Proceedings of the Third International DERIVE/TI-92 Conference

The Traveling Salesperson's Toolkit

Mark Michael, Ph.D.
King's College, 133 N. River St.
Wilkes-Barre, PA 18711, USA
mmichael@gw02.kings.edu

Introduction

In theoretical computer science, NP-complete problems play an important role. Of hundreds of such problems **[2]**, the most famous is the Traveling Salesperson's Problem or TSP. It is actually a combinatorial optimization problem (*Given a weighted, directed graph, find the directed Hamiltonian cycle of minimum weight.*) which is recast as a decision problem (*Given a weighted, directed graph and a constant* `W`*, is there a directed Hamiltonian cycle of weight less than* `W`*?*) to satisfy membership in the class NP. The problem and the challenge it presents can easily be explained to the uninitiated. It is germane to courses in Discrete Mathematics, Graph Theory, Combinatorics, and Theory of Algorithms.

While it is important for students to see abstract mathematical proofs that a problem is NP-complete or that a approximate solution is within `X` of being optimal, it is also important that students work with concrete instances of problems. The TSP Toolkit provides a collection of *DERIVE* functions which allow students to experiment with specific graphs without having to perform tedious calculations by hand or having to write computer programs. The Toolkit is a work in progress: functions can be combined or modified, and there is room for growth in a number of directions.

Preliminaries

In this discussion, our weighted, directed graph is represented by a matrix whose `[i,j]`-th entry is the cost of traveling from Vertex `i` to Vertex `j`. We use ∞ as the value for (1) all the entries along the diagonal (to prohibit unnecessary travel from a vertex immediately back to itself) and (2) any entry representing a *nonexistent* edge.

In all our functions, the following variables will be used consistently to represent certain types of data:

`m`	a matrix
`p`	a path *or* a permutation (which is a special kind of path)
`v`	a vector (not necessarily a path)
`e`	an edge (a vector of length `2`)
`i,j,k`	a vertex *or* an index variable
`n`	the number of vertices in a graph *or* the size of a vector

Usually, `n` will be computed as the `DIMENSION` of an input value, such as `m`, `p`, or `v`.

The most fundamental tool in the TSP Toolkit computes the total weight of a given directed path in a given graph. It is a constituent of many other Toolkit functions.

$$\text{TOTAL\_PATH\_WT}(m,p) := \sum_{i=1}^{\text{DIMENSION}(p)-1} m\downarrow[p\downarrow i, p\downarrow(i+1)]$$

A student could apply this function in trial-and-error fashion (perhaps in a friendly competition with classmates) to candidates for Shortest Hamiltonian Cycle.

Brute Force

The trial-and-error approach can be carried to the point of leaving no stone unturned. To compare the weights of all possible Hamiltonian cycles, one needs to generate the complete set of permutations of $[1,2,3,\ldots,n]$. The usual method — though not the most efficient — is to list them in the canonical order **[1]**. This hinges on being able to compute the lexicographic successor of any given permutation $[p_1,p_2,p_3,\ldots,p_n]$. The algorithm for doing so is:

Step 1: Find the largest k such that $p_k < p_{k+1}$.
Step 2: Find the largest $j > k$ such that $p_j > p_k$.
Step 3. Interchange p_j and p_k.
Step 4. Reverse the order of $p_{k+1}, p_{k+2}, \ldots, p_n$.

Step 1 can be accomplished by first constructing a vector which encodes the positions in which the inequality is satisfied and then letting MAX pick the last position:

```
FIND_K(p):=MAX(VECTOR(IF(p↓k<p↓(k+1),k,0),k,1,DIMENSION(p)-1))
```

Encoding positions is also needed in Step 2, but starting in Position k+1. There is an implicit special case to handle: if the original permutation was in descending order, FIND_K returned a value of zero, and FIND_J will do likewise since we do not need to construct a vector.

```
FIND_J(p):=IF(FIND_K(p) = 0, 0,
  MAX(VECTOR(IF(p↓j>p↓
FIND_K(p),j,0),j,FIND_K(p)+1,DIMENSION(p))))
```

It is possible to combine Steps 3 and 4 by taking care to keep track of one special position, viz., the entry which receives what was originally p_k and was moved to Position j (which is greater than k) by Step 3. We can categorize the final values in each position of the new permutation in the following table:

Positions 1 to k-1	receive	their original values.
Position k	receives	the value from Position j.
Position n-j+k+1	receives	the value from Position k.
Any other Position i	receives	the value from Position n-i+k+1.

