Articles

Exact Solution of Large Asymmetric Traveling Salesman Problems

DONALD L. MILLER AND JOSEPH F. PEKNY

The traveling salesman problem is one of a class of difficult problems in combinatorial optimization that is representative of a large number of important scientific and engineering problems. A survey is given of recent applications and methods for solving large problems. In addition, an algorithm for the exact solution of the asymmetric traveling salesman problem is presented along with computational results for several classes of problems. The results show that the algorithm performs remarkably well for some classes of problems, determining an optimal solution even for problems with large numbers of cities, yet for other classes, even small problems thwart determination of a provably optimal solution.

FEW MATHEMATICAL PROBLEMS HAVE COMMANDED AS MUCH attention as the traveling salesman problem (TSP). Given a list of cities $\{1, \ldots n\}$ and costs c_{ij} for traveling between all pairs of cities, the TSP involves specifying a minimum-cost tour that visits each city once and returns to the starting point. The TSP is simply stated, has practical applications, and is representative of a large class of important scientific and engineering problems that have defied complete understanding. The nature of this representation has been made precise through the theory of nondeterministic polynomial time completeness (NP-completeness) (*1, 2*). The crux of this theory states that if an efficient algorithm could be found for the TSP, then efficient algorithms could be constructed for all problems in class NP-complete. Here, the designation "efficient" has a very precise meaning. An algorithm is deemed efficient if its execution time is bounded by a polynomial function written in terms of some reasonable measure of problem size; for example, the number of bytes on a computer needed to describe a particular instance of a problem. The theory has gained almost mythical significance because, despite years of effort by many researchers, no algorithm for the TSP or any other NP-complete problem has been found to meet this standard of efficiency. This failure has led to the widely held conjecture that no such efficient algorithm will ever be found. Practitioners faced with the need to solve real problems have sought to circumvent this state of affairs by developing approximate (*3, 4*) [heuristic (*5*)] algorithms. These algorithms do not solve the problem in a rigorous sense; they seek to find acceptable, but possibly inexact (suboptimal), solutions. Another tactic for circumventing the pessimistic conjecture is to relax the demanding notion of efficiency and explore algorithms that admit the possibility of a large, essentially infinite, execution time in the search for a rigorous solution (*6*). The exact approach has the advantage that it determines a best possible (optimal) answer on completion. Furthermore, if appropriately designed, exact algorithms may be made to yield approximate solutions of a known quality on premature termination.

We investigated whether the exact solution of large NP-complete problem instances is a reasonable goal. Our results show that a well-designed algorithm executed on a powerful computer can optimally solve a particular class of instances of the TSP many hundreds of times larger than has been reported. The question of whether an optimal solution can be obtained is of more than academic interest because the ability to determine optimal or near-optimal solutions can be of considerable economic importance. In fact, the TSP algorithm reported in this article is used to generate schedules for a variety of chemical manufacturing facilities.

As a simple example, consider the sequencing of jobs $\{1, \ldots n\}$ on a facility. The cost of switching production from job i to job j (c_{ij}) is frequently asymmetric; it is easier to paint a white car and then a black one than vice versa, so $c_{ij} \neq c_{ji}$. The processing sequence that is lowest in cost may be calculated by solution of the corresponding TSP. For this and more complicated examples, the exact algorithm may fail to produce a provably optimal schedule in a reasonable time. However, a comparison of the schedules generated by the exact algorithm versus those generated by simple heuristics sometimes shows a wide discrepancy in favor of the exact algorithm. The scheduling of a no-wait flowshop provides a more sophisticated application of the TSP (*7–9*). A no-wait flowshop consists of jobs $\{1, \ldots n\}$, each of which is processed on a set of machines $\{1, \ldots m\}$; first on machine 1, then on machine 2, and so on in increasing machine number. The no-wait condition implies that a job can only be delayed before processing begins on machine 1, after which no interruption can occur. Associated with each job is an m-tuple of numbers that indicate the processing time on each machine. A schedule consists of the processing sequence on machine 1 and the length of delay to insert between the jobs in the sequence. This delay is the minimum time necessary to prevent a job from catching up with its immediate predecessor on a downstream machine. Because the sum of the processing times is fixed, finding the sequence that minimizes the sum of the delays, hence the time to process all jobs, is equivalent to a TSP (*7*). From a general point of view, the TSP is well suited for modeling the behavior of any system where the performance attribute is simply an additive function of consecutive system states.

The enormous literature on the TSP may be broadly classified

D. L. Miller, Central Research & Development Department, E. I. du Pont de Nemours and Company, Wilmington, DE 19880. J. F. Pekny, School of Chemical Engineering, Purdue University, West Lafayette, IN 47907.

according to the origin and structure of the intercity costs. A summary through 1985 is available (*10*). Instances involving symmetric cost matrices ($c_{ij} = c_{ji}$) are widely studied, with applications documented in x-ray crystallography ($n \leq 14{,}000$) (*11*), circuit board drilling ($n \leq 17{,}000$) (*12*), very large scale integrated circuit fabrication ($n \leq 1.2$ million) (*13*), circuit board assembly ($n \leq 104$) (*14*), and the study of protein conformations (*15*). Sometimes, the cities can be represented as points on a Euclidean plane. For this case, and for some other symmetric matrix applications, the Lin-Kernighan heuristic (*16*) has proven to be highly successful, often coming within a few percent of optimal as proven by the lower bounding technique of Held and Karp (*17*). Recent work by Johnson (*4*) provides a detailed comparison of Lin-Kernighan with heuristics based on simulated annealing, genetic algorithms, and neural networks. Execution times are reasonable for Lin-Kernighan, even for instances with up to 100,000 cities, and Johnson (*4, 18*) makes a strong case that it is the best available heuristic for Euclidean and other symmetric cost matrices. Padberg and Rinaldi (*19, 20*) have successfully applied an exact branch and bound algorithm to some symmetric cost matrices. Using a Cyber-205, they report optimally solving instances involving 318 cities in 3.4 hours, 532 cities in 6 hours, 1002 cities in 7.3 hours, and 2392 cities in 27.3 hours. With an IBM 3090/600 and algorithmic modifications, the 2392-city instance was solved in 2.6 hours. Routine solution of 100-city instances requires 15 to 30 min of Vax 11/780 time. Independently, but using similar techniques, Grötschel and Holland (*21*) have solved a 666-city instance based on locations of cities throughout the world in about 9 hours and a 1000-city random-distance matrix in 30 min on an IBM 3081D.

