Constrained TSP and Low-Power Computing

Moses Charikar[⋆1], Rajeev Motwani[1⋆⋆1], Prabhakar Raghavan[⋆⋆⋆2], Craig Silverstein[†1]

[1] Department of Computer Science, Stanford University, Stanford, CA-94305, USA
[2] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, USA

Abstract. In the *precedence-constrained traveling salesman problem (PTSP)* we are given a partial order on n nodes, each of which is labeled by one of k points in a metric space. We are to find a visit order consistent with the precedence constraints that minimizes the total cost of the corresponding path in the metric space. We give negative results on approximability by relating the problem to the Shortest Common Supersequence problem, helping to explain why there has been very little success in approximation algorithms for this problem. We also give approximation algorithms for a number of special cases, included cases appropriate for a problem in low-power computing; in the process, we show that algorithms for the k-server problem and the traveling salesman problem can be used to derive approximation algorithms for the PTSP. We give tight bounds on the approximation ratios achieved by natural classes of algorithms for this optimization problem (which include algorithms proposed and used in empirical studies of this problem). We briefly summarize results of experiments with several algorithms on a standard set of compiler benchmarks, comparing several known and new algorithms.

1 Overview

In the *precedence-constrained traveling salesman problem (PTSP)*, each node of a given directed, acyclic graph (DAG) is labeled by a point of a metric space $\mathcal{M}$. We wish to serialize the DAG so that the induced tour through its vertices is of the shortest possible length, where the distance between successive nodes on the tour is the distance between the corresponding points in $\mathcal{M}$. This classic generalization of the TSP has been extensively studied for its applications to vehicle routing [3, 4, 8, 12], with essentially no theoretical approximation guarantees. In this paper we explain this lack of success through

⋆ E-mail: `moses@cs.stanford.edu`. Supported by Stanford School of Engineering Groswith Fellowship, an ARO MURI Grant DAAH04-96-1-0007 and NSF Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

⋆⋆ E-mail: `rajeev@cs.stanford.edu`. Supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Partnership Award, an ARO MURI Grant DAAH04-96-1-0007, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

⋆⋆⋆ E-mail: `pragh@almaden.ibm.com`.

† E-mail: `csilvers@cs.stanford.edu`. Supported by the Department of Defense, with partial support from ARO MURI Grant DAAH04-96-1-0007 and NSF Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

a negative result showing that the PTSP is hard to approximate even on simple metric spaces such as the line and the hypercube, showing that it is not possible to exploit geometry to obtain good approximation algorithms for PTSP. (Previous hardness results, due to Bhatia, Khuller and Naor [2], do not apply to these simple metric spaces.) The PTSP on the hypercube is important because of its connection to an application to low-power computing, which we describe below. Because of the difficulty of approximation, we attack the PTSP problem by developing heuristics that work well for special cases. We prove upper bounds for several heuristics, some previously proposed in the context of low-power computing and some newly motivated by theoretical connections between the PTSP and the k-server problem. For this latter class of heuristics we can show our upper bound is tight. One of our results is related to the *dial-a-ride* or *taxicab problem*.

Section 3 below proves our main negative result. This result uses a connection of the PTSP to the Shortest Common Supersequence problem (SCS), thereby explaining the lack of prior success in finding approximation algorithms for the PTSP. We give our positive results in Section 4. We conclude in Section 5 with an experimental summary.

1.1 Low-Power Computing

The demand for ubiquitous mobile computing [23] has greatly outstripped the improvement in the performance of lightweight batteries. All notebook computers on the market employ clever power management techniques, and researchers are converging to CPU power minimization as key issue in mobile systems [6, 9, 13, 19, 20, 21]. Recently, Yao *et al.* [24] have applied competitive analysis to an online model for a scheduling problem related to power minimization. Their approach has its limitations, partly due to processor technology and more importantly due to the fact that operating system scheduling is inapplicable to mobile embedded systems. For example, it does not help in embedded DSP (digital signal processing) chips designed for dedicated applications [13, 20].