(Alternatively, one could outline this in pseudo-code with nested if-then-else structures.) A single function can incorporate all four steps of the Next-Permutation Algorithm and even provide an "error message" for the case when [n,n-1,n-2,...,1] is input:

```
NEXT_PERM(p):=IF(FIND_K(p)=0,"No Next Perm",
  VECTOR(IF(i<FIND_K(p),p↓i,IF(i=FIND_K(p),p↓FIND_J(P),
  IF(i=DIMENSION(p)-FIND_J(p)+FIND_K(p)+1,p↓FIND_K(p),
  p↓(DIMENSION(p)-i+FIND_K(p)+1)))),i,1,DIMENSION(p)))
```

To generate the entire set of permutations, one could start with `[1,2,3,...,n]` and iteratively compute `n!-1` successors.

```
LIST_PERMS(n):=ITERATES(NEXT_PERM(p),p,VECTOR(i,i,1,n),n!-1)
```

In practice, the time-complexity of this function is worse than O(`n!`). In fact, whenever an intermediate value is needed for several subsequent calculations, a *DERIVE* function typically exceeds the time-complexity indicated by pseudo-code and obtainable via a traditional programming language. This is because the intermediate value cannot merely be stored and retrieved as needed; it must be recomputed each time it is accessed. For example, `NEXT_PERM` must *at least* call `FIND_K` (and possibly `FIND_J`, which itself calls `FIND_K`) *for <u>each</u> element of the vector!*

One could define a function which applies `TOTAL_PATH_WT` to each element of `LIST_PERMS`. This is inappropriate for two reasons. First, each permutation is a Hamiltonian path, but not a cycle — it visits each vertex exactly once without returning to the starting vertex as we need. It is possible to attach to the end of each permutation its own first element. However, it is simpler and computationally faster to adjust how we compute the weight of a path, as in:

```
TOTAL_CYCLE_WT(m,p):=TOTAL_PATH_WT(m,p) + m↓[p↓DIMENSION(p),p↓1]
```

Given a simple path `p`, this adds in the weight of an edge back to the starting vertex.

The second problem is that there are actually only `(n-1)!` distinct Hamiltonian cycles, each of which can begin at `n` different vertices. Thus, all our permutations can begin at Vertex `1`. One way to accomplish this is to cut off `LIST_PERMS` at `(n-1)!-1` iterations, just as it is about to generate the first permutation starting with `2`. An alternative is to generate the permutations of `[2,3,4,...,n]` and add another extra term when computing the total weight; this is less expensive because `NEXT_PERM` is operating on shorter vectors.

It is possible to have a single *DERIVE* function combine many jobs, even keeping track of which permutation is optimal. This is not only a programming morass in *DERIVE*, it is prohibitively slow. It is better to provide students with a set of tools which they can apply in sequence. They should be able to understand why each is needed.

To list all the permutations of `[2,3,4,...,n]`, use:

```
LIST_1LESS_PERMS(n):=
  ITERATES(NEXT_PERM(p),p,VECTOR(i,i,2,n),(n-1)!-1)
```

To take a short permutation $[p_2, p_3, p_4, \ldots, p_n]$ and compute the total weight of the cycle $[1, p_2, p_3, p_4, \ldots, p_n, 1]$, use:

```
TOTAL_HAM_CYCLE_WT(m,p):=
  m↓[1,p↓1] + TOTAL_PATH_WT(m,p) + m↓[p↓DIMENSION(p),1]
```

For a list of all total tour weights, feed the matrix *and* the set of permutations to:

```
HAM_CYCLE_WTS(m,s):=
  VECTOR(TOTAL_HAM_CYCLE_WT(m,s↓i),i,1,DIMENSION(s))
```

To find the position of the (last) entry in vector `v` having value `x`, use:

```
FIND_ENTRY(x,v):=
  (ITERATE(IF(x=v↓i,[i+1,i],[i+1,j]),[i,j],[1,0],DIMENSION(v)))↓2
```

An index of zero is returned if no entry with the desired value is found.