Padberg and Rinaldi (*19*) report that the Lin-Kernighan heuristic found a solution within 1.7% of optimal for the 532-city instance in the first 7% of the total execution time. The remaining 93% of the time was required to find and guarantee an optimal solution. Taken together, these results indicate that many symmetric instances have, for practical purposes, yielded to the carefully designed Lin-Kernighan heuristic. Furthermore, the exact results show that optimal solutions for some instances of the symmetric problem are within reach of a well-crafted algorithm.

The case of asymmetric intercity costs has proven considerably more problematic for heuristics and has received less attention, although Kanellakis and Papadimitriou (*22*) have proposed a Lin-Kernighan–type heuristic. We are interested in this case because asymmetric costs are typical in the scheduling of chemical processes. Other asymmetric cost applications arise in the solution of pattern allocation problems in the glass industry (*23*) and in the overhaul of gas turbine engines (*24*). In this article we summarize an exact algorithm for the asymmetric TSP (ATSP). The algorithm derives its strength from several techniques that avoid an explicit search of the large number of possible solutions, although its performance deteriorates to an exhaustive search when the assumptions inherent in these techniques fail. For one class of problem instances drawn at random from the domain of all possible instances, the algorithm obtains optimal solutions in reasonable time for many thousands of instances of sizes up to 5,000 cities and for a few instances of sizes up to 500,000 cities. For the symmetric TSP and many other classes of instances, however, the algorithm only guarantees solution optimally for less than 30 cities. How are these apparently contradictory results to be reconciled?

Superficially, solution of large random problem instances has little practical significance, but the computational results presented below show that the algorithm obtains an optimal solution for some practical instances and reasonable approximate solutions for other instances that it cannot solve to optimality. More importantly, the large random asymmetric results, in conjunction with the results of other researchers on certain symmetric instances, seem to suggest that some instances of the TSP are tractable. Favorable computational results on artificially contrived problems must be kept in perspective; certainly it is not very difficult to confound all existing exact algorithms. Although, the fact that any nontrivial large instances can be solved to optimality offers hope that many large instances of practical importance may someday be solved to provable optimality.

Development of an effective exact algorithm requires that one have intimate knowledge of the problem and that one make some assumptions about the domain of instances to be encountered. When these assumptions are violated, an algorithm performs poorly. Alternate exact algorithms may be appropriate, although there are structures for which no known algorithm is effective (*25*). Our effort builds on the fact that other researchers have been successful at exactly solving the ATSP (*10*).

Preliminaries

Formulations that concisely define the ATSP are important for the development of exact algorithms because they provide a formalism through which knowledge can be acquired and synthesized into algorithms with provable properties. The primal and dual formulations of the related assignment problem (AP) imply a few simple results that are at the heart of our algorithm.

The question of which order to visit the cities can be resolved by consideration of a collection of yes-no decisions posed in the following form: Should city j be visited immediately after city i? If the answer is yes, then city j will immediately follow city i in a tour and cost c_{ij} will be incurred. If no, then city i should be immediately followed by some city other than j. Given n cities, there are n^2 unique yes-no questions of this form. A feasible tour results from precisely n yes answers in such a way that each city is entered and exited once in a single continuous tour. A mathematical formulation of the problem makes this notion of a related collection of decisions precise. Given a directed graph, that is, $G = (V, A)$, with vertex set $V = \{1, \ldots n\}$; arc set $A = \{(i, j) \mid i, j = 1, \ldots n\}$; and cost c_{ij} associated with each arc, the ATSP may be stated as an integer program of the following form

$$\text{minimize} \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \tag{1}$$

$$\sum_{i \in V} x_{ij} = 1, j \in V \tag{2}$$

$$\sum_{j \in V} x_{ij} = 1, i \in V \tag{3}$$

for all proper subsets $S \subset V, S \neq \varnothing$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1 \tag{4}$$

$$x_{ij} \in \{0, 1\}, i, j \in V \tag{5}$$

Each vertex in G represents a city, and an arc (i, j) represents the possibility of traveling from city i to city j. A feasible solution to Eqs. 2 to 5 consists of a single directed cycle that covers (visits) all the vertices of G, that is, a Hamiltonian cycle. There are $(n - 1)!$ feasible solutions. If arc (i, j) is present in a solution, travel occurs from city i to j, $x_{ij} = 1$, and cost c_{ij} is incurred; otherwise $x_{ij} = 0$. An optimal solution is a minimum-cost Hamiltonian cycle. A cost matrix for a small example problem and an optimal tour of cost 16 are shown in Fig. 1. This example will be used to explain our algorithm.

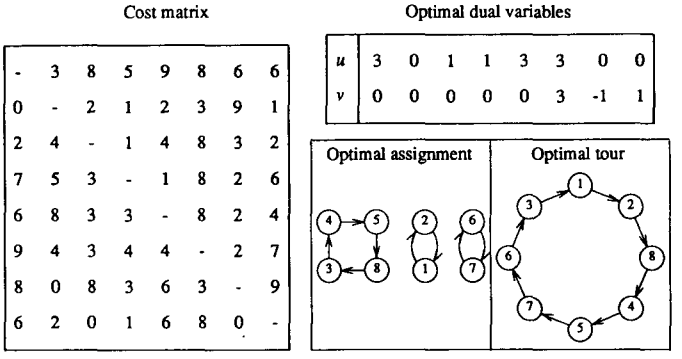

Fig. 1. Example cost matrix and dual variables for optimal AP solution. An optimal assignment (cost = 14) and optimal tour (cost = 16) are shown in the boxes.

