

A CAUTIONARY TALE: FALSE EFFICIENCIES IN THE TRAVELING SALESMAN PROBLEM*

Timothy J. Rolfe
Computer Science Department
Eastern Washington University
319F Computing & Engineering Bldg.
Cheney, WA 99004-2493
509 359-6162
Timothy.Rolfe@ewu.edu

ABSTRACT

The Traveling Salesman Problem provides an opportunity to explore whether attempts to optimize an algorithm in fact succeed or fail. Attempted optimizations themselves have time penalties in their execution and those time penalties may be greater than the improvements provided to the algorithm. When approaching the Traveling Salesman Problem not as a graph traversal problem but as a backtracking permutation problem, one may attempt a “best-first” optimization in the recursion. This significantly slows the program. One may also attempt forward-bounding functions. Two such functions are examined, of which one slows the program and the other neither slows nor speeds the program.

INTRODUCTION

The author teaches undergraduate courses in a regional state university, and so is familiar with the phenomenon of someone’s trying to avoid work — and spending more effort in that avoidance than the original task would have required.

The same phenomenon can be seen in computer algorithms. For instance, a well-known example occurs in Selection Sort — avoiding the swap of a cell with itself.

```
for (lim = n-1; lim > 0; lim--)  
{ int big = findLargest(x, lim);  
  if (big != lim)
```

* Copyright © 2008 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

```

    swap(x, lim, big);
}

```

Although the integer comparison “big != lim” is fast, it still takes time. For random data the probability that one can omit the swap is $1/n$ — very small. Thus the faster approach is to omit the comparison and allow the rare swap of an item with itself.

Another example is provided by binary search: whether to allow for an early exit or to drive the search region down to a single cell:

```

// Early exit
for (lo = 0, hi = n-1; lo <= hi; )
{
    int mid = (lo + hi) / 2;
    if (x[mid] < sought)
        lo = mid + 1;
    else if (x[mid] > sought)
        hi = mid - 1;
    else
        return mid;
}
return -1;
// No early exit
for (lo = 0, hi = n-1; lo < hi; )
{
    int mid = (lo + hi) / 2;
    if (x[mid] < sought)
        lo = mid + 1;
    else
        hi = mid;
}
return (x[lo] == sought)? lo : -1;

```

If the three-way comparison is handled as shown in the above code, each loop iteration requires on average $1\frac{1}{2}$ comparisons (though the compiler may optimize it by comparing $x[mid]$ with `sought` once and then using machine instructions such as `BRANCH_MINUS` and `BRANCH_PLUS`).

Under these circumstances, it is obvious that the early exit option is inappropriate on searches that fail: $\frac{3}{2} \lceil \lg(n) \rceil$ comparisons are always required rather than $\lceil \lg(n) \rceil$ without the early exit. On searches that succeed, the early exit option can be shown to reduce loop iterations on average only by one. [1]

TRAVELING SALESMAN AS BACKTRACKING RECURSION

While the Hamiltonian Cycle Problem (and hence the Traveling Salesman Problem) can be solved by a traversal method exploring all possible paths, such as one proposed by Antonakos and Mansfield, [2] it can also be considered as discovering a permutation of vertices within a directed graph that represents a “cycle” (or “tour”); that is, edges exist between all vertex pairs in the permutation, and in addition an edge exists from the last vertex in the permutation to the first. Solving the problem does not require generating the full weighted graph application; one just needs to be able to detect whether there is a connection between two vertices. Thus from the graph specification one can

capture just the weight matrix: let a zero in `wt[j][k]` mean that there is no edge from vertex `j` to vertex `k`; otherwise the entry is the weight of that edge.