To retrieve the `i`-th permutation from the entire set `s` and extend it to a cycle, use:

```
FIX_CHOSEN_CYCLE(s,i):=APPEND([1],s↓i,[1])
```

The student can use these tools in the following fashion:

1) Use `LIST_1LESS_PERMS` to generate all permutations of `[2,3,4,...,n]`.
2) Apply `HAM_CYCLES_WTS` to the matrix and the list of proto-cycles obtained in 1).
3) Apply the standard `MIN` function to the list of tour weights obtained in 2).
4) Apply `FIND_ENTRY` to the minimum value from 3) and the list of weights from 2) to find the (last) position in the list where the minimum is attained.
5) Apply `FIX_CHOSEN_CYCLE` to the list of permutations from 1) and the index from 4) to retrieve the proper permutation and print it as a cycle.

There many ways to combine these functions to further automate the process of finding an optimal tour, but the programmer is warned of time-complexity problems, both in programming and execution!

The preceding development is not meant to imply that exhaustive search is one of our goals. It is merely a means to an end, that being to verify exact solutions or evaluate the merit of approximate solutions obtained by other methods introduced herein.

Branching, Bounding, *and* Backtracking

The NP-completeness of TSP does not imply that an exhaustive search among all possible Hamiltonian cycles is required to be assured of finding the optimal tour. The Branch-and-Bound Method is one way to obtain a cycle of minimum weight while judiciously ignoring some cycles. It illustrates pruned searching using an objective function (the lower bound), which is an important technique in Artificial Intelligence.

What we are searching and pruning is a binary decision tree. At each node in the tree, the decision is whether or not to include a particular edge in the cycle. (Consequently, the theoretical time-complexity of this approach is exponential, rather than factorial.) The edge is chosen for consideration (inclusion/exclusion) based on its having the minimum *reduced* weight among unused edges emanating from the vertex most recently added to the cycle. The key to this method is, reducing matrices and computing lower bounds on the total weight of tours (1) that include or (2) that exclude a given edge.

DERIVE is poorly suited for executing this entire method from start to finish in a single function. However, it can provide a few tools that will allow the student to interactively work through the method more quickly or in more instances than would be possible by hand.

Associated with the root of the search tree is a reduced matrix obtained by subtracting from each row the minimum value for that row and then subtracting from each column the minimum value for that column.

```
REDUCE_ROWS(m):=VECTOR
  (m↓i-MIN(m↓i)·VECTOR(1,i,1,DIMENSION(m)),i,1,DIMENSION(m))
REDUCE_COLUMNS(m):=REDUCE_ROWS(m`)`
REDUCE_MATRIX(m):=REDUCE_COLUMNS(REDUCE_ROWS(m))
```

The lower bound for the original matrix is a cost which *must* be incurred by *any* Hamiltonian tour. It is also the total of all the minima subtracted by `REDUCE_MATRIX`.

$$\text{LOWER\_BOUND1}(m):=\sum_{i=1}^{\text{DIMENSION}(m)}\text{MIN}(m\downarrow i)$$

```
LOWER_BOUND2(m):=LOWER_BOUND1(m`)
LOWER_BOUND(m):=LOWER_BOUND1(m)+LOWER_BOUND2(REDUCE_ROWS(m))
```

As we move down the decision tree, we update our current lower bound two ways: with and without that edge in the cycle. The calculation is performed differently in each case.