The AP consists of Eqs. 1 to 3 and 5, and we denote the optimal value by *val(AP)*. In terms of the above discussion, a solution to the AP may be interpreted as precisely n yes answers in such a way that every city is exited and entered exactly once, but it does not necessarily imply a single continuous tour and hence is physically unrealistic. Clearly $val(AP) \leq val(ATSP)$, where *val(ATSP)* is the optimal tour cost, because the ATSP has the same constraints as the AP with the additional requirement of a physically realistic tour. In terms of graph G, each of the $n!$ possible AP solutions consists of a collection of one or more disjoint cycles that cover all the vertices (cyclic cover). An optimal AP solution of cost 14 is shown for the example in Fig. 1. The AP is known to be a natural integer program in the sense that it may be solved as a linear program by replacement of Eq. 5 with the following:

$$x_{ij} \geq 0, \; i,j \in V \tag{6}$$

Associated with the AP is an intimately related problem (dual-AP) with optimal value $val(dual\text{-}AP) = val(AP)$, whose formulation is crucial to our algorithm:

$$\text{maximize} \sum_{i \in V} u_i + \sum_{j \in V} v_j \tag{7}$$

$$c_{ij} - u_i - v_j \geq 0, \qquad i,j \in V \tag{8}$$

Figure 1 lists a set of optimal dual variables for the example. A reduced cost element $\tilde{c}_{ij}$ is defined to be $c_{ij} - u_i - v_j$. From elementary linear programming theory, the reduced costs have a straightforward interpretation: the quantity $val(AP) + \tilde{c}_{ij}$ is a lower bound on the cost of an AP solution that includes arc (i, j). For example, the reduced cost of matrix element row 1, column 3 in Fig. 1 is 5 (= 8 − 3 − 0), which means that an AP (and ATSP) solution including arc (*1, 3*) would cost at least 19 (= 14 + 5). The reduced costs gauge the impact of forcing a transition between two cities. This information is critical to eliminating a large number of unattractive solutions and concentrating effort on those solutions that can be optimal.

Overall Algorithm

Any effective exact algorithm must clearly avoid explicitly examining the vast majority of the solutions. One potential method for pruning a large number of solutions from consideration is the creation of a simpler problem with an identical set of optimal solutions that can be solved much more rapidly. To this end, consider a cost matrix (c'_{ij}) associated with ATSP′, relaxation AP′, and dual-AP′ defined as follows:

$$c'_{ij} = \begin{cases} c_{ij} \text{ if } c_{ij} \leq \lambda \\ \infty \text{ otherwise} \end{cases} \tag{9}$$

The following simple proposition defines conditions under which an optimal solution to ATSP′ is also optimal for ATSP: an optimal solution x' for ATSP′ is an optimal solution for ATSP if $val(ATSP) - val(AP) \leq \lambda + 1 - u'_i - v'_{max}$ and $\lambda + 1 - u'_i - v'_{max} \geq 0$ for all $i \in V$, where u' and v' are optimal solutions to dual-AP′ and v'_{max} is the maximum element of v'. A complete discussion of this proposition may be found in (*26*). The quantity $\lambda + 1 - u'_i - v'_{max}$ underestimates the smallest reduced cost of any discarded matrix element; hence, satisfying the proposition simply guarantees that no excluded element can lead to a better solution. We use the proposition as the basis for our ATSP algorithm as follows.

1) Choose λ.
2) Construct (c'_{ij}) according to Eq. 9 and solve ATSP′.
3) If $val(ATSP') - val(AP) \leq \lambda + 1 - u'_i - v'_{max}$ and $\lambda + 1 - u'_i - v'_{max} \geq 0$, then the optimal solution to ATSP′ found in step 2 is optimal for ATSP; otherwise, double λ and repeat steps 2 and 3.

If necessary, this procedure will increase λ until (c'_{ij}) consists of the entire original cost matrix. For the computational results reported below, we estimated a starting value for λ in step 1 by solving small problems with similar cost matrix structure. This estimation strategy was successful in that none of the trials reported below required a second iteration of steps 2 and 3. Alternatively, λ could have been estimated from the largest arc cost in a heuristic solution to the ATSP. In all computational trials reported below, the value of λ used to satisfy the conditions of the proposition was such that only a small fraction of the original matrix was retained. The algorithm embodied in steps 1 to 3 is superior to the direct solution of ATSP only to the extent that an optimal solution to ATSP′ is more rapidly obtained. In the next few sections we discuss a branch and bound approach for solving ATSP′ that is effective for some cost matrix structures.

Branch and Bound

Branch and bound is a well-known enumerative search technique for obtaining optimal solutions to the ATSP (*10*). The feasible solutions are divided into a collection of disjoint sets, lower bounds are obtained for each set, and the lower bounds are used in conjunction with upper bounds so that some sets are discarded from further consideration. The procedure is applied recursively to each of the sets. The recursion explodes exponentially whenever the lower bounds are weak with respect to the optimal solution value. Each disjoint set may be viewed as a more constrained ATSP. In our algorithm, the AP is used to determine lower bounds for the ATSP. The algorithm creates disjoint sets based on an optimal solution to the AP and determines upper bounds by splicing two or more cycles in an AP solution into a possibly suboptimal Hamiltonian cycle. Each of these operations has been specialized to take advantage of the sparsity of the ATSP′ cost matrix to accelerate performance.

We have added a feature to the basic branch and bound approach that attempts to quickly find an ATSP solution among the set of optimal solutions to the AP. This feature is denoted Hamiltonian cycle problem reduction because the goal is to reduce the solution of the ATSP to the solution of a Hamiltonian cycle problem on a suitably defined graph. Given an unweighted graph, the Hamiltonian cycle problem requires finding a single cycle that visits all vertices once. We define the admissible graph to be $\overline{G} = (V, \overline{A})$, where V is the vertex set given above, $\overline{A} = \{(i, j) \mid c_{ij} - u_i^* - v_j^* = 0\}$, and u^* and v^* are optimal AP dual variables. In an intuitive sense, the admissible graph contains the most attractive travel

possibilities. In fact, a Hamiltonian cycle on $\overline{G}$ is an optimal solution to the ATSP (*27*). If $\overline{G}$ does not have a Hamiltonian cycle, then the AP lower bound may be increased by the minimum nonzero reduced cost element with respect to the optimal dual solution (*27*). Within the context of branch and bound, the reduction procedure prunes sets of feasible solutions from further consideration whenever a Hamiltonian cycle is found or if the increase in the lower bound strength is sufficiently large. For the computational results reported below, the reduction procedure frequently enabled pruning. In many cases, particularly for large problems, we would have been unable to determine optimal ATSP solutions without the reduction procedure.

In order to determine how close $\overline{G}$ is to a true Hamiltonian cycle, we use an exact algorithm (*28*). It is analogous to the ATSP algorithm except that the AP relaxation is replaced by an unweighted bipartite matching relaxation. The virtue of testing the Hamiltonicity of the admissible graph is that the unweighted bipartite matching problem is a strong relaxation for the admissible graphs we encountered, and algorithms for solving the unweighted bipartite matching problem are very efficient.

