Advanced Algorithms assignment 1

Applied in c++

Ellis Rourke

S5057468

question 1: List storage

Problem

We are tasked with implementing a data structure able to store a list of n numbers and simultaneously keep track of the first k smallest numbers. This problem is easy solved using any sorted tree-based structure such as a min heap or balanced binary search tree.

The list needs to facilitate a very large number of random numbers being entered while still maintaining the k smallest. The target space complexity is O(k+n) although our tree-based solutions will maintain the list in just O(n) space.

Solution

As mentioned above, we could solve this problem with an array of different tree structure, but given our course work… I have chosen to implement a red-black tree.

The red-black tree allows us to insert and delete any number of numbers while maintaining a space complexity of O(n).

The red, black tree has complexities of O(Log n) for Insert, Delete and search allowing us to perform are most crucial operations in an efficient manner. Given this, we can find the first k elements in O(k log n) time.

RESULTS

AMORTISED ANALYSIS

question 2: maze generation

problem

The problem states we are to find a way to programmatically generate a simple maze by randomly knocking down walls in a 50 x 88 grid until a specified entry and exit points are connected: creating a valid solvable maze.

Solution

We can implement use a disjoint set structure to solve this problem. Disjoint sets work by keeping a list of items in unique sets where a set is some number of keys where there is no overlap with another set.

We can make use of this property by implementing the two main methods of a disjoint set, Union and Find. Union is used to join two subsets such that they are a single set and find is used to test if two items are in the same set.

We can solve the problem by first selecting two edge walls at random and breaking their respective walls to create an entry and exit point for the maze.

We then randomly knock down walls (union 2 random walls) until the entry and exit points are connected.

The complexity of the structure is O(Log n) for union and find in the worst case however the amortized cost is much closer to O(n).

RESULTS

question 3: red black vs veb

Problem

Solution

RESULTS

However, for small trees the overhead associated with vEB trees

is enormous: on the order of 2^(m/2)

question 4: kevin bacon

problem

The Kevin Bacon problem, or six degrees of bacon is a graph computation problem where we are tasked with linking one actor to another by connections that represent movies that the two actors appear in; It is said that any actor has a bacon number of no more than 6.

solution

We can solve the Kevin bacon problem by building a graph of movies and actors such that we can expand the graph and update the connections of each actor and movie as we load more data from the input file.  
  
Each line of input consists of the actor’s name, and the movie they appeared in. For each line we create and new node for the actor and movie. Unless they already exist in the graph, in this case we update the connections for the node. The resulting graph will resemble the figure below,

Diagram

Description automatically generated

Given the graph is acyclic and connected, we can simply use a modified depth first search algorithm to find a single, or the longest path between any two nodes.

The modification we can make to a standard BFS search lets us use a start and end point to set the two actors in which we are trying to connect. We also implement a findmax flag where we can search for the largest set of connections in the input file.

**Breadth\_first\_search**(graph g, node start, node end, bool findmax):

Define set s

Define queue q

**If** start **or** end == null:  
 **return** #Node does not exist

Q <- start

S <- start

**While** q not empty:

Current = q.front

Last = current

q.pop

**if** current.name == end.name **and** !findmax:

return current

**for** connection in current.connections:

**if** s.count(current.connections[connection] == 0:

s <- current.connections[connections]

q <- current.connections[connection]

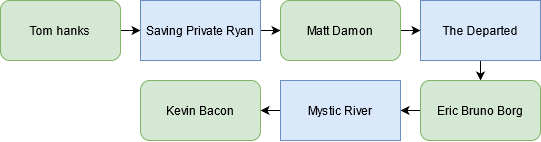
current.connections[connection].parent = current

**Return** last

RESULTS(bacon2.txt)

a)

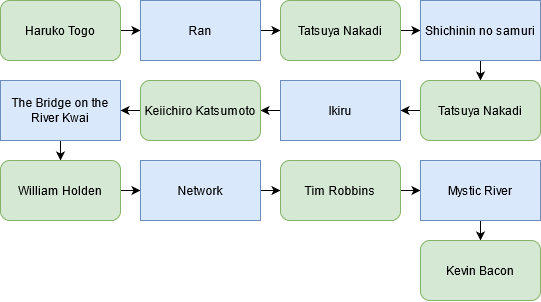
As you can see below, we can connect Kevin Bacon to Tom hanks in just 2 intermediate links, this is a low bacon number; Although, this could be expected as their careers coexisted in the same time frame.



b) find the actor with the highest bacon number

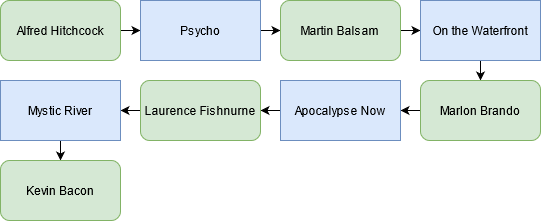
We can find the actor with the largest bacon number by slightly adapting our depth first search.

Here you can see Haruko Togo has a bacon number of 6, this is quite a large gap, although this could be expected as the actors do not have many similarities.



c) Finds the minimum number of links between two arbitrary actors

Here we are essentially doing a Bacon number of 2 given actors that do not necessarily need to be Kevin bacon. Although; in this example I have used Alfred Hitchcock and Kevin Bacon.  
As you can see, they have 4 links between them, this reflects what is expected as there is a significant time gap between their careers.



question 5: min vertex cover

Introduction

The min vertex problem is I widely studied optimization problem where given a graph G, we need to find the smallest set of vertices such that every edge in the graph has one of its endpoints in the cover. Vertex cover is an extensively researched graph theory problem.

A common derivative of this problem is to calculate the minimum number of security cameras that are required to cover every street in a city, where the graph G is the city, streets are edges and vertices are cameras.

The problem is NP-Hard and as such cannot be solved in polynomial time; rather approximation algorithms are required the calculate and search for possible solution, usually with so randomness or heuristic to discover global minimum.

Literature review

Algorithm description

The implementation of my algorithm used the depth first search algorithm to search in complement graphs. Firstly, we read the given CLQ file specified in the programs arguments then build our graph structure using an unordered map.

We then create a generic cover by adding all vertices of the graph and poll a number of random vertices to find one with the smallest degree.

When we have our start vertex, we use a generic depth first search algorithm where each time it is called, we remove the vertex with the smallest degree and call another instance of the dfs algorithm using a copy of the previous cover with the node removed.

When this is returned, we check if it is smaller than our previous cover.

Our nodes also have a penalty factor attached to them that is used to score certain nodes against others when doing our comparison to find the smallest degree. These penalty factors are also randomly reset during execution.

Cover <- Init

Graph <- Input file

Depth First Search

best cover size <- Infinity

for node in graph:

append node to cover

smallest = node in graph with smallest degree

remove node with smallest degree from cover

duplicate cover <- create a copy of the cover

search for next smallest degree node

add if to the duplicate cover

remove it from the list of options

for node in duplicate cover

if node is removed

recursive dfs

check if new cover is smaller than previous best

reset penalty factor sometimes

Experimental results

comparisons

conclusioN

RESULTS

|  |  |  |
| --- | --- | --- |
| Graph | Minimum Vertex Cover | CPU Time (Second) |
| Brock800\_1 | 783 | 0.583 |
| Brock800\_2 | 783 | 5.263 |
| Brock800\_3 | 783 | 5.556 |
| Brock800\_4c | 782 | 19.933 |
| C2000.9 | 1946 | 46.833 |
| C4000.5 | 3988 | 2.233 |
| MANN\_a45 | 690 | 0.282 |
| P\_hat1500-1 | 1492 | 0.259 |