Finding Optimal Data Layouts for Known Access Patterns: A Constructive Algorithm

Nathaniel Jones

Introduction

The traveling salesman problem is well-studied in mathematics. For good reason, too: finding the shortest route to visit every point in a given set of points is a problem with many real-world applications. Far less studied, however, is the reverse problem: given a route through a set of points, organize the points such that the distance traveled on that route is minimized. This paper presents a constructive algorithm that uses a heuristic to compute a nearly optimal solution to this problem in one dimension.

Motivation

The primary motivation for solving the aforementioned problem is improving the rate of cache hits in computer programs by increasing the locality of stored data. When loading data on a modern computer, it is substantially faster to load pieces of data that are stored close together than stored far apart. Generally, the data access pattern of a given program will be fixed or at least largely consistent, hence the need to find a data layout that minimizes the distance between subsequent data reads given a known access pattern. Even more than optimizing a program's own data reads, solving this problem is useful for the execution of the program itself. Modern processors fetch and decode instructions in large chunks rather than one-at-a-time, and they store these decoded instructions in small instruction caches. The logic for this instruction fetching can be quite complex, requiring the processor to follow jumps and to accurately predict branches to ensure that precious cycles are not wasted by decoding instructions that, despite their proximity to the current instruction, will not be executed. However, if it were known ahead of time that the majority of instructions close to the current instruction were going to be executed, much of this complex circuitry could be simplified, as the processor could just decode nearby instructions without much consideration of how the program might jump between them.

Problem Definition

Let N be a set of nodes, and P be a permutation containing all nodes in N. Suppose we start at a node n_0 in P. Another node n_1 in P can be *traversed* to by *moving* between adjacent nodes in P. Let S be an ordered list of nodes in N that defines a series of traversals between nodes. The goal, then, is to choose a permutation P such that the total number of moves necessary to complete the series of traversals S is minimized.

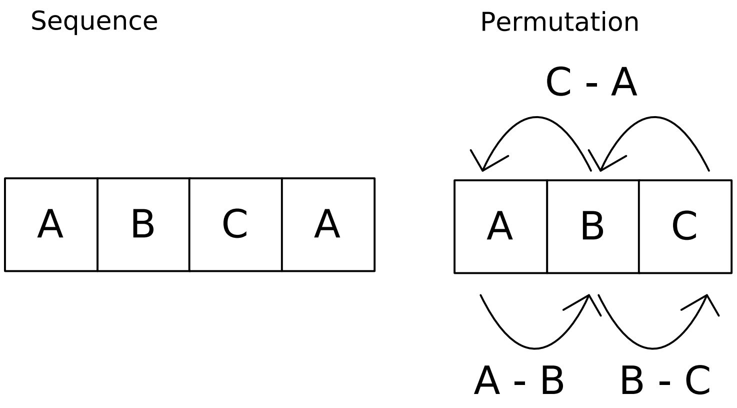

Figure 1: Illustration of following a sequence of traversals. To follow the sequence ABCA in a permutation ABC, we first start at A and traverse to B. Then we traverse from B to C. To traverse from C to A, we must move to B before moving to A, since C is not adjacent to A.

It is possible to find P by checking every possible permutation of nodes in N, computing the length of the series of traversals, and choosing the permutation that results in the minimum. However, this quickly becomes impractical; the time complexity of this brute-force search is O(n!) with respect to the number of different nodes. This fact makes brute-force searching an unappealing or even impossible option when dealing with more than a handful of nodes. A better method of finding an optimal permutation is therefore necessary.

The Algorithm

The algorithm is designed around an intuitive principle: the nodes most frequently traversed between should be closest together. The first step, then, is to determine which traversals are most frequent in S. Each possible pair of nodes in N is added to a set L and assigned a rank of 0 to start. Each individual node in N is also assigned a rank of 0. For each set of adjacent nodes in S, the respective pair's rank in L is incremented by 1, and the rank of each individual node in the pair is also incremented by 1. L is then sorted by rank in descending order.

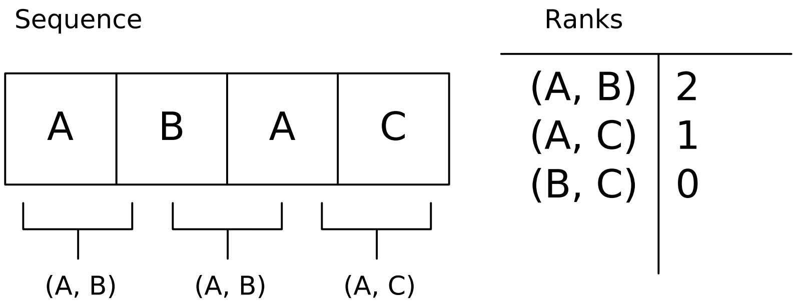

Figure 2: Illustration of ranking calculation. In the sequence ABAC, the pair (A, B) has rank 2 since A and B are adjacent to one another twice in the sequence. (A, C) has rank 1 since A and C are adjacent once. (B, C) has rank 0 since B and C are adjacent nowhere in the sequence.