We now summarize the complete branch and bound approach used in step 2 of the previous section to obtain the results reported below. Let Q be a set of problems related to the ATSP to be solved and designate *UB* (upper bound) the best known solution for this problem. Denote *val(z)* to be the cost of feasible solution z and let *L(ATSP)* be the value of the AP lower bound of ATSP.

1) Initialization. Place ATSP in Q. Set *UB* to $\varnothing$ (null solution). By definition $val(\varnothing) = \infty$.

2) Termination test. If Q is empty, terminate; *UB* is an optimal solution with value *val(UB)*.

3) Selection. Remove a problem $\hat{P}$ with the smallest lower bound from Q. For purposes of selection, problem $\hat{P}$ inherits the lower bound of its parent; see step 7.

4) Lower bounding. If $L(\hat{P})$ is not less than *val(UB)*, discard $\hat{P}$ and repeat from step 2. The optimal AP solution, z, may be a Hamiltonian cycle. In this case, *UB* is replaced by z.

5) Upper bounding. Determine a feasible solution z to problem $\hat{P}$. If *val(z)* is less than *val(UB)*, replace *UB* with z.

6) Reduction. Construct an admissible graph $\overline{G}(\hat{P})$ from the optimal dual variables computed in step 4. If $\overline{G}(\hat{P})$ has a Hamiltonian cycle, z, replace *UB* by z; otherwise, add the minimum reduced cost to $L(\hat{P})$ to obtain a higher lower bound for $\hat{P}$. Discard $\hat{P}$ if the new lower bound is not less than *val(UB)*.

7) Branching. Create new problems $\hat{P}_1, \hat{P}_2, \ldots, \hat{P}_k$ from problem $\hat{P}$ so that an optimal solution to one of the new problems is also optimal for $\hat{P}$. Place each of the newly created problems into Q. Discard problem $\hat{P}$ and repeat from step 2.

The branch and bound procedure of steps 1 to 7 may be executed on a parallel computer so that execution time is decreased (*26, 29*). The result of steps 1 to 7 may be viewed as a search tree. Each of the vertices of the tree uniquely represents one of the problems $\hat{P}$ removed from Q, and the edges of the tree connect problems that are directly related through step 7. The tree contains edge $(\hat{P}, \hat{P}_j)$ if problem $\hat{P}_j$ was directly created from problem $\hat{P}$. The root of the tree represents the original problem. Figure 2 illustrates application of the algorithm on the example of Fig. 1 ($\lambda = \infty$). Although there are 7! (5040) feasible tours, the algorithm determines an optimal solution using five search tree vertices.

Lower Bounding

We use a shortest augmenting path algorithm (*30*) to solve the first AP encountered in the branch and bound procedure. The key component of the algorithm is a modification of E. Dijkstra's well-known labeling procedure for finding the shortest path between two vertices in a graph. The algorithm has worst-case complexity of $O(n^3)$, but in practice is almost always faster. Subsequent APs are solved with a parametric version of the algorithm that uses the parent AP solution as a starting point to reduce the worst-case complexity to $O(n^2)$. The AP algorithm contains several features essential for the solution of large instances of a problem. In particular, the temporary column labels (*30*) are managed by use of d-heaps (*31*); when combined with a sparse cost matrix, this greatly enhances performance. Also, the parametric version of the algorithm terminates prematurely when the value of the parent lower bound plus the minimum temporary column label is not strictly less than the current upper bound. For this reason, a strong upper bound early in the calculation eliminates many vertices in the search tree without complete solution of the associated AP. A complete discussion of AP solution in the context of the ATSP algorithm is given by Pekny and Miller (*26*).

Upper Bounding

A patching procedure (*32*), tailored to exploit cost matrix sparsity, is used as an upper bounding technique. The patching procedure tries to splice together multiple cycles of an AP solution to form a possible suboptimal Hamiltonian cycle. Any improvement heuristic may be applied to the result of the patching algorithm. In some cases, we apply interchange heuristics (*4*) to the result of patching to improve heuristic solutions. An upper bound is important in the sense that no search tree vertices may be eliminated until an upper bound is established. When the ATSP algorithm is prematurely terminated, the quality of the upper bounds provided by the patching procedure may be estimated by use of search tree lower bounds. This feature is important for industrial applications that require a feasible solution of known quality in a predictable amount of time.

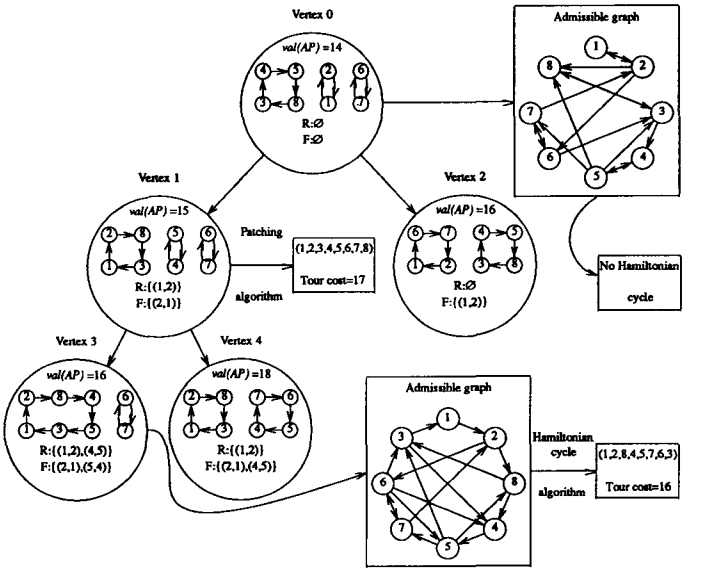

Fig. 2. Search tree for solution of example problem in Fig. 1. Each vertex illustrates an optimal assignment and the arc sets "R" and "F" denote required and forbidden arcs, respectively. The cost of an optimal assignment at the root vertex is 14 and the corresponding admissible graph was found to be non-Hamiltonian. At vertex 1, the optimal assignment of cost 15 was patched to form a tour at cost 17. Vertex 3 has an optimal assignment cost of 16 and the admissible graph was found to contain a tour. This tour is optimal because vertices 2 and 4 have optimal assignment costs ≥ 16.