To exclude an edge `e` from consideration in a calculation, we "kill" its position in the matrix by setting it equal to ∞. We can then compute a lower bound for all tours that *do not* use that edge. Note that, in general, this is higher than the original lower bound for `m`.

```
AVOID(m,e):=VECTOR(VECTOR(
  IF(i=e↓1∧j=e↓2,∞,m↓[i,j]),j,1,DIMENSION(m)),i,1,DIMENSION(m))
LOWER_BOUND_WITHOUT(m,e):=
  LOWER_BOUND(m)+LOWER_BOUND(AVOID(REDUCE_MATRIX(m),e))
```

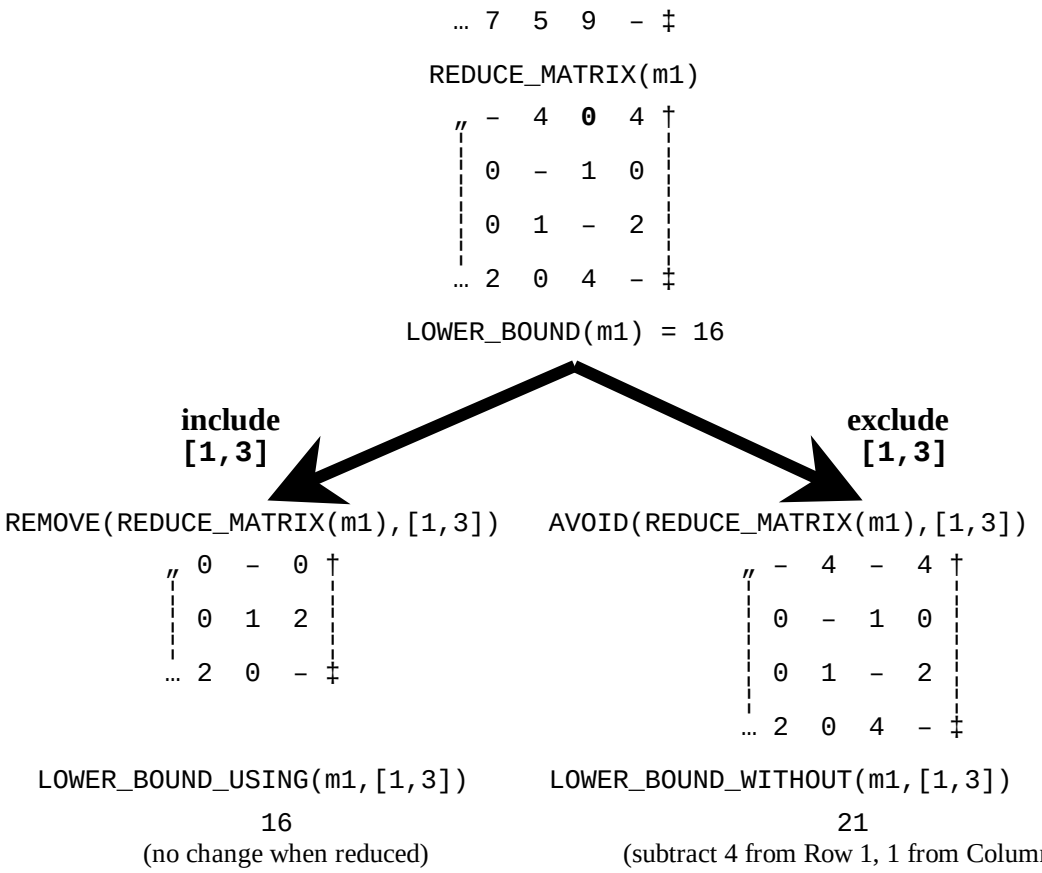
If we want to commit (tentatively) to including a particular directed edge `[i,j]` in the cycle, we remove it from the matrix in a more drastic way: we delete the entire `i`-th row so no other edge can travel from Vertex `i`, and we delete the entire `j`-th column so no other edge can travel to Vertex `j`. Then, of course, we compute the updated lower bound.

```
REMOVE(m,e):=DELETE_ELEMENT(DELETE_ELEMENT(m,e↓1)`,e↓2)`
LOWER_BOUND_USING(m,e):=
  LOWER_BOUND(m)+LOWER_BOUND(REMOVE(REDUCE_MATRIX(m),e))
```

On the top of the next page is the beginning of the decision tree for one particular example (taken from **[5]**) with both lower bounds and some intermediate matrices shown.

m1

–	7	3	8
4	–	5	5
3	4	–	6



To be honest, the preceding tools are mere stone knives, too crude to be used conveniently in this context. Because rows and columns must be removed from `m`, we need to keep track of which rows and columns are still represented in the updated matrix. It is possible to create an `(n+1)`-by-`(n+1)` matrix with row and column headings, but this makes some of the functions very awkward to write. It is easier to define functions to operate on a small data structure consisting of a matrix and two vectors, one for row numbers and the other for column numbers.