The highest-ranked pair of nodes is then added to P and removed from L. For each remaining pair in L, it must be determined whether to append or prepend to P, and in which order the nodes in the pair should be inserted (assuming they are not both already present in P). For each node n_i in the current pair not already inserted into P, the node p_i in P such that the pair (n_i, p_i) has the highest rank is found. If neither node n_0, n_1 in the current pair is in P, the ranks of the (n_0, p_0) and (n_1, p_1) are compared and whichever has the greatest rank is stored in (n_r, p_r). If p_0 and p_1 do not exist (i.e. there are no nodes currently in P that are in a traversal with n_0 or n_1), n_0 and n_1 are appended or prepended to P such that the highest ranked nodes out of the head of P, the tail of P, n_0, and n_1 are the outermost nodes in P. This is done so that the highest ranked nodes can be placed next to nodes involved in traversals with them in future iterations.

If p_r does exist, the decision to append or prepend to P is made by examining the location in P of p_r. If p_r is closer to the beginning of P (i.e. its position is less than half the length of P), n_r is prepended to P, else it is appended. In either case, if the other node in the current pair is not in P, it is respectively prepended/appended after n_r. Once the nodes have been added, the current pair is removed from L. This process continues until all nodes in N have been added to P.

Results

The algorithm was implemented in Python and was tested against brute-force searching to determine the optimality of the P generated by the algorithm. 8 different nodes were used to generate 10,000 random sequences ranging from 50 to 150 nodes in length. The results are summarized in Figure 3.

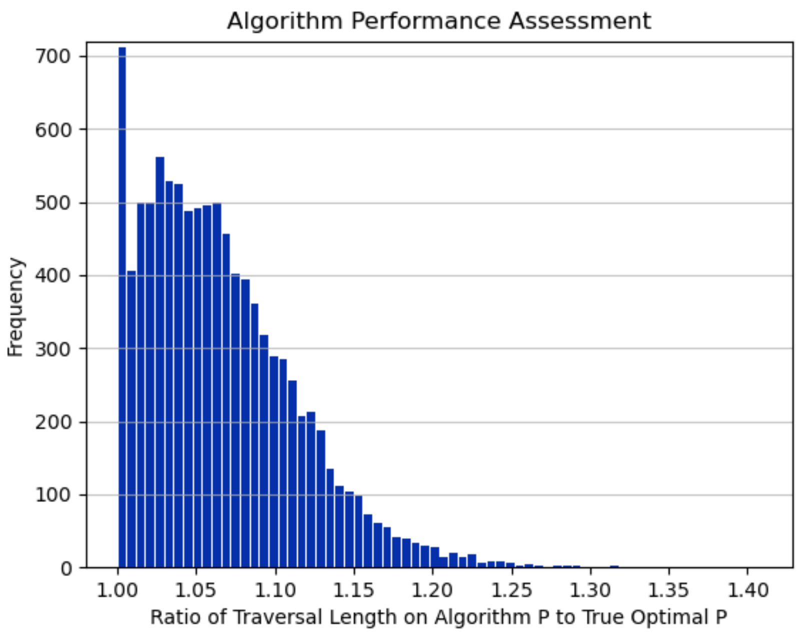

Figure 3: Histogram of algorithm performance.

As shown in the histogram, the algorithm attains at most a 15% increase in total traversal length over the true optimal solution in the vast majority of cases. In fact, in more than half the cases, the algorithm attains less than a 10% increase over the true minimum. However, the distribution has a long tail; there are several cases where the algorithm performs somewhat poorly, with a handful of cases resulting in more than a 25% increase in total traversal time over the true minimum. This may still be acceptable, though, as the algorithm is much faster to run than a brute-force search. Whereas brute-force searching has a time complexity of O(n!) with respect to the number of different nodes in the sequence, the presented algorithm has a time complexity of O(n^2) with respect to the length of the sequence.