The major source of power consumption in a CMOS processor is in energy dissipation due to state changes in the logic circuits. Thus, the main avenue for reducing CPU power consumption is in reducing the switching activity [1, 11]. Several papers in the low-power literature have made proposals on reducing switching activity during a clock cycle [16, 17, 22]. Su, Tsui, and Despain [19] model power consumption at the architecture level by the switching activity at the input/output of the processor modules/chips. Specifically, they model the power consumption by the number of bit switches at all positions of the instruction word as the CPU executes a stream of instructions.

Su *et al.* [19] perform instruction scheduling within basic blocks (that is, branch-free segments of code) to minimize the number of bit switches. Instructions within basic blocks can be freely reordered subject to data dependencies that are represented as a DAG. Su *et al.* [19] adopt the approach of scheduling basic blocks so as to minimize the total amount of bit switches rather than the common approach of trying to minimize running time. In experiments with a simple greedy scheduler, they discovered that it is possible to reduce the number of bit switches by as much as 30% over the schedule geared towards optimizing the running time, without degrading the running time performance by more than 5%.

This basic block scheduling problem is an instance of PTSP, where the metric space is a hypercube since the distance between instructions is the Hamming distance. Though the lower bounds in Section 3 apply to this metric space, experimental data in Section 5 show that many scheduling algorithms for PTSP work well on standard benchmarks.

2 Preliminaries

Let $\mathcal{M}$ be a metric space of size k. For points $i, j \in \mathcal{M}$, let $d(i, j)$ be the distance between i and j in $\mathcal{M}$. Let $X = x_1, x_2, \ldots, x_m$ be a sequence of points in $\mathcal{M}$. We define the *cost* of a sequence X as follows:

$$\text{cost}(X) = \sum_{i=1}^{m-1} d(x_i, x_{i+1}). \tag{1}$$

Definition 1 *A* labeled directed acyclic graph (LDAG) *is a DAG $G(V, E)$ in which each vertex $v \in V$ has a label $l(v) \in \mathcal{M}$, where $\mathcal{M}$ is a metric space.*

Definition 2 *A* repetition-free LDAG *is an LDAG where each vertex has a distinct label.*

A *schedule* for an LDAG $G(V, E)$ is an ordering of V. The cost of a schedule $S = s_1, \ldots, s_n$ is the cost of its label sequence, namely cost$(l(s_1), l(s_2), \ldots, l(s_n))$. A schedule S for an LDAG G is said to be *legal* for G if it respects the precedence constraints of G (i.e., for all $v_i, v_j \in V$, if v_i is a predecessor of v_j, then v_i occurs before v_j in S.)

Definition 3 *The Precedence-Constrained TSP (PTSP) problem is the following: given an LDAG $G(V, E)$, find a legal schedule of minimum cost.*

In applications such as low-power computing, the precedence constraints could necessitate revisiting vertices, making the solution differ from that of standard TSP. To verify that our hardness results do not depend on this property, we study the following variant:

Definition 4 *The PTSP without repetitions is defined as follows: given a repetition-free LDAG $G(V, E)$, find a legal schedule of minimum cost.*

3 Lower Bounds

In this section, we develop a connection between PTSP and SCS, the Shortest Common Supersequence problem. The SCS problem is as follows: Given a set of strings $X = s_1, s_2, \ldots s_m$, construct a string S, of minimum length, such that each s_i is a (possibly non-contiguous) subsequence of S. The SCS problem is known to be difficult to solve exactly [7] or even approximately [2]. We first review the hardness results for SCS from Bhatia, Khuller, and Naor [2].