It must be remembered that the decision to choose an edge at any stage is not irrevocable. The "Branch-and-Bound" title fails to convey that *backtracking* may also be needed. One continues downward in the decision tree only as long as one of the two lower bounds associated with a decision does not exceed any of the lower bounds already computed and (temporarily) rejected. Thus, as a student uses these tools, an auxiliary tool needed is a hand-drawn tree with lower bounds attached to each node.

Greedy Algorithms

Greedy algorithms are the most simplistic. They are short-sighted but fast. Yet in many situations in computer science, a greedy algorithm will actually succeed in finding the correct answer. However, when the problem is NP-complete, a greedy algorithm will fail in general. For such a problem, students can gain a better feeling for its intractability by

seeing how poorly a greedy algorithm performs. Our brute-force tools will give a reliable measure of the extent to which a "greedy cycle" fails to be optimal.

In the case of TSP, the obvious greedy approach is to start at a vertex and, at each step, proceed to the nearest unvisited neighbor. It is possible for the weight of the greedy solution to exceed the optimal solution by an arbitrarily high factor **[4]**. The instructor — or, better yet, the students — can concoct examples to make this phenomenon tangible. The TSP Toolkit has functions which can be modified and combined to pursue the kind of greedy search described.

Alternatively, a more far-sighted greedy search can be accomplished by merely applying the Branch-and-Bound Method *without* backtracking, i.e., by stubbornly heading downward in the decision tree, taking the seemingly best branch at each level and never looking back at any previously computed lower bounds.

Lowering the Goal, Raising the Speed

A standard tactic in grappling with optimization problems related to NP-complete decision problems is to settle for a less-than-perfect solution in return for polynomial computation time. To be of any value, the algorithm for producing an approximate solution must come with a guarantee as to how far from optimal the approximation might be. In the case of TSP, the most popular such algorithm in textbooks, the Twice-around-the-Minimal-Spanning-Tree Algorithm, as well as the Minimum-Weight-Matching Algorithm each require first procuring a minimal spanning tree; consequently, they do not readily lend themselves to *DERIVE*'s limited functional programming capabilities.

In the case of symmetric graphs satisfying the triangle inequality, one approximation algorithm that can be implemented via *DERIVE* involves successively enlarging an initially trivial cycle. The Quick Traveling Salesperson Tour **[5]** takes a broader view than our greediest algorithm in that, at each stage, it looks for the unvisited vertex nearest any point on the path chosen so far. The guarantee is that the total weight of the cycle obtained from QTST is no more than twice the true minimum. (The guarantee is the same for the Twice-around-the-Minimal-Spanning-Tree Algorithm; the maximum error factor is `1.5` for the Minimum-Weight-Matching Algorithm **[3]**.)

First we create a Boolean function to determine if the value `x` is an entry of vector `v`.

```
IS_IN(x,v):=
  ITERATE([k+1,b∨(x=v↓k)],[k,b],[1,false],DIMENSION(v))↓2
```

Next, we kill all edges except those going from a vertex *not* on the path `p` to a vertex that *is* on the path. This involves making all entries in certain rows and columns ∞.

```
NONPATH2PATH(m,p):=VECTOR(VECTOR
  (IF(¬IS_IN(i,p)∧IS_IN(j,p),m↓[i,j],∞),j,1,DIMENSION(m)),
  i,1,DIMENSION(m))
```

