The Traveling Salesman Problem for Cubic Graphs

David Eppstein

School of Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
eppstein@ics.uci.edu

Abstract. We show how to find a Hamiltonian cycle in a graph of degree at most three with n vertices, in time $\mathcal{O}(2^{n/3}) \approx 1.25992^n$ and linear space. Our algorithm can find the minimum weight Hamiltonian cycle (traveling salesman problem), in the same time bound, and count the number of Hamiltonian cycles in time $\mathcal{O}(2^{3n/8} n^{\mathcal{O}(1)}) \approx 1.29684^n$. We also solve the traveling salesman problem in graphs of degree at most four, by a randomized (Monte Carlo) algorithm with runtime $\mathcal{O}((27/4)^{n/3}) \approx 1.88988^n$. Our algorithms allow the input to specify a set of forced edges which must be part of any generated cycle.

1 Introduction

The traveling salesman problem and the closely related Hamiltonian cycle problem are two of the most fundamental of NP-complete graph problems [5]. However, despite much progress on exponential-time solutions to other graph problems such as chromatic number [2, 3, 6] or maximal independent sets [1, 7, 8], the only worst-case bound known for finding Hamiltonian cycles or traveling salesman tours is that for a simple dynamic program, using time and space $\mathcal{O}(2^n n^{\mathcal{O}(1)})$, that finds Hamiltonian paths with specified endpoints for each induced subgraph of the input graph (D. S. Johnson, personal communication). Therefore, it is of interest to find special cases of the problem that, while still NP-complete, may be solved more quickly than the general problem.

In this paper, we consider one such case: the traveling salesman problem in graphs with maximum degree three. Bounded-degree maximum independent sets had previously been considered [1] but we are unaware of similar work for the traveling salesman problem. More generally, we consider the *forced traveling salesman problem* in which the input is a multigraph G and set of *forced edges* F; the output is a minimum cost Hamiltonian cycle of G, containing all edges of F. A naive branching search that repeatedly adds one edge to a growing path, choosing at each step one of two edges at the path endpoint, and backtracking when the chosen edge leads to a previous vertex, solves this problem in time $\mathcal{O}(2^n)$ and linear space; this is already an improvement over the general graph dynamic programming algorithm. We show that more sophisticated backtracking

F. Dehne, J.-R. Sack, M. Smid (Eds.): WADS 2003, LNCS 2748, pp. 307–318, 2003.

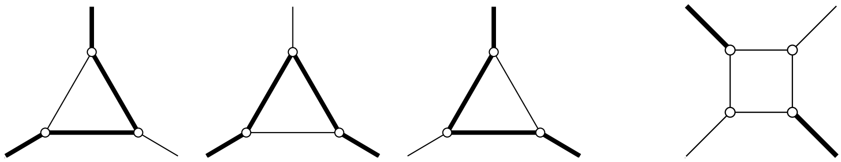

Fig. 1. Left: Case analysis of possible paths a Hamiltonian cycle can take through a triangle. Edges belonging to the Hamiltonian cycle are shown as heavier than the non-cycle edges. Right: Cycle of four unforced edges, with two forced edges adjacent to opposite cycle vertices (step 1(j)).

can solve the forced traveling salesman problem (and therefore also the traveling salesman and Hamiltonian cycle problems) for cubic graphs in time $\mathcal{O}(2^{n/3}) \approx 1.25992^n$ and linear space. We also provide a randomized reduction from degree four graphs to degree three graphs solving the traveling salesman problem in better time than the general case for those graphs. We then consider a weighted counting version of the Hamiltonian cycle problem. Let each edge of G has a weight, and let the weight of a Hamiltonian cycle to be the product of the weights of its edges. We show that the sum of the weights of all Hamiltonian cycles, in graphs with forced edges and maximum degree three, can be found in time $\mathcal{O}(2^{3n/8} n^{\mathcal{O}(1)}) \approx 1.29684^n$. If all weights are one, this sum of cycle weights is exactly the number of Hamiltonian cycles in the graph.