Applying the patching procedure at every vertex of the search tree expends unjustified computational effort. Instead, the patching procedure is frequently applied to the AP solutions near the beginning of the calculation and less frequently after an upper bound has been established. Incorporation of the patching procedure as an upper bounding technique guarantees that the ATSP algorithm produces an answer of known quality in a worst case of $O(n^3)$ steps (*33*).

Branching Rules

We utilized a well-known branching rule (*10*) that uses the concept of required and forbidden arc sets that are associated with each vertex of the search tree. The required arc set contains those arcs that must appear in an AP solution and the forbidden arc set contains those that must not appear. A free arc is neither required nor forbidden. The branching rule operates on the cycle with the smallest number of free arcs, creating a new search tree vertex (child) for each free arc. At each child vertex, the associated free arc is converted to a forbidden arc. Required arcs are used to force the sets of feasible solutions among these children to be disjoint so that there can be no duplicate search tree vertices. As an example of the branching rule, consider a cycle (i_1, i_2, i_3), which is used to create three children, from an AP solution containing all free arcs. In the first child, arc (i_1, i_2) is forbidden. In the second child, arc (i_1, i_2) is required and arc (i_1, i_3) is forbidden. In the third child, arcs (i_1, i_2) and (i_2, i_3) are required and arc (i_3, i_1) is forbidden. For each child, the additional required and forbidden arcs augment a copy of the parent's arc sets. The search tree shown provides an illustration of the branching rule throughout the solution of the example (Fig. 2). The alternative branching rule of (*34*) has been found to be effective on the instances arising from no-wait flowshop and product wheel scheduling, although on other classes of instances this branching rule impairs performance.

Computational Results

Because the TSP is an NP-complete problem, all known exact algorithms have worst-case execution time that scales exponentially in the number of cities. Despite this worst-case scenario, exact algorithms for the TSP can deliver remarkable performance on certain problem structures. This performance depends on exploiting the structure of the cost matrix. The best known exact algorithms for symmetric matrices do not work with asymmetric matrices. Similarly, our algorithm, designed for asymmetric cost matrices, can optimally solve only small instances with symmetric matrices and other structures where AP bounds are weak.

We tested our algorithm on seven cost matrix structures: (i) matrix elements drawn from a uniform distribution of integers in the range $[0, n]$; (ii) matrix elements drawn from a uniform distribution of integers in the range $[0, n]$ and satisfying the triangle inequality; (iii) matrix elements drawn from a uniform distribution of integers in the range $[0, i \times j]$ where i and j are the row and column index of the element; (iv) matrices designed to confound local search heuristics (*35*); (v) matrices generated from Euclidean TSPs in which the cities are randomly placed on a unit square; (vi) structured asymmetric matrices that are difficult to solve to provable optimality; and (vii) matrices derived from chemical industry applications. We chose the variable cost range $[0, n]$ for the first two test classes because randomly generated problems with fixed cost range tend to become easier as problem size grows (*27*). The results of computational experiments on each of the first five cost matrix structures are shown in Table 1. Results for the remaining two classes are summarized below.

Random asymmetric problems. The algorithm performs well on large randomly generated problems. For n = 1000, 2000, 3000, 4000, and 5000, we solved 1000 problem instances at each size. Average search tree exploration times ranged from 4 s at n = 1000 to 38 s at n = 5000 on a workstation. The standard deviations for these trials were 3.03, 5.63, 10.7, 16.0, and 21.1 s for n = 1000, . . . , 5000, respectively. We solved larger problems using Cray YMP and Cray 2 supercomputers. Search tree exploration times ranged from 13 min for n = 50,000 to 3.5 hours for n = 500,000. There are two explanations for this performance: the strength of the AP lower bound and the exact algorithm for the directed Hamiltonian cycle problem. The AP bound is very good even on small problems (n = 1000), and the bound improves with problem size (Table 1). Frequently, *val(AP)* was identical to *val(ATSP),* which allowed the Hamiltonian cycle reduction procedure to find an optimal tour using the root vertex admissible graph. In these cases, all implicit enumeration was confined to the Hamiltonian cycle algorithm and the full branch and bound algorithm was not needed. The largest computational result is significant from a linear programming standpoint since it required the solution of an AP with 1×10^6 constraints and 2.5×10^{11} variables. The performance of the AP algorithm, unlike that of the ATSP algorithm, is relatively insensitive to cost matrix structure, so that large APs are routinely solvable.

Even for cases that required exploration of more than one search tree vertex, for example, the 30,000- and 100,000-city problems, the Hamiltonian cycle reduction procedure is crucial to finding an optimal solution. When an AP solution has the same cost as the optimal tour, there are usually many alternate optimal solutions to the AP. One, or perhaps a few, of these imply Hamiltonian cycles, but the majority do not. Without the Hamiltonian cycle reduction procedure, the basic branch and bound algorithm must be used to find one of the alternate optimal assignments that imply a Hamiltonian cycle. Because the AP algorithm has no way to preferentially select solutions that imply Hamiltonian cycles, a large number of search tree vertices of the same lower bound value can be explored before the discovery of an optimal tour. The Hamiltonian cycle algorithm quickly does the same enumeration by using a bipartite marching algorithm on the admissible graph.

So that the impact of the cost range could be tested, instances were also solved that had cost matrix elements drawn from $[0, 0.2 \times n]$ and $[0, 5 \times n]$ for up to 5000 cities. Results for the smaller cost range are similar to those shown in Table 1, whereas, for the larger cost range, solution times are about five times those listed. The instances of cost range $[0, 5 \times n]$ are more difficult because the root-node admissible graphs rarely have a Hamiltonian cycle necessitating AP-based branch and bound. Instances of larger cost range are about as difficult as a range of $[0, 5 \times n]$.

Random asymmetric cost matrices with triangle inequality. Table 1 shows the performance of the algorithm on random asymmetric cost matrices that satisfy the triangle inequality. We generated random matrices with each element drawn from a uniform distribution of integers in the range $[0, n]$ and then used a closure algorithm to enforce the triangle inequality (*36*). The $O(n^3)$ closure algorithm limited our testing to small problems, n = 100 to 500. Search tree exploration times are comparable to the times on random matrices not satisfying the triangle inequality and take on the average 1.5 to 2 times longer. The enforcement of the triangle inequality decreases the value of many cost matrix elements, making the problem more amenable to solution by the Hamiltonian cycle reduction procedure. This tends to decrease the number of vertices in the search tree. However, the admissible graph becomes more dense, and this adds

to the solution time.