The discussion below assumes that the recursive method for the backtracking has the following parameter list: the `index` within the permutation being explored, the permutation vector itself (`vect`), and finally the distance represented by the partial path represented by `vect[0]` up through `vect[index-1]` (`so_far`). In addition, there is a global variable `tourCost` for the best tour so far and the global weight matrix `wt`. The code that handles partial permutations follows.

```
int k, hold;
for (k = index; k < n; k++)
{ swap (vect, index, k);
  if (wt[vect[index-1]][vect[index]] > 0) // Edge exists
  { // Total length must not be more than a known solution
    int testLength = so_far + wt[vect[index-1]][vect[index]];
    if (testLength < tourCost) // Process if not too long
      tour(index+1, vect, testLength);
  }
}
// Regenerate the permutation vector
hold = vect[index];
for ( k = index+1; k < n; k++ )
  vect[k-1] = vect[k];
vect[n-1] = hold;
```

The available vertices are those indicated in `vect[index]` through `vect[n-1]`. Cycle these through `vect[index]`, then check whether there is an edge from `vect[index-1]` into `vect[index]`. If there is, and if adding it does not bring the length committed to more than what one has already discovered as a tour length, proceed with the recursion. The swaps performed have the effect of a circular rotation to the right by one in positions `[index]` through `[n-1]`, so a circular rotation by one to the left corrects for that. [3]

Since the recursion proceeds only where there is an edge between two vertices, one examines massively fewer permutations that would occur if you just examined all $n!$ possible permutations.

BEST-FIRST OPTIMIZATION

Sahni, in his discussion of this problem [4], suggests a “best-fit-first” approach (specifically in a branch-and-bound context). In the backtracking context, this would mean generating a vector of adjacent vertices ordered from nearest to farthest. Since only certain vertices are being swapped into the `[index]` position, each vertex needs to be swapped back into position following use.

```
int j, k; // Loop variables
int nEdges = 0, adjacent[] = new int[n];

// Fill the adjacency vector -- effectively an insertion sort
for ( k = index; k < n; k++ )
```

```

if (wt[vect[index-1]][vect[k]] > 0)
{ for (j = nEdges; j > 0; j--)
  {
    if (wt[vect[index-1]][vect[adjacent[j]]] >
        wt[vect[index-1]][vect[k]])
      adjacent[j] = adjacent[j-1];
    else
      break;
  }
  adjacent[j] = k;
  nEdges++;
}
// Best-first portion
for (j = 0; j < nEdges; j++)
{ int testLength;

  k = adjacent[j];
  testLength = so_far + wt[vect[index-1]][vect[k]];
  if (testLength < tourCost)
  { swap (vect, index, k);      // Move k to index position
    tour (index+1, vect, testLength);
    swap (vect, index, k);      // Undo
  }
}

```

Unfortunately the additional expense of handling the sorted adjacency vector totally swamps the benefit of finding a short tour early. Indeed, one can isolate the expense of using an adjacency vector by simply removing the positioning logic. Once that is done, one finds that there is a noticeable expense to using the adjacency vector. One also finds that the sorting logic by itself requires significantly more time, with no balancing benefit from the early discovery of a short tour.

FORWARD BOUNDING OPTIMIZATIONS

Another possible optimization is to find a lower bound for tours sharing the same front end in their permutations. Sahní [4] suggests adding together the shortest edges out of the vertices that have not yet been committed (that is, in positions [index+1] through [n-1]). The tour of any permutation growing out of this partial permutation necessarily has a length greater than or equal to the length committed plus this sum.

To support this, during reading in the weight matrix for the graph one can generate a vector (minAddedCost) to hold the length of the shortest edge out of each vertex. Then, when there is an edge between vect[index-1] and vect[index], one can quickly add together the sum.

```

if (wt[vect[index-1]][vect[index]] > 0)
{ int increment = 0, j,

  for (j = index+1; j < n; j++)
    increment = increment + minAddedCost[vect[j]];
  testLength = so_far + wt[vect[index-1]][vect[index]];
  if (testLength+increment < tourCost)
    tour (index+1, vect, testLength);
}

```

}

Unfortunately, the expense of that little loop to generate the increment swamps the benefit of pruning the decision tree early.