2 The Algorithm and Its Correctness

Our algorithm is based on a simple case-based backtracking technique. Recall that G is a graph with maximum degree 3, while F is a set of edges that must be used in our traveling salesman tour. For simplicity, we describe a version of the algorithm that returns only the *cost* of the optimal tour, or the special value *None* if there is no solution. The tour itself can be reconstructed by keeping track of which branch of the backtracking process led to the returned cost; we omit the details The steps of the algorithm are listed in Table 1. Roughly, our algorithm proceeds in the following stages. Step 1 of the algorithm reduces the size of the input without branching, after which the graph can be assumed to be cubic and triangle-free, with forced edges forming a matching. Step 2 tests for a case in which all unforced edges form disjoint 4-cycles; we can then solve the problem immediately via a minimum spanning tree algorithm. Finally (steps 3-6), we choose an edge to branch on, and divide the solution space into two subspaces, one in which the edge is forced to be in the solution and one in which it is excluded. These two subproblems are solved recursively, and it is our goal to minimize the number of times this recursive branching occurs.

All steps of the algorithm either return or reduce the input graph to one or more smaller graphs that also have maximum degree three, so the algorithm must eventually terminate. To show correctness, each step must preserve the existence and weight of the optimal traveling salesman tour. This is easy to

Table 1. Forced traveling salesman algorithm for graph G and forced edge set F.

1. Repeat the following steps until one of the steps returns or none of them applies:
   a) If G contains a vertex with degree zero or one, return *None*.
   b) If G contains a vertex with degree two, add its incident edges to F.
   c) If F consists of a Hamiltonian cycle, return the cost of this cycle.
   d) If F contains a non-Hamiltonian cycle, return *None*.
   e) If F contains three edges meeting at a vertex, return *None*.
   f) If F contains exactly two edges meeting at some vertex, remove from G that vertex and any other edge incident to it; replace the two edges by a single forced edge connecting their other two endpoints, having as its cost the sum of the costs of the two replaced edges' costs.
   g) If G contains two parallel edges, at least one of which is not in F, and G has more than two vertices, then remove from G whichever of the two edges is unforced and has larger cost.
   h) If G contains a self-loop which is not in F, and G has more than one vertex, remove the self-loop from G.
   i) If G contains a triangle xyz, then for each non-triangle edge e incident to a triangle vertex, increase the cost of e by the cost of the opposite triangle edge. Also, if the triangle edge opposite e belongs to F, add e to F. Remove from G the three triangle edges, and contract the three triangle vertices into a single supervertex.
   j) If G contains a cycle of four unforced edges, two opposite vertices of which are each incident to a forced edge outside the cycle, then add to F all non-cycle edges that are incident to a vertex of the cycle.
2. If $G \setminus F$ forms a collection of disjoint 4-cycles, perform the following steps.
   a) For each 4-cycle C_i in $G \setminus F$, let H_i consist of two opposite edges of C_i, chosen so that the cost of H_i is less than or equal to the cost of $C_i \setminus H_i$.
   b) Let $H = \cup_i H_i$. Then $F \cup H$ is a degree-two spanning subgraph of G, but may not be connected.
   c) Form a graph $G' = (V', E')$, where the vertices of V' consist of the connected components of $F \cup H$. For each set H_i that contains edges from two different components K_j and K_k, draw an edge in E' between the corresponding two vertices, with cost equal to the difference between the costs of C_i and of H_i.
   d) Compute the minimum spanning tree of (G', E').
   e) Return the sum of the costs of $F \cup H$ and of the minimum spanning tree.
3. Choose an edge yz according to the following cases:
   a) If $G \setminus F$ contains a 4-cycle, two vertices of which are adjacent to edges in F, let y be one of the other two vertices of the cycle and let yz be an edge of $G \setminus F$ that does not belong to the cycle.
   b) If there is no such 4-cycle, but F is nonempty, let xy be any edge in F and yz be an adjacent edge in $G \setminus F$.
   c) If F is empty, let yz be any edge in G.
4. Call the algorithm recursively on $G, F \cup \{yz\}$.
5. Call the algorithm recursively on $G \setminus \{yz\}, F$.
6. Return the minimum of the set of at most two numbers returned by the two recursive calls.