Instances with c_{ij} drawn from [0, i × j]. Cost matrices with elements drawn from a uniform distribution of integers in the range [0, $i \times j$] are known to be more difficult for many AP algorithms. The difficulty stems from concentration of small costs in the upper left portion of the matrix. This cost matrix structure is more difficult for the ATSP algorithm. Instances of size $n = 1000$ take about six times as long to solve as random problems of the same size; for $n = 5000$ the factor increases to nearly 20. The largest instances of this structure are solved in less than 4 hours with a workstation (Table 1).

Directed diamond instances. Papadimitriou and Steiglitz (*35*) have proposed a class of TSP instances that are pathological for many heuristics. This class has small identical subgraphs (directed diamonds with six vertices) that are interconnected to construct weighted graphs with the following properties: (i) The optimal tour cost is 0. (ii) There are $(n/6)!$ next-best tours of arbitrarily high cost. (iii) None of the edges in the optimal tour are to be found in any of the next-best tours. Because the optimal tour cost is 0, the Hamiltonian cycle reduction procedure finds the optimal tour at the root vertex in the search tree (Table 1). Analysis of the data reveals that it scales as slightly worse then $O(n^3)$. At first glance, such performance on a highly structured class of difficult problem instances may seem surprising. However, the structure that renders these problems pathological for tour improvement heuristics does not necessarily have the same effect on other algorithms. In this case, the exact Hamiltonian cycle algorithm is able to identify an optimal tour in reasonable time. Although there are many problem structures that defeat the Hamiltonian cycle algorithm as well, these may or may not defeat tour improvement heuristics. A problem class that appears difficult with respect to one algorithm may be well solved by another. Thus a repertoire of algorithms seems to be essential.

Euclidean problems. Our algorithm was used to solve Euclidean problems corresponding to points randomly placed on a unit square (Table 1). As expected (*10*), the AP lower bound is poor and the algorithm is able to solve only small problems to optimality. The unsuitability of AP-based lower bounds for symmetric and Euclidean problems is well known, but we include these results for the sake of completeness, and to demonstrate the behavior of the algorithm when the underlying assumptions are violated. However, inability to find provably optimal solutions in a reasonable time does not mean inability to find good approximate solutions. In fact, for the 532-city instance solved to optimality by Padberg and Rinaldi in 6 hours on a Cyber-205 supercomputer (*19*), the ATSP algorithm finds a solution within 8.2% of optimal in 18 s and within 6.8% of optimal in 130 s on a Sun 4/330 workstation. The exact algorithm is considered to have failed because the lower bound is only approximately 80% of optimal after a very long time, but, from a

Table 1. Performance of algorithm for several classes of problems. "Instances" refers to the number of trials at each problem size. "Search tree vertices" indicates the number of vertices explored in the search tree. "Solution time" is the time required to explore the search tree, and "matrix conversion" is the time required to obtain the sparse cost matrix for the indicated λ. This time is an estimate because we coded matrix generation and conversion as a seamless operation because of memory limitations. Where there is no entry, time was negligible. Total algorithm execution time is the sum of matrix conversion and solution times.

n	Instances	λ	Search tree vertices	$\nu(AP)/\nu(ATSP)$	Solution time (s)	Matrix conversion (s)	Machine
			Random asymmetric problems, $c_{ij} \in [0, n]$				
1,000	1,000	40	10.3	0.999110	4.09	0.64	Sun4/330
2,000	1,000	40	8.14	0.999587	9.64	2.1	Sun4/330
3,000	1,000	40	7.47	0.999738	16.7	4.6	Sun4/330
4,000	1,000	40	6.92	0.999807	25.9	11	Sun4/330
5,000	1,000	40	7.97	0.999851	38.1	17	Sun4/330
30,000	1	40	213	0.999943	1122	13	CrayYMP
50,000	1	40	1	1	789	25	CrayYMP
100,000	1	50	10	0.999991	1847	70	CrayYMP
150,000	1	50	1	1	3100	117	CrayYMP
200,000	1	60	1	1	6276	216	CrayYMP
500,000	1	40	1	1	12,623	1107	Cray2
			Random asymmetric matrices with $c_{ij} \in [0, n]$ *with triangle inequality enforced*				
100	3	20	3.46	0.990741	0.423	0.01	Sun4/330
200	3	20	2.00	0.997138	6.60	0.03	Sun4/330
300	3	20	1.33	0.998978	2.39	0.06	Sun4/330
400	3	20	1	1	3.11	0.1	Sun4/330
500	3	20	1	1	5.40	0.16	Sun4/330
			Difficult matrices, $c_{ij} \in [0, i \times j]$				
1,000	3	5000	1647	0.998769	81.4	0.64	Sun4/330
5,000	3	50,000	2448	0.999623	723	17	Sun4/330
10,000	3	63,000	17,242	0.999925	4138	68	Sun4/330
20,000	3	100,000	32,713	0.999972	12,329	272	Sun4/330
			Diamond graphs				
60	1	1	1	1	0.13		Sun4/330
180	1	1	1	1	1.18		Sun4/330
300	1	1	1	1	5.83		Sun4/330
420	1	1	1	1	17.7		Sun4/330
600	1	1	1	1	65.6		Sun4/330
900	1	1	1	1	301.5		Sun4/330
1,200	1	1	1	1	899.7		Sun4/330
1,500	1	1	1	1	2176.8		Sun4/330
			Euclidean matrices from unit square				
10	1	1.0	65	0.86	0.5		Sun4/330
15	1	1.0	211	0.77	1.36		Sun4/330
20	1	1.0	5041	0.74	29.62		Sun4/330

practical perspective, the heuristic solutions reported by the algorithm are reasonable even very early in the calculation. This is not to suggest that the ATSP algorithm should be used as a heuristic on symmetric matrices; certainly there are better heuristics. However, exact algorithms that fail to find a provably optimal solution can still deliver reasonable performance.