Theorem 1. *For the SCS problem with a fixed alphabet size k, there is no polynomial time k^α-approximation algorithm for some $\alpha > 0$, unless $P = NP$.*

Theorem 2. *For the SCS problem of size n, and for any constant δ, there is no polynomial time $(\log n)^\delta$-approximation algorithm unless $NP \subseteq DTIME(n^{O(\log \log n)})$.*

These hardness results for SCS apply directly to PTSP on the uniform metric space. However, they do not rule out the possibility of finding better approximations for PTSP on simple geometric spaces such as the line, the plane and — the case of most interest in low-power computing — the hypercube. By proving hardness of approximation results similar to Theorems 1 and 2 for the line and the hypercube, we show that, in fact, we cannot hope to exploit the structure of these simple spaces to obtain better approximations for PTSP.

3.1 Hardness on the Line and Hypercube

By converting instances of SCS to "large" instances of PTSP, we can prove hardness results even when $\mathcal{M}$ is a simple metric space. We first consider when $\mathcal{M}$ is the line.

Consider an instance X of the SCS problem over a fixed alphabet $\{c_1, \ldots, c_k\}$. The instance X is a set of strings over the alphabet. Given X, we construct an instance $P(X)$ of PTSP, where $P(X)$ consists of a collection of chains, one for each string in X. The length of each chain is much longer than the length of its corresponding string, because we associate with each letter c_i an entire chain C_i to be described below. The chain corresponding to the string $c_{i_1}, c_{i_2}, \ldots, c_{i_r}$ is $C_{i_1} \prec C_{i_2} \prec \cdots \prec C_{i_r}$.

C_i (for $1 \leq i \leq k$) is defined to be the $m = 2n^2$-length chain $(a_i \prec b_i)^{m/2}$, where a_i and b_i are points with coordinates $2i-1$ and $2i$ respectively. D_i is the corresponding schedule $(a_i b_i)^{m/2}$. Clearly D_i is a legal schedule for C_i. A *regular* solution of a PTSP instance is a solution of the form $D_{i_1} D_{i_2} \ldots D_{i_r}$. If $S = c_{i_1} \ldots c_{i_r}$ solves an instance X of SCS then $R = D_{i_1} \ldots D_{i_r}$ solves the corresponding instance $P(X)$ of PTSP, and *vice versa*. S has cost r while $(m-1)r + r - 1 \leq \text{cost}(R) \leq (m-1)r + 2n(r-1)$. This leads to the following result.

Lemma 3. *Given a solution S of X, we can construct a (regular) solution R of $P(X)$ of cost at most $(m+2n)$cost(S). Conversely, given any regular solution R of $P(X)$ we can obtain a solution S for X with cost at most cost$(R)/m$.*

A solution U of $P(X)$ might schedule parts of different C_i's interspersed with each other. However, we can construct a regular solution which schedules the C_i's in the order in which they are completed by U. This gives us the following result.

Lemma 4. *Given any solution U of $P(X)$, we can obtain a* regular *solution of $P(X)$ with cost at most $(1 + 1/n)$cost(U).*

Using our reduction of SCS to PTSP and the results of the previous lemmas, we can show that a c-approximation algorithm for PTSP on the line can be used to construct a $c(1+1/n)^2$-approximation algorithm for SCS. Thus the hardness results of Theorem 1 extend to PTSP on the line.

Theorem 5. *For the PTSP problem on the line with k points, there is no polynomial time k^α-approximation algorithm for some $\alpha > 0$, unless $P = NP$.*

We can modify the above construction to work for the hypercube as well. In this case the points a_i and b_i are chosen such that $d(a_i, b_i) = 1$. The construction and proof proceed as before.

Theorem 6. *For the PTSP problem on the hypercube, there is no polynomial time k^α-approximation algorithm for some $\alpha > 0$, unless $P = NP$.*

3.2 Hardness for PTSP Without Repetitions

The construction in the previous section used the fact that the precedence constraints could force multiple visits to the same point in the metric space. If we replace each point from the previous reduction by a closely packed cluster of points, we can extend the hardness results of PTSP to cover PTSP without repetitions.