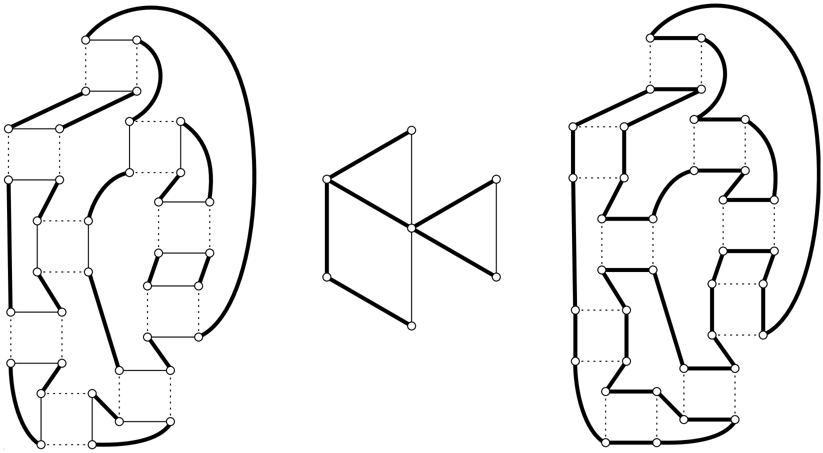

Fig. 2. Step 2 of the traveling salesman algorithm. Left: Graph with forced edges (thick lines), such that the unforced edges form disjoint 4-cycles. In each 4-cycle C_i, the pair H_i of edges with lighter weight is shown as solid, and the heavier two edges are shown dashed. Middle: Graph G', the vertices of which are the connected components of solid edges in the left figure, and the edges of which connect two components that pass through the same 4-cycle. A spanning tree of G' is shown with thick lines. Right: The tour of G corresponding to the spanning tree. The tour includes $C_i \setminus H_i$ when C_i corresponds to a spanning tree edge, and includes H_i otherwise.

verify for most cases of steps 1 and 3–6. Case 1(i) performs a so-called Δ-Y transformation on the graph; case analysis (Figure 1, left) shows that each edge of the contracted triangle participates in a Hamiltonian cycle exactly when the opposite non-triangle edge also participates. Case 1(j) concerns a 4-cycle in G, with edges in F forcing the Hamiltonian cycle to enter or exit on two opposite vertices (Figure 1, right). If a Hamiltonian cycle enters and exits a cycle in G only once, it does so on two adjacent vertices of the cycle, so the 4-cycle of this case is entered and exited twice by every Hamiltonian cycle, and the step's addition of edges to F does not change the set of solutions of the problem.

It remains to prove correctness of step 2 of the algorithm.

Lemma 1. *Suppose that G, F can not be reduced by step 1 of the algorithm described in Table 1, and that $G \setminus F$ forms a collection of disjoint 4-cycles. Then step 2 of the algorithm correctly solves the forced traveling salesman problem in polynomial time for G and F.*

Proof. Let C_i, H_i, H, and G' be as defined in step 2 of the algorithm. Figure 2(left) depicts F as the thick edges, C_i as the thin edges, and H_i and H as the thin solid edges; Figure 2(middle) depicts the corresponding graph G'.

We first show that the weight of the optimal tour T is at least as large as what the algorithm computes. The symmetric difference $T \oplus (F \cup H)$ contains edges only from the 4-cycles C_i. Analysis similar to that for substep 1(j) shows that, within each 4-cycle C_i, T must contain either the two edges in H_i or the two edges in $C_i \setminus H_i$. Therefore, $T \oplus (F \cup H)$ forms a collection of 4-cycles which is a subset of the 4-cycles in $G \setminus F$ and which corresponds to some subgraph S of G'. Further, due to the way we defined the edge weights in G', the difference between the weights of T and of $F \cup H$ is equal to the weight of S. S must be a connected spanning subgraph of G', for otherwise the vertices in some two components of $F \cup H$ would not be connected to each other in T. Since all edge weights in G' are non-negative, the weight of spanning subgraph S is at least equal to that of the minimum spanning tree of G'.

