the sorted linked list.

2. UpdateList

UpdateList is a function requiring a given value. The progress of this function is like drawing a line in the entire structure, and returning left nearest elements in each rank. The basic theory of the entire skip list is a linked list. So, whatever the program is inserting or removing, the skip list only needs to change the values of the left most nodes, which are listed by the UpdateList function.

Using UpdateList can make the codes shorter and efficient. When inserting a new pair, the program creates a new node whose pointers point to the elements return by UpdateList point. Then, update the returned elements by letting them point to the new node. When deleting, just update the returned elements by letting them point to the elements pointed by the removed node. It can also be used to find the nearest neighborhood by comparing the value of the lowest node returned by UpdateList and the value of the node that the lowest node points to.

3. Determine the Height of a New Node

The way to determine the height of a new node in this skip list is to use a random flip coin method. When flipping head (0), it stops and returns a value that how many tails appear in this flip progress. When flipping tail (1), it keeps flipping until it flips a head. The number of tails returned from the flips experiment plus one is the height of the new node, which is also the number of pointers this new node has. In test_skip_list.py, there is a progress testing the average difference between log2(n) and the maximum height of all nodes in the entire skip list after inserting or deleting nodes for given times.

4. Max or Min Priority Queue

In order to make the skip list fit more situations, one switch is implied into this new data structure. This switch allows the skip list to change from sorting from min to max into sorting from max to min. The skip list can be regarded as a "black box," which means the programmers don't need to make sure the structure looks "logically," just let it return the right value. When the program adds a new node into our queue, the skip list makes the value= negative(value) if the switch is on, otherwise keep it. So, there can be a structure both works as max priority queue and min priority queue. Although, it may save values as negative numbers. From digital design theories, saving negative values does not require extra space in computer memory. There is already a binary digit to indicate a value is positive or negative.

Test and Debug

There is a file called test_skip_list.py included in the submitted file. It is used to test whether the skip list has a correct length of the entire linked list and saves every node in order. It tests both the max priority queue pattern and min priority queue pattern. It also tests whether the max height of the skip list is acceptable, which is mentioned in Part.3 of the Design section.

Difference between Required Skip List and Regular Priority Queue

After applying the test codes, one bug was founded. In regular priority queue, before adding a new value, the program will check whether this value is already inside this queue. If it is already there, the program will not waste time inserting this new value. However, this skip list is designed to save data rather than just saving similarity values. In the previous version, when inserting a pair whose similarity value is already inside the skip list, this new pair will not be inserted, which causes a huge amount of data to be lost. The most resent version does not check whether a new value is in the list or not. It will directly insert this new value into the list.

Future Plan

The new skip list data structure allows to pop a value efficiently. However, the randomness in this data structure is with some bias. The similarity value who is in the spare range is more likely to be chosen, while the value who is close to others has really little chance to be chosen. Therefore, there could be a curve based on the distribution of all similarity values. Get a random value, project left and project down to get the final choice. This curve needs to be updated after every insertion or deletion using O (1) time.