Nearly symmetric and clustered asymmetric problems. Clustering is said to occur within a cost matrix when cities can be grouped so that intragroup travel costs are low relative to intergroup travel costs. Symmetric cost matrices tend to have clusters of size 2 because small cost elements have an equally small twin. An AP lower bound is poor because the larger intergroup costs do not contribute and many cycles of size 2 appear in the optimal AP solution. This poor lower bound translates into poor ATSP algorithm performance. Matrices with a small but significant asymmetry also tend to form clusters of size 2. The presence of even minor cost matrix asymmetry precludes the use of algorithms designed for symmetric problems. Even with a high degree of asymmetry, strong clustering can occur. In fact, the AP bounds can be arbitrarily weak on such problems, with concomitant poor ATSP algorithm performance. Unfortunately, these structures occur frequently in practice; products in a chemical plant may belong to families, and transitions within a family may be much cheaper than those that cross family boundaries. For this reason, our algorithm can require long times to solve real world problems even of small size. Some real world problems involving only a few dozen cities cannot be solved to proven optimality by our algorithm. Fischetti and Toth (*25*) have proposed a technique that has potential for improving performance on nearly symmetric and clustered asymmetric problems through the combination of many different lower bounding techniques, but they still report difficulties.

Product wheel and no-wait flowshop scheduling applications. Many multiproduct chemical plants operate on a product wheel; that is, the plant cycles through the products in a fixed sequence. The determination of an optimal sequence is a TSP in which the products represent cities and the cost matrix is given by the cost of product transitions. These problems frequently display clustering and can be difficult to solve. We obtained ten example problems ranging in size from 14 to 60 products from manufacturing facilities. The algorithm solved problems of size 14, 17, 17, 17, 19, 23, and 24 products to optimality in less than 10 s on a Sun 4/330. The three largest problems, of size 35, 43, and 60 products, were all either nearly symmetric or highly clustered and could not be solved to optimality.

The scheduling of a no-wait flowshop provided a further test on an industrially derived structure. Schedules were generated for three problems of size 60, 80, and 100 jobs using data representative of a chemical manufacturing facility. For each problem, jobs can be grouped into one of six families. The facility has four reactors, and each job must be processed on the reactors in the same order and for a fixed time in each reactor. The processing times in each reactor vary markedly (factors of 10) from family to family but are similar (~10% variance) within a family. Like the product wheel problems, these flowshop problems display clustering, but to a weaker extent. All three test problems were solved to optimality in less than 20 s with a Sun 4/330.

Conclusions

We have reviewed recent computational successes for the symmetric TSP, summarized a new algorithm for exactly solving the ATSP, and presented computational results that demonstrate a wide range of behavior. How is this data and that of other researchers to be understood within the theory of NP-completeness?

The theory of NP-completeness deals only with worst-case algorithmic performance, while the data reflect actual performance on particular cost matrices. Our experiments with both contrived and practical instances have shown that, with respect to a given exact algorithm, instances can display varying degrees of tractability. At least some large instances are tractable and the TSP is not necessarily impossible to optimally solve. This is of little consolation if one is faced with an actual instance on which no known algorithm is effective, but NP-completeness provides no reason that a given instance cannot be solved, only that a difficult instance can be produced to defeat a given algorithm. At present, theoretical understanding cannot easily predict actual performance on any given instance, which is crucial in deciding whether an algorithm is computationally useful, or, for practical purposes, whether a problem is tractable or intractable. The theory is incomplete and, as is often the case in the physical sciences, a more complete theory may not appear until substantial experimental work is done in this area. As a result, the investigation of the nature of NP-completeness should be considered at least partly an experimental science. Work on the TSP indicates that a prerequisite for success is an intimate knowledge of both the problem and the types of instances to be encountered. The lack of reliable guarantees of performance should not discourage research on exact algorithms because the potential practical and scientific benefits of success for many NP-complete problems are substantial.

REFERENCES AND NOTES

1. NP-completeness: consider the multiplication of two $n \times n$ matrices. The straightforward multiplication algorithm requires n^3 multiplications and $n^3 - n^2$ additions. The multiplication of two matrices is said to be solved through an efficient algorithm because no more than a number of steps bounded by a polynomial in some measure of problem size (number of rows and columns) is required regardless of the actual numbers contained in the matrices. No such efficient algorithm for solving the TSP has been conceived that is guaranteed to produce a minimum cost solution for every combination of intercity costs.
2. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, New York, 1979).
3. S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi, *Science* **220**, 671 (1983).
4. D. S. Johnson, *Proceedings of the 17th Colloquium on Automata, Languages and Programming* (Springer-Verlag, New York, 1990), p. 446.
5. A heuristic is a solution strategy that produces an answer without any formal guarantee as to the quality. A very simple heuristic for the TSP would be to begin at the starting city and travel to the nearest city, then proceed from this city to the nearest city not already visited, and continue this nearest neighbor strategy until all cities had been visited once and only once. In the final step of the heuristic, the last city in the tour is connected to the starting city. This heuristic can produce an arbitrarily bad answer because the initial travel choices can result in high-cost transitions later in the tour.
6. A commonly held misperception is that an NP-complete problem cannot be solved to optimality in a reasonable amount of time. In fact, the NP-completeness of a problem simply says that hard instances can be produced that will confound any known exact algorithm. An optimistic interpretation of NP-completeness suggests that for any given problem instance, an effective exact algorithm could be developed. Under this optimistic interpretation, many different exact algorithms may have to be developed for instances encountered in practice; however, the conjecture that an effective exact algorithm can be developed for any instance is not inconsistent with complexity theory. Keep in mind that this conjecture says nothing about the expense of developing exact algorithms. In fact, developing exact algorithms for certain instances may entail great effort.
7. D. A. Wismer, *Oper. Res.* **20**, 689 (1972).
8. J. N. D. Gupta, *Naval Res. Logist. Q.* **23**, 235 (1976).
9. C. H. Papadimitriou and P. C. Kanellakis, *J. Assoc. Comput. Mach.* **27**, 533 (1980).
10. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization* (Wiley, New York, 1985).
11. R. G. Bland and D. F. Shallcross, *Oper. Res. Lett.* **8**, 125 (1989).
12. J. D. Litke, *Commun. ACM* **27**, 1227 (1984).
13. B. Korte, paper presented at the 13th International Mathematical Programming Symposium, Tokyo, 1988.
14. D. Chan and D. Mercier, *Int. J. Prod. Res.* **27**, 1837 (1989).
15. H. Bohr and S. Brunak, *Complex Syst.* **3**, 9 (1989).
16. S. Lin and B. W. Kernighan, *Oper. Res.* **21**, 498 (1973).
17. M. Held and R. M. Karp, *ibid.* **18**, 1138 (1970).