One can, however, generate a forward bounding vector that will just require subscripting in an array. While reading in the weight matrix, one can generate a vector with all of the edge weights. After sorting that, one can generate a vector (`wtBound`) so that `wtBound[k]` represents the sum of the $(n-k-1)$ lowest edge weights. If `lngth[]` is a vector that contains all `nEdge` weights, one can do the following:

```
Arrays.sort(lngth, 0, nEdge);
// Fill [0] through [n-2], [n-1] is 0.
for (j = 0; j < n-1; j++)
// Sum into cells.
    for ( k = n-j-2; k > 0; --k )
        wtBound[k] += lngth[j];
```

Once one has this bounding vector, pruning the decision tree early is easy.

```
if (wt[vect[index-1]][vect[index]] > 0)
{ testLength = so_far + wt[vect[index-1]][vect[index]];
  if (testLength+wtBound[index] < tourCost)
    tour (index+1, vect, testLength);
}
```

At last there is an optimization that does not end up requiring *more* time than the bare un-“optimized” code. One could say that it at least meets the medical requirement “*primum non nocere*” (firstly, do no harm). [5] Unfortunately it also does not reduce the time required.

NUMERICAL RESULTS

Java programs representing the above optimization attempts were run on a sample data set (eastern Washington and northern Idaho towns and cities, with 27 vertices and 50 edges) on the author’s office computer for 100 times so as to capture times in the seconds, eight runs per program. Because of space constraints, only the averages are shown (individual results are available on the web page given below). The first entry is from the original code. The second entry shows the full Best-First optimization, while the third entry shows the cost of simply *having* an adjacency vector rather than examining the weight matrix to determine adjacency. The final two entries show the two forward bounding methods in the order in which they were discussed above.

Bare Cost	Best First	Adj. Vect.	1 st Bound	2 nd Bound
4.32	5.39	4.84	5.18	4.35

SUMMARY

As the Shaker hymn says, it’s a “gift to be simple.” [6] If the underlying algorithm is extremely terse, attempts to improve on it by adding complexity may well actually hurt rather than help. For instance, the author has seen the presentation of selection sort with early exit (found when the search for largest entry never skips over a cell) — even though on random data such an early exit is rare. The expense of maintaining the boolean flag

for sorted data obliterates any benefit obtained when occasionally the loop does not execute the full $n-1$ times — about a 5% slow-down. [If the data might be nearly sorted, one should be using insertion sort anyway.]

WEB RESOURCE

<http://penguin.ewu.edu/~trolfe/TSP/index.html>. This page provides access to this paper, and also to the full Java programs exercising the variations on the Traveling Salesman Problem from which the above code excerpts were taken, along with several specimen data files, an Excel workbook giving the results of the numerical experiment reported above, and, in addition, material developed since the preparation of this paper in May and the delivery of the paper in October — including equivalent C code.

ACKNOWLEDGEMENTS

These results were obtained using equipment within the Computer Science Department at Eastern Washington University.

REFERENCES

- [1] Rolfe, T., “Analytic Derivation of Comparisons in Binary Search”, **SIGNUM Newsletter**, Vol. 32, No. 4 (October 1997), pp. 15-19. A link to the article is available in the “Publications area” of the author’s web site: <http://penguin.ewu.edu/~trolfe/#PUB>
- [2] Antonakos, J., and Mansfield, K., *Practical Data Structures Using C/C++*, Prentice-Hall, 1999, pp. 268 ff. The irony is that they develop their `all_paths` procedure (potentially exponential or worse in time) to find the shortest path in a graph, which can be done by Dijkstra’s algorithm in quadratic time. See, for instance, <http://mathworld.wolfram.com/DijkstrasAlgorithm.html> or <http://www.nist.gov/dads/HTML/dijkstraalgo.html>
- [3] Rolfe, T., “Backtracking Algorithms”, **Dr. Dobb's Journal**, Vol. 29, No. 5 (May 2004), pp. 48, 50-51. A link to the article is available in the “Publications area” of the author’s web site: <http://penguin.ewu.edu/~trolfe/#PUB>
- [4] Sahni, S, *Data Structures, Algorithms, and Applications in Java*, 2nd ed. Silicon Press, 2005, pp. 891-95 — <http://www.cise.ufl.edu/~sahni/dsaj/chapters.htm>, retrieved May 6, 2008.
- [5] See http://en.wikipedia.org/wiki/Primum_non_nocere
- [6] See, for instance, http://en.wikipedia.org/wiki/Simple_Gifts