In the other direction, one can show by induction that, if T' is any spanning tree of G', such as the one shown by the thick edges in Figure 2(middle), and S' is the set of 4-cycles in G corresponding to the edges of T', then $S' \oplus (F \cup H)$ is a Hamiltonian cycle of G with weight equal to that of $F \cup H$ plus the weight of T' (Figure 2(right)). Therefore, the weight of the optimal tour T is at most equal to that of $F \cup H$ plus the weight of the minimum spanning tree of G'.

We have bounded the weight of the traveling salesman tour both above and below by the quantity computed by the algorithm, so the algorithm correctly solves the traveling salesman problem for this class of graphs. □

We summarize our results below.

Theorem 1. *The algorithm described in Table 1 always terminates, and returns the weight of the optimal traveling salesman tour of the input graph G.*

3 Implementation Details

Define a *step* of the algorithm of Table 1 to be a single execution of one of the numbered or lettered items in the algorithm description. As described, each step involves searching for some kind of configuration in the graph, and could therefore take as much as linear time. Although a linear factor is insignificant compared to the exponential time bound of our overall algorithm, it is nevertheless important (and will simplify our bounds) to reduce such factors to the extent possible. As we now show, we can maintain some simple data structures that let us avoid repeatedly searching for configurations in the graph.

Lemma 2. *The algorithm of Table 1 can be implemented in such a way that step 3, and each substep of step 1, take constant time per step.*

Proof. The key observation is that most of these steps and substeps require finding a connected pattern of $\mathcal{O}(1)$ edges in the graph. Since the graph has bounded degree, there can be at most $\mathcal{O}(n)$ matches to any such pattern. We can maintain the set of matches by removing a match from a set whenever one of the graph transformations changes one of its edges, and after each transformation searching within a constant radius of the changed portion of the graph for new matches to add to the set. In this way, finding a matching pattern is a constant time operation (simply pick the first one from the set of known matches), and updating the set of matches is also constant time per operation.

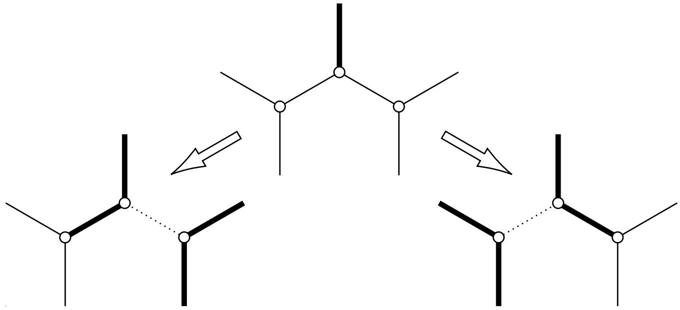

Fig. 3. Result of performing steps 2-5 with no nearby forced edge: one of edges yz and yw becomes forced (shown as thick segments), and the removal of the other edge (shown as dotted) causes two neighboring edges to become forced.

The only two steps for which this technique does not work are 1(c) and 1(d), which each involve finding a cycle of possibly unbounded size in G. However, if a long cycle of forced edges exists, step 1(e) or 1(f) must be applicable to the graph; repeated application of these steps will eventually either discover that the graph is non-Hamiltonian or reduce the cycle to a single self-loop. So we can safely replace 1(c) and 1(d) by steps that search for a one-vertex cycle in F, detect the applicability of the modified steps 1(c) and 1(d) by a finite pattern matching procedure, and use the same technique for maintaining sets of matches described above to solve this pattern matching problem in constant time per step. □

To aid in our analysis, we restrict our implementation so that, when it can choose among several applicable steps, it gives first priority to steps which immediately return (that is, steps 1(a) and 1(c–e), with the modifications to steps 1(c) and 1(d) described in the lemma above), and second priority to step 1(f). The prioritization among the remaining steps is unimportant to our analysis.

4 Analysis

By the results of the previous section, in order to compute an overall time bound for the algorithm outlined in Table 1, we need only estimate the number of steps it performs. Neglecting recursive calls that immediately return, we must count the number of iterations of steps 1(b), 1(f–h), and 3–6.