Let m be as before. We replace a_i and b_i by sets A_i and B_i, where (1) the size of each set is at least $nm = 2n^3$, (2) the distance between two points in distinct sets is at least d_1, (3) the distance between points in A_i and B_i is at most d_2, (4) the maximum distance between any two points is at most d_3, and (5) the points in any set A_i (or B_i) have a TSP tour $T(A_i)$ (or $T(B_i)$) of length at most d_4.

Since, in the reductions of the last section, each a_i and b_i was visited at most nm times, we can replace each occurrence of a_i (or b_i) with a unique element from A_i (or B_i). The version of $P(X)$ formed this way, without repetition, is denoted $P_d(X)$ (d is for "distinct"). A *regular* solution generalizes in the obvious way: $D_i = a_i^1 b_i^1 a_i^2 b_i^2 \ldots$, where each a_i^j is a unique element of A_i. The analogue of Lemma 3 is as follows.

Lemma 7. *Given a solution S of X, we can construct a* regular *solution R of $P_d(X)$ of cost at most $(d_2 m + d_3)cost(S)$. Conversely, given a* regular *solution R of $P_d(X)$ we can obtain a solution S for X with cost at most $\frac{1}{md_1}cost(R)$.*

Again, we can convert an arbitrary solution to a regular solution. The regular solution is built from sequences of the form $(T(A_i)T(B_i))^{m/2}$.

Lemma 8. *Given a solution U of $P_d(X)$ we can obtain a regular solution of $P_d(X)$ with cost at most $\frac{m(d_4+d_2)+d_3}{md_1}cost(U)$.*

For the line, we choose $A_i = [(3i-1)n^4 - 2n^3, (3i-1)n^4]$ and $B_i = [(3i)n^4, (3i)n^4 + 2n^3]$, for $1 \leq i \leq n$. For this choice of A_i, B_i, we have $d_1 = n^4$, $d_2 = n^4 + 4n^3$, $d_3 = 3n^5$, and $d_4 = 4n^3$. Thus, given an instance X of SCS of size n, our reduction constructs an instance of PTSP without repetitions on a discrete line of length $O(n^5)$. The previous lemmas imply that a c-approximation algorithm for PTSP without repetitions in the discrete line can be used to construct a $c(1+10/n)^2$-approximation algorithm for SCS. The hardness result of Theorem 2 now extends to PTSP without repetitions.

Theorem 9. *For PTSP without repetitions on the discrete line of length k, and for any $\delta > 0$, there is no polynomial time $(\log k)^\delta$-approximation algorithm, unless $NP \subseteq DTIME(n^{\log\log n})$.*

Observe that we can embed a discrete line of length l in the l-dimensional hypercube. We obtain the same approximation results as before, but now the size of the metric space is $2^{O(n^5)}$, where n is the size of the original SCS problem. This gives us the following hardness result.

Theorem 10. *For PTSP without repetitions on the hypercube of size k, and for any $\delta > 0$, there does not exist a polynomial time $(\log\log k)^\delta$-approximation algorithm, unless $NP \subseteq DTIME(n^{\log\log n})$.*

4 Upper Bounds

Before proving upper bounds, we define a connection between schedules for G and sequences on $\mathcal{M}$. There is a natural connection due to the fact that each vertex in G has an associated point in $\mathcal{M}$.

Recall that a *schedule* is a list of vertices in G. We define a *sequence* as a (possibly infinite) list of points in $\mathcal{M}$. Every schedule $v_1, \ldots, v_n$ has a sequence *corresponding to* it, namely $l(v_1), \ldots, l(v_n)$ with duplicates removed — that is, we remove $l(v_i)$ if $l(v_i) = l(v_{i-1})$. We say a sequence X is *legal* for G if some (possibly non-contiguous) subsequence of X is a sequence corresponding to a legal schedule. (Legal sequences are thus sequences associated with legal schedules, possibly augmented by dummy nodes.)

For a chain $\mathcal{C} = v_1 \prec \cdots \prec v_m \in G$, let $l(\mathcal{C})$ be the sequence $l(v_1), \ldots, l(v_n)$ with duplicates removed. The following lemma characterizes legal sequences for G.