We need the ability to locate the edge of minimum weight in a matrix. An iterative structure keeps track of where the minimum is attained. The matrix is traversed in left-to-

right, top-to-bottom order, and the first occurrence of the minimum is the one reported. The position in the matrix (in the order traversed) is encoded as a single integer which runs from `0` for `[1,1]` to n^2-1 for `[n,n]`. (This is the same as accessing a two-dimensional array as if it were a one-dimensional array in programming languages which allow this.) In the following function, `a` represents the lowest amount encountered so far, `c` is the position of that amount, and `d` is the position index which is always incremented. Both `a` and `d` act as temporary memories only. The `c`-component holds the value ultimately returned by the function.

```
FIND_MIN(m):=(ITERATE
  (IF(m↓[FLOOR(d,DIMENSION(m))+1,MOD(d,DIMENSION(m))+1]<a,
  [d+1,d,m↓[FLOOR(d,DIMENSION(m))+1,MOD(d,DIMENSION(m))+1]],
  [d+1,c,a]),[d,c,a],[0,0,∞],DIMENSION(m)^2))↓2
```

To find the edge in `m` whose distance to some vertex in the current path `p` is minimal, we apply `FIND_MIN` to `NONPATH2PATH` and use integer division and `MOD` to decode the resulting integer into a pair `[i,j]` of vertices. The first vertex, `i`, is inserted into the path prior to the last occurrence of the vertex to which it is closest, namely, `j`. ("*Last* occurrence" is handy for us because we will initially be inserting a vertex into `[f,f]`.)

```
INSERT_NEAREST_VERTEX(m,p):=INSERT_ELEMENT
  (FLOOR(FIND_MIN(NONPATH2PATH(m,p)),DIMENSION(m))+1,p,
  FIND_ENTRY(MOD(FIND_MIN(NONPATH2PATH(m,p)),DIMENSION(m))+1,p))
```

With the machinery developed, it is now easy to write the entire Quick Traveling Salesperson Tour as a single function. We specify `f`, the first vertex of the cycle, and a simple iteration adds vertices to the trivial cycle `[f,f]`.

```
QTST(m,f):=
  ITERATE(INSERT_NEAREST_VERTEX(m,p),p,[f,f],DIMENSION(m)-1)
```

Simply changing "`ITERATE`" with "`ITERATES`" gives us a complete picture of how the final answer was built up, step by step.

Following is an illustration of matrices examined by `QTST`, with the chosen edge in boldface within each matrix and the current path (as an argument of `NONPATH2PATH`) shown above. The final tour generated for `m2` is `[1,5,3,4,2,6,1]`; its weight is `24`.

m2 (original)

–	3	3	2	7	3
3	–	3	4	5	5
3	3	–	1	4	4
2	4	1	–	5	5
7	5	4	5	–	4
3	5	4	5	4	–

NONPATH2PATH(m2,[1,1])

–	–	–	–	–	–
3	–	–	–	–	–
3	–	–	–	–	–
2	–	–	–	–	–
7	–	–	–	–	–
3	–	–	–	–	–

NONPATH2PATH(m2,[1,4,1])

–	–	–	–	–	–
3	–	–	4	–	–
3	–	–	**1**	–	–
–	–	–	–	–	–
7	–	–	5	–	–
3	–	–	5	–	–

NONPATH2PATH(m2,[1,3,4,1])

–	–	–	–	–	–
3	–	3	4	–	–
–	–	–	–	–	–
–	–	–	–	–	–
7	–	4	5	–	–
3	–	4	5	–	–

NONPATH2PATH(m2,[1,3,4,2,1])

–	–	–	–	–	–
–	–	–	–	–	–
–	–	–	–	–	–
–	–	–	–	–	–
7	5	4	5	–	–
3	5	4	5	–	–

NONPATH2PATH(m2,[1,3,4,2,6,1])

–	–	–	–	–	–
–	–	–	–	–	–
–	–	–	–	–	–
–	–	–	–	–	–
7	5	**4**	5	–	4
–	–	–	–	–	–

In this implementation, when there are several edges of minimum weight at some stage, we use the first one encountered. If the tie is broken differently, a very different cycle and weight can result. For example, **[5]** applies the same algorithm to the same matrix and starting vertex, yet obtains the cycle `[1,2,3,4,5,6,1]` with a weight of `19`. This is because, **[5]** also chooses Vertex `2` as the initial vertex of the minimal-weight edge at the third iteration, but chooses Vertex `3` as the terminal vertex instead of Vertex `1`; likewise, Vertex `5` is inserted before Vertex `6` rather than before Vertex `3`.