Lemma 3. *If we prioritize the steps of the algorithm as described in the previous section, the number of iterations of step 1(f) is at most $\mathcal{O}(n)$ plus a number proportional to the number of iterations of the other steps of the algorithm.*

Proof. The algorithm may perform at most $\mathcal{O}(n)$ iterations of step 1(f) prior to executing any other step. After that point, each additional forced edge can cause at most two iterations of step 1(f), merging that edge with previously existing forced edges on either side of it, and each step other than 1(f) creates at most a constant number of new forced edges. □

It remains to count the number of iterations of steps 1(b), 1(g), 1(h), and 3–6. The key idea of the analysis is to bound the number of steps by a recurrence involving a nonstandard measure of the size of a graph G: let $s(G, F) = |V(G)| - |F| - |C|$, where C denotes the set of 4-cycles of G that form connected components of $G \setminus F$. Clearly, $s \leq n$, so a bound on the time complexity of our algorithm in terms of s will lead to a similar bound in terms of n. Equivalently, we can view our analysis as involving a three-parameter recurrence in n, $|F|$, and $|C|$; in recent work [4] we showed that the asymptotic behavior of this type of multivariate recurrence can be analyzed by using weighted combinations of variables to reduce it to a univariate recurrence, similarly to our definition here of s as a combination of n, $|F|$, and $|C|$. Note that step 1(f) leaves s unchanged and the other steps do not increase it.

Lemma 4. *Let a graph G and nonempty forced edge set F be given in which neither an immediate return nor step 1(f) can be performed, and let $s(G, F)$ be as defined above. Then the algorithm of Table 1, within a constant number of steps, reduces the problem to one of the following situations:*

- *a single subproblem G', F', with $s(G', F') \leq s(G, F) - 1$, or*
- *subproblems G_1, F_1 and G_2, F_2, with $s(G_1, F_1), s(G_2, F_2) \leq s(G, F) - 3$, or*
- *subproblems G_1, F_1 and G_2, F_2, with $s(G_1, F_1) \leq s(G, F) - 2$ and $s(G_2, F_2) \leq s(G, F) - 5$.*

Proof. If step 1(b), 1(g), 1(h), or 1(j) applies, the problem is immediately reduced to a single subproblem with more forced edges, and if step 1(i) applies, the number of vertices is reduced. Step 2 provides an immediate return from the algorithm. So, we can restrict our attention to problems in which the algorithm is immediately forced to apply steps 3-6. In such problems, the input must be a simple cubic triangle-free graph, and F must form a matching in this graph, for otherwise one of the earlier steps would apply.

We now analyze cases according to the neighborhood of the edge yz chosen in step 3. To help explain the cases, we let yw denote the third edge of G incident to the same vertex as xy and yz. We also assume that no immediate return is performed within $\mathcal{O}(1)$ steps of the initial problem, for otherwise we would again have reduced the problem to a single smaller subproblem.

- In the first case, corresponding to step 3(a) of the algorithm, yz is adjacent to a 4-cycle in $G \setminus F$ which already is adjacent to two other edges of F. Adding yz to F in the recursive call in step 4 leads to a situation in which step 1(j) applies, adding the fourth adjacent edge of the cycle to F and forming a 4-cycle component of $G \setminus F$. Thus $|F|$ increases by two and $|C|$ increases by one. In step 5, yz is removed from F, following which step 1(b) adds two edges of the 4-cycle to F, step 1(f) contracts these two edges to a single edge, shrinking the 4-cycle to a triangle, and step 1(i) contracts the triangle to a single vertex, so the number of vertices in the graph is decreased by three.
- In the next case, yz is chosen by step 3(b) to be adjacent to forced edge xy, and neither yz nor yw is incident to a second edge in F. If we add yz to

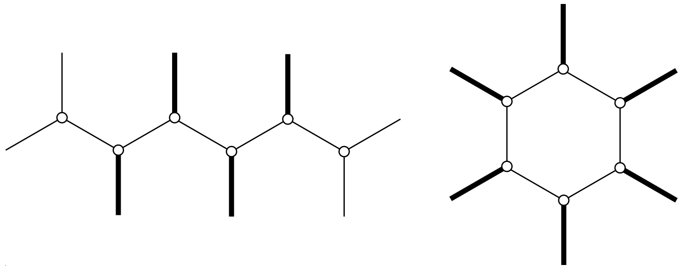

Fig. 4. Chains of two or more vertices each having two adjacent unforced edges. Left: chain terminated by vertices with three unforced edges. Right: cycle of six or more vertices with two unforced edges.