Conclusion and Future Work

This paper has presented an effective algorithm for finding a layout of nodes to minimize the amount of movement needed to complete a sequence of traversals. However, it is not a perfect algorithm, as it rarely finds a perfectly optimal solution despite frequently getting close. Furthermore, there appear to be certain edge cases where the layout generated by the algorithm is substantially worse than the true optimum, with a total traversal cost more than 25% larger than the true minimum. Determining how these edge cases occur and altering the algorithm to account for them would be a useful contribution to the research on this problem. It might also be useful to explore how this problem could be solved in higher dimensions, where the nodes are placed on, for example, a 2D grid. Whatever direction future research takes, I hope that this paper and the algorithm presented within serves as a helpful jumping-off point and as a useful resource in its own right.

Appendix: Python Code to Generate Histogram

```
import itertools
import random
import matplotlib.pyplot as plt
import math

## User configurable parameters

# N
nodes = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

## Construct Layout

def construct_layout(seq):
    # P
    layout = []
    # L
    rankings = {}
    # Rankings of nodes in N
    node_rankings = {}

    for n in nodes:
        node_rankings[n] = 0

    for i in range(len(seq) - 1):
        pair = [seq[i], seq[i + 1]]
        pair.sort()
        node_rankings[pair[0]] += 1
        node_rankings[pair[1]] += 1

        edge = (pair[0], pair[1])
        if not edge in rankings:
            rankings[edge] = 0
        rankings[edge] += 1

    sorted_rankings = []
    for k in rankings.keys():
        sorted_rankings += [(k, rankings[k])]

    sorted_rankings.sort(reverse=True, key=lambda k: k[1])

    layout += [sorted_rankings[0][0][0]]
    layout += [sorted_rankings[0][0][1]]

    for k, r in sorted_rankings[1:]:
        highest_ranked = (k[0], k[0], len(sorted_rankings))

        for node in k:
            if node in layout:
                continue

            for v in layout:
                idx = 0
                key = sorted([node, v])
                key = (key[0], key[1])

                for edge in sorted_rankings:
                    if edge[0] == key:
                        break
                    else:
                        idx += 1

                if idx <= highest_ranked[2]:
                    highest_ranked = (node, v, idx)

        # If neither node on this edge is in the layout already nor connected to any
        # nodes that are, just append them both to the end
        if highest_ranked[2] == len(sorted_rankings) and not k[0] in layout and k[1] not in layout:
            head_rank = node_rankings[layout[0]]
            tail_rank = node_rankings[layout[1]]
            if node_rankings[k[0]] > node_rankings[k[1]]:
                if head_rank < tail_rank:
                    layout = [k[0], k[1]] + layout
                else:
                    layout += [k[1], k[0]]

            else:
                if head_rank < tail_rank:
                    layout = [k[1], k[0]] + layout
                else:
                    layout += [k[0], k[1]]
        elif not highest_ranked[0] in layout:
            not_highest = k[0]
            if highest_ranked[0] == k[0]:
                not_highest = k[1]

            connected_idx = layout.index(highest_ranked[1])
            if connected_idx < len(layout) // 2:
                if not not_highest in layout:
                    layout = [not_highest, highest_ranked[0]] + layout
                else:
                    layout = [highest_ranked[0]] + layout

            else:
                if not not_highest in layout:
                    layout += [highest_ranked[0], not_highest]
                else:
                    layout += [highest_ranked[0]]

    return layout


## Compute distance

def distance(seq, arr):
    cur_idx = arr.index(seq[0])
    dist_sum = 0

    for node in seq[1:]:
        idx = arr.index(node)
        dist_sum += abs(idx - cur_idx)
        cur_idx = idx

    return dist_sum

p = list(itertools.permutations(nodes))

ratios = []
for k in range(10000):
    sequence_length = random.randint(50, 150)
    sequence = [random.choice(nodes)]
    while len(sequence) < sequence_length:
        c = random.choice(nodes)
        while c == sequence[-1]:
            c = random.choice(nodes)

        sequence += [c]

    min_brute_force = (0, distance(sequence, p[0]))
    for i in range(len(p) - 1):
        l = p[i+1]
        d = distance(sequence, l)
        if d < min_brute_force[1]:
            min_brute_force = (i+1, d)

    algo_dist = distance(sequence, construct_layout(sequence))
    ratios += [algo_dist / min_brute_force[1]]

m, bins, patches = plt.hist(x=ratios, bins='auto', color='#0530aa', rwidth=0.85)
plt.grid(axis='y', alpha=0.75)
plt.xlabel('Ratio of Traversal Length on Algorithm P to True Optimal P')
plt.ylabel('Frequency')
plt.title('Algorithm Performance Assessment')
maxfreq = m.max()
plt.ylim(ymax=math.ceil(maxfreq / 10) * 10 if maxfreq % 10 else maxfreq + 10)
plt.savefig("performance.png")
```