Even for a single implementation, the output of most approximation algorithms depends on the starting point input. So it is of interest to compute and compare multiple approximations. Using an iteration-and-memory scheme as in `FIND_MIN`, it is possible to define a *DERIVE* function which performs the approximation starting with each of the vertices *and* keeps track of the best approximation. But this is overkill considering the programming headache, the severe computational penalty, and the relatively small number

of approximations one needs to compare. In fact, it is far more interesting to see the output of the much simpler function:

```
QTSTS_AND_WTS(m):=

VECTOR([QTST(m,f),TOTAL_PATH_WT(m,QTST(m,f))],f,1,DIMENSION(m))
```

For the initial matrix `m2` considered above, this function displays:

```
„ [1, 5, 3, 4, 2, 6, 1]  24 †
¦ [2, 5, 3, 4, 6, 1, 2]  21 ¦
¦ [3, 2, 6, 1, 4, 5, 3]  22 ¦
¦ [4, 5, 3, 2, 6, 1, 4]  22 ¦
¦ [5, 2, 6, 1, 4, 3, 5]  20 ¦
… [6, 5, 3, 4, 2, 1, 6]  19 ‡
```

Besides changing the starting vertex, the student can experiment with renumbering the vertices to see how that effects the cycle created by `QTST`. To reorder the vertices of a graph, the original matrix and a permutation are input to:

```
PERMUTE_GRAPH(m,p):=
  VECTOR(VECTOR(m↓[p↓i,p↓j],j,1,DIMENSION(m)),i,1,DIMENSION(m))
```

Since the approximation algorithm implemented above comes with a guarantee only under certain circumstances, it is reasonable that our Toolkit provide the means to verify that a given graph has the required properties. Because *DERIVE* rightly computes ∞-∞ as ?, we first provide a conversion from our representation of a graph to the more traditional form for cost matrices, namely, with the diagonal consisting of all zeroes. Note that we assume we have a complete directed graph, with no values of ∞ off the diagonal.

```
CONVERT(m):=VECTOR(VECTOR(IF(i=j,0,m↓[i,j]),j,1,DIMENSION(m)),
  i,1,DIMENSION(m))
```

Assuming the matrix has been so converted, it is then easy to define Boolean functions to test whether a graph is symmetric and satisfies the triangle inequality:

```
IS_SYMMETRIC(m):=IF(MIN(MIN(m-m`))=0,true,false)
SATISFIES_TRIANGLE_INEQUALITY(m):=IF(MIN(MIN(MIN
  (VECTOR(VECTOR(VECTOR(m↓[i,j]+m↓[j,k]-m↓[i,k],
  k,1,DIMENSION(m)),j,1,DIMENSION(m)),
  i,1,DIMENSION(m)))))>=0,true,false)
```

Pedagogical Issues

It is possible to "elicit" some of the TSP Toolkit functions from students in several ways. The instructor could ask for *ideas* on how to get *DERIVE* to perform certain tasks, then show students how their ideas would be coded. Alternatively, the instructor could provide a key idea and have students translate it into a function. A few of the Toolkit functions are simple enough that students could write the code for them just from their performance specifications.

But for the most part, the functions defined above are trickier than what the average student should be expected to write. While pushing *DERIVE* into territory for which it was not designed may appeal to some students, it should not be a required activity. Clever programming in *DERIVE* is neither an important life skill nor a route to deeper understanding of TSP and the broader issues associated with it.

The learning comes from hands-on experimentation. The Toolkit is meant to facilitate that experimentation. Exercises involving the Toolkit should be motivated by what we want students to understand about the problem, the algorithms, and the solutions, not about the peripheral problem of programming in DERIVE to implement the algorithms.

The point is to use the tools to learn, not to learn to build the tools.

References

[1] John A. Dossey, et al., *Discrete Mathematics, Third Edition*, Addison-Wesley, 1997.

[2] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[3] Alan Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1985.

[4] James A. McHugh, *Algorithmic Graph Theory*, Prentice-Hall, 1990.

[5] Alan Tucker, *Applied Combinatorics, Third Edition*, John Wiley & Sons, 1995.