F, an application of step 1(f) removes yw, and another application of step 1(b) adds the two edges adjoining yw to F, so the number of forced edges is increased by three. The subproblem in which we remove yz from F is symmetric. This case and its two subproblems are shown in Figure 3.

– If step 3(b) chooses edge yz, and z or w is incident to a forced edge, then with y it forms part of a chain of two or more vertices, each incident to exactly two unforced edges that connect vertices in the chain. This chain may terminate at vertices with three adjacent unforced edges (Figure 4, left). If it does, a similar analysis to the previous case shows that adding yz to F or removing it from G causes alternating members of the chain to be added to F or removed from G, so that no chain edge is left unforced. In addition, when an edge at the end of the chain is removed from G, two adjacent unforced edges are added to F, so these chains generally lead to a greater reduction in size than the previous case. The smallest reduction happens when the chain consists of exactly two vertices adjacent to forced edges. In this case, one of the two subproblems is formed by adding two new forced edges at the ends of the chain, and removing one edge interior to the chain; it has $s(G_1, F_1) = s(G, F) - 2$. The other subproblem is formed by removing the two edges at the ends of the chain, and adding to F the edge in the middle of the chain and the other unforced edges adjacent to the ends of the chain. None of these other edges can coincide with each other without creating a 4-cycle that would have been treated in the first case of our analysis, so in this case there are five new forced edges and $s(G_2, F_2) = s(G, F) - 5$.

– In the remaining case, step 3(b) chooses an edge belonging to a cycle of unforced edges, each vertex of which is also incident to a forced edge (Figure 4, right). In this case, adding or removing one of the cycle edges causes a chain reaction which alternately adds and removes all cycle edges. This case only arises when the cycle length is five or more, and if it is exactly five then an inconsistency quickly arises causing both recursive calls to return within a constant number of steps. When the cycle length is six or more, both resulting subproblems end up with at least three more forced edges.

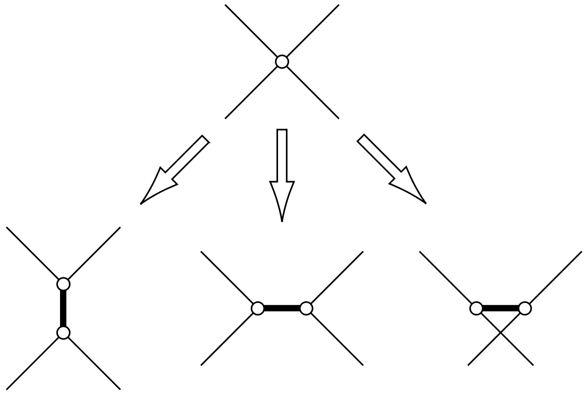

Fig. 5. Reducing degree four vertices to degree three vertices, by randomly splitting vertices and connecting the two sides by a forced edge.

Note that the analysis need not consider choices made by step 3(c) of the algorithm, as F is assumed nonempty; step 3(c) can occur only once and does not contribute to the asymptotic complexity of the algorithm. In all cases, the graph is reduced to subproblems that have sizes bounded as stated in the lemma. □

Theorem 2. *The algorithm of Table 1 solves the forced traveling salesman problem on graphs of degree three in time $\mathcal{O}(2^{n/3})$.*

Proof. The algorithm's correctness has already been discussed. By Lemmas 1, 2, 3, and 4, the time for the algorithm can be bounded within a constant factor by the solution to the recurrence

$$T(s) \leq 1 + \max\{s^{\mathcal{O}(1)}, T(s-1), 2T(s-3), T(s-2) + T(s-5)\}.$$

Standard techniques for linear recurrences give the solution as $T(s) = \mathcal{O}(2^{s/3})$. In any n-vertex cubic graph, s is at most n, so expressed in terms of n this gives a bound of $\mathcal{O}(2^{n/3})$ on the running time of the algorithm. □