18. D. Johnson, *Nature* **335**, 155 (1987).
19. M. Padberg and G. Rinaldi, *Oper. Res. Lett.* **6**, 1 (1987).
20. _____, Report R. 247 (Instituto di Analisi Dei Sistimi ed Informatica del CNR, Rome, 1988).
21. M. Grötschel and O. Holland, Report No. 73 (Institut für Mathematik, Universität Augsburg, Germany, 1988).
22. P. C. Kanellakis and C. H. Papadimitriou, *Oper. Res.* **28**, 1086 (1980).
23. O. B. G. Madsen, *J. Oper. Res. Soc.* **39**, 249 (1988).
24. R. D. Plante, T. J. Lowe, R. Chandrasekaran, *Oper. Res.* **35**, 772 (1987).
25. M. Fischetti and P. Toth, "An additive bounding procedure for the asymmetric traveling salesman problem" (DEIS, University of Bologna, Italy, 1989).
26. J. F. Pekny and D. L. Miller, *Math. Program.*, in press.
27. _____, D. Stodolsky, *Oper. Res. Lett.*, in press.
28. D. Stodolsky, J. F. Pekny, D. L. Miller, *Eng. Des. Res. Cent. Rep. 05-25-88* (Carnegie Mellon Univ., Pittsburgh, PA 1988).
29. D. L. Miller and J. F. Pekny, *Oper. Res. Lett.* **8**, 129 (1989).
30. E. Balas, D. L. Miller, J. F. Pekny, P. Toth, *J. Assoc. Comput. Mach.*, in press.
31. R. E. Tarjan, *Data Structures and Network Algorithms* (Society for Industrial and Applied Mathematics, Philadelphia, 1983).
32. R. M. Karp, *SIAM J. Comput.* **8**, 561 (1979).
33. Solution of the root vertex AP followed by patching with $\lambda = \infty$.
34. M. Bellmore and J. C. Malone, *Oper. Res.* **19**, 278 (1971).
35. C. H. Papadimitriou and K. Steiglitz, *ibid.* **26**, 434 (1978).
36. $c_{ij} \leq c_{ik} + c_{kj}$ for all $i, j, k \in V$.
37. We thank Cray Research Inc. for computer access and technical support.

The *myoD* Gene Family: Nodal Point During Specification of the Muscle Cell Lineage

HAROLD WEINTRAUB, ROBERT DAVIS, STEPHEN TAPSCOTT, MATTHEW THAYER, MICHAEL KRAUSE, ROBERT BENEZRA, T. KEITH BLACKWELL, DAVID TURNER, RALPH RUPP, STANLEY HOLLENBERG, YUAN ZHUANG, ANDREW LASSAR

The *myoD* gene converts many differentiated cell types into muscle. MyoD is a member of the basic-helix-loop-helix family of proteins; this 68–amino acid domain in MyoD is necessary and sufficient for myogenesis. MyoD binds cooperatively to muscle-specific enhancers and activates transcription. The helix-loop-helix motif is responsible for dimerization, and, depending on its dimerization partner, MyoD activity can be controlled. MyoD senses and integrates many facets of cell state. MyoD is expressed only in skeletal muscle and its precursors; in nonmuscle cells *myoD* is repressed by specific genes. MyoD activates its own transcription; this may stabilize commitment to myogenesis.

THE *myoD* GENE IS CAPABLE OF ACTIVATING PREVIOUSLY silent muscle-specific genes when introduced into a large variety of differentiated cell types with a viral long terminal repeat (LTR) used to promote constitutive transcription (*1–4*). In certain cell types, the entire program for muscle differentiation seems to be activated (*3*). The range of cell types converted to muscle by *myoD* includes a number of fibroblast cell lines, adipocytes, melanoma cells, a hepatoma cell line, neuroblastoma cells, osteosarcoma cells and P19 teratocarcinoma cells, as well as primary cultures of chondrocytes, smooth muscle, retinal pigment, fibroblasts, and brain cells. MyoD is expressed only in skeletal muscle. Cardiac and smooth muscle, which express many of the same muscle-specific structural genes as skeletal muscle, do not express MyoD (*1, 5*). The MyoD protein seems to activate myogenesis by directly binding to the control regions of muscle-specific genes (*6*). On the basis of these properties, we refer to *myoD* as a "master regulatory gene." Implicit in this shorthand is the fact that other factors must be responsible for the initial activation of *myoD*, and that the activity of the MyoD protein, itself, could be and is regulated. We view *myoD* as a "nodal point" (*2*) in the flow of myogenic information from the early embryo to the mature myofiber and, as discussed below, consider all members of the *myoD* family (*myogenin*, *myf-5*, *mrf-4-herculin*) to perform more or less the same "function," because assays to date have not dramatically distinguished one from another. In contrast to segmentation genes, homeotic genes, lineage genes, and the like, studied in *Drosophila* or *Caenorabditis elegans*, *myoD* seems to affect the identity of a single cell type, not constellations of many types of cells.

We describe here the structure of MyoD; how it activates the myogenic program; and how *myoD*, itself, is transcriptionally and posttranscriptionally regulated during development. The story can appear extremely simple; however, not unexpectedly, new findings bring new paradoxes and complexities, which we will also explore.

The authors are at the Howard Hughes Medical Institute, Fred Hutchinson Cancer Research Center, 1124 Columbia Street, Seattle, WA 98104.

A Single Genetic Function Can Activate Myogenesis

The notion that myogenesis can be controlled by a single master regulatory gene originates in a series of experiments from Holtzer's lab (*7*). When the thymidine analog bromodeoxyuridine (BrdU) was incorporated into DNA, myogenesis in culture was inhibited, but, with continued growth in the absence of BrdU, the reappearance of muscle was very rapid. This suggested that incorporation of BrdU into DNA inhibited one or a few targets on one or a few chromosomes and, in turn, these targets were capable of reactivating the myogenic program when subsequently segregated to daughter cells after growth and division in the absence of BrdU (*8*). We now know that BrdU turns off transcription of *myoD*, although we do not know how (*9*).

The idea of a pivotal control gene gained momentum from the work of Taylor and Jones (*10*), who showed that brief treatment of C3H $10T\frac{1}{2}$ fibroblastic cells with 5-azacytidine induced the formation of large numbers (25 to 50%) of myogenic colonies, as well as fewer numbers of chondrogenic and adipogenic colonies. The high frequency of myogenic colonies suggested that 5-azacytidine, which