5 Degree Four

It is natural to ask to what extent our algorithm can be generalized to higher vertex degrees. We provide a first step in this direction, by describing a randomized (Monte Carlo) algorithm: that is, an algorithm that may produce incorrect results with bounded probability. To describe the algorithm, let f denote the number of degree four vertices in the given graph. The algorithm consists of $(3/2)^f$ repetitions of the following: for each degree four vertex, choose randomly among the three possible partitions of its incoming edges into two sets of two edges; split the vertex into two vertices, with the edges assigned to one or the other vertex according to the partition, and connect the two vertices by a new forced edge (Figure 5). Once all vertices are split, the graph has maximum degree 3 and we can apply our previous forced TSP algorithm.

It is not hard to see that each such split preserves the traveling salesman tour only when the two tour edges do not belong to the same set of the partition, which happens with probability 2/3; therefore, each repetition of the algorithm has probability $(2/3)^f$ of finding the correct TSP solution. Since there are $(3/2)^f$ repetitions, there is a bounded probability that the overall algorithm finds the correct solution. Each split leaves unchanged the parameter s used in our analysis of the algorithm for cubic graphs, so the time for the algorithm is $\mathcal{O}((3/2)^f 2^{n/3}) = \mathcal{O}((27/4)^{n/3})$. By increasing the number of repetitions the failure probability can be made exponentially small with only a polynomial increase in runtime. We omit the details as our time bound for this case seems unlikely to be optimal.

6 Weighted Counting

Along with NP-complete problems such as finding traveling salesman tours, it is also of interest to solve #P-complete problems such as counting Hamiltonian cycles. More generally, we consider the following *weighted counting* problem: the edges of G are assigned weights from a *commutative semiring*: that is, an algebraic system with commutative and associative multiplication and addition operations, containing an additive identity, and obeying the distributive law of multiplication over addition. For each Hamiltonian cycle in G, we form the product of the weights of the edges in the cycle, and then sum the products for all cycles, to form the *value* of the problem.

The traveling salesman problem itself can be viewed as a special case of this semiring weighted counting problem, for a semiring in which the multiplication operation is the usual real number addition, and the addition operation is real number minimization. The additive identity in this case can be defined to be the non-numeric value $+\infty$. The problem of counting Hamiltonian cycles can also be viewed in this framework, by using the usual real number multiplication and addition operations to form a semiring (with additive identity zero) and assigning unit weight to all edges.

As we show in Table 2, most of the steps of our traveling salesman algorithm can be generalized in a straightforward way to this semiring setting. However, we do not know of a semiring analogue to the minimum spanning tree algorithm described in step 2 of Table 1, and proven correct in Lemma 1 for graphs in which the unforced edges form disjoint 4-cycles. It is tempting to try using the matrix-tree theorem to count spanning trees instead of computing minimum spanning trees, however not every Hamiltonian cycle of the input graph G corresponds to a spanning tree of the derived graph G' used in that step. Omitting the steps related to these 4-cycles gives the simplified algorithm shown in Table 2. We analyze this algorithm in a similar way to the previous one; however in this case we use as the parameter of our analysis the number of unforced edges $U(G)$ in

Table 2. Forced Hamiltonian cycle counting algorithm for graph G, forced edges F.

1. Repeat the following steps until one of the steps returns or none of them applies:
   a) If G contains a vertex with degree zero or one, return zero.
   b) If G contains a vertex with degree two, add its incident edges to F.
   c) If F consists of a Hamiltonian cycle, return the product of edge weights of this cycle.
   d) If F contains a non-Hamiltonian cycle, return zero.
   e) If F contains three edges meeting at a vertex, return zero.
   f) If F contains exactly two edges meeting at some vertex, remove from G that vertex and any other edge incident to it; replace the two edges by a single edge connecting their other two endpoints, having as its weight the product of the weights of the two replaced edges' costs.
   g) If G contains two parallel edges, exactly one of which is in F, and G has more than two vertices, remove the unforced parallel edge from G.
   h) If G contains two parallel edges, neither one of which is in F, and G has more than two vertices, replace the two edges by a single edge having as its weight the sum of the weights of the two edges.
   i) If G contains a self-loop which is not in F, and G has more than one vertex, remove the self-loop from G.
   j) If G contains a triangle xyz, then for each non-triangle edge e incident to a triangle vertex, multiply the weight of e by the weight of the opposite triangle edge. Also, if the triangle edge opposite e belongs to F, add e to F. Remove from G the three triangle edges, and contract the three triangle vertices into a single supervertex.
2. If F is nonempty, let xy be any edge in F and yz be an adjacent edge in $G \setminus F$. Otherwise, if F is empty, let yz be any edge in G.
3. Call the algorithm recursively on $G, F \cup \{yz\}$.
4. Call the algorithm recursively on $G \setminus \{yz\}, F$.
5. Return the sum of the two numbers returned by the two recursive calls.

the graph G. Like $s(G)$, U does not increase at any step of the algorithm; we now show that it decreases by sufficiently large amounts at certain key steps.

Lemma 5. *Let a graph G be given in which neither an immediate return nor step 1(f) can be performed, let F be nonempty, and let $U(G)$ denote the number of unforced edges in G. Then the algorithm of Table 2, within a constant number of steps, reduces the problem to one of the following situations:*

– *a single subproblem G', with $U(G') \leq U(G) - 1$, or*
– *two subproblems G_1 and G_2, with $U(G_1), U(G_2) \leq U(G) - 4$, or*
– *two subproblems G_1 and G_2, with $U(G_1) \leq U(G) - 3$ and $U(G_2) \leq U(G) - 6$.*

We omit the proof, which is similar to that for Lemma 4.

Theorem 3. *For any graph G with maximum degree 3, set F of forced edges in G, and assignment of weights to the edges of G from a commutative semiring, we can compute the semiring sum, over all forced Hamiltonian cycles in G, of the product of weights of the edges in each cycle, in $\mathcal{O}(2^{3n/8})$ semiring operations.*

Proof. By the previous lemma, the number of semiring operations in the algorithm can be bounded within a constant factor by the solution to the recurrence

$$T(u) \leq 1 + \max\{T(u-1), 2T(u-4), T(u-3) + T(u-6)\}.$$

Standard techniques for linear recurrences give the solution as $T(u) = \mathcal{O}(2^{u/4})$. In any n-vertex cubic graph, u is at most $3n/2$, so expressed in terms of n this gives a bound of $\mathcal{O}(2^{3n/8})$ on the number of operations. □

Corollary 1. *We can count the number of Hamiltonian cycles in any cubic graph in time* $\mathcal{O}(2^{3n/8} n^{\mathcal{O}(1)})$.

The extra polynomial factor in this time bound accounts for the time to perform each multiplication and addition of the large numbers involved in the counting algorithm. However, the numbers seem likely to become large only at the higher levels of the recursion tree, while the bulk of the algorithm's time is spent near the leaves of the tree, so perhaps this factor can be removed.

References

1. R. Beigel. Finding maximum independent sets in sparse and general graphs. *Proc. 10th ACM-SIAM Symp. Discrete Algorithms*. pp. S856–S857, January 1999, http://www.eecs.uic.edu/~beigel/papers/MIS-SODA:ps:GZ:
2. J.M. Byskov. Chromatic number in time $\mathcal{O}(2.4023^n)$ using maximal independent sets. Tech. Rep. RS-02-45, BRICS. December 2002
3. D. Eppstein. Small maximal independent sets and faster exact graph coloring. *Proc. 7th Worksh. Algorithms and Data Structures*, pp. 462–470. Springer-Verlag. Lecture Notes in Computer Science 2125, August 2001, arXiv:cs.Ds/0011009.
4. D. Eppstein. Quasiconvex analysis of backtracking algorithms. ACM Computing Research Repository, April 2003, arXiv:cs.DS./0304018.
5. M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979
6. E.L. Lawler. A note on the complexity of the chromatic number problem. *Information Processing Letters* 5(3):66–67, August 1976
7. J.M. Robson. Algorithms for maximum independent sets. *J. Algorithms* 7(3):425–440, September 1986
8. R.E Tarjan and A. Trojanowski. Finding a maximum independent set. *SIAM J. Comput.* 6(3):537–546, September 1977