Lemma 11. *X is legal for G if and only if X is a supersequence of $l(\mathcal{C})$ for every maximal chain $\mathcal{C}$ in G.*

Note that the cost of a schedule, as defined in Section 2, is exactly the cost of its corresponding sequence, since removing duplicates does not affect the cost.

To define the flip side of this correspondence, we must introduce the concept of *eager schedules*. A *ready vertex* is a vertex all of whose predecessors have been scheduled. An *eager scheduling algorithm* proceeds as follows: initially, pick a point $p \in \mathcal{M}$. While there is still a ready vertex with label p, schedule it. Repeat this until no ready vertex with label p remains. Repeat this scheduling process with a new point, continuing until all vertices in G have been scheduled. An *eager schedule* is a schedule produced by an eager scheduling algorithm. Eager schedules differ from arbitrary schedules in that they do not move in the metric space unless they have to. Thus, an eager schedule is uniquely defined by the sequence of points picked by the algorithm. The schedule *corresponding to* a sequence X is the unique eager schedule produced by an eager scheduling algorithm which selects X as the sequence of points to be scheduled.

The cost of a sequence X is an upper bound on the cost of its corresponding schedule S. The sequence could be more expensive if the eager scheduling algorithm were to choose a point p for which there was no ready vertex. Then X has to pay the cost for switching to p while S does not. We can remove a point x_i from X if, while constructing an eager schedule based on X, there is no ready vertex with label x_i when we encounter x_i. This act of removing unused points is called *short-circuiting*. If we restrict our attention to short-circuited sequences and eager schedules, the correspondence mapping preserves cost. The following lemma justifies restricting ourselves to eager schedules.

Lemma 12. **(a)** *Given any schedule for G with cost c, there is an eager schedule for G with cost at most c.* **(b)** *Given any sequence legal for G with cost c, there is a short-circuited sequence legal for G with cost at most c.*

A special kind of short-circuiting occurs when X is much longer than is needed by any eager scheduling algorithm (for instance, when X is infinite). In this case, we wish to restrict our attention to the shortest prefix of X which is legal for G, denoted pre(X, G). We also define $f(X, X')$ to be the shortest prefix of X that is a supersequence of X'.

From now on, we will refer to our solution as a sequence X (on $\mathcal{M}$) instead of a schedule (on V). The schedule implied is the eager schedule corresponding to pre(X, G). Thus, the cost of scheduling X for G is at most cost(pre(X, G)), though it could be lower due to short-circuiting.

In the following sections, we give good approximation algorithms for several special cases of the PTSP. We should note that an exact solution can be easily obtained, using dynamic programming, in time n^w, where w is the size of the largest anti-chain [5]. In general, however, we will be searching for polynomial time algorithms.

We start with an upper bound for a special case of PTSP on the line. This case is exactly the *dial-a-ride problem* on the line [12].

Theorem 13. *There is a polynomial time exact algorithm for PTSP (without repetitions) on the line when the precedence constraints are a collection of chains of length* 2.

4.1 $k-1$ Approximation via Universal Sequences

Definition 5 *A* universal sequence algorithm *for PTSP is an algorithm that produces a schedule corresponding to a fixed sequence* U*, where* U *depends only on the metric space* $\mathcal{M}$.

That is, a universal sequence algorithm ignores the structure of the LDAG. We refer to U as the *universal sequence* of the algorithm. For a finite sequence X, X^* denotes the sequence X repeated infinitely often. If a universal sequence U equals X^* for some finite sequence X, then the corresponding algorithm is called a *repeating sequence algorithm*.

For a universal sequence U on $\mathcal{M}$, we say that U is an (r, β)*-sequence* if for all finite sequences X without duplicates, cost$(f(U, X)) \leq r \cdot$ cost$(X) + \beta$.

Lemma 14. *If a universal sequence is an* (r, β)*-sequence, then its corresponding universal sequence algorithm is an* r*-approximation algorithm for PTSP, with additive error at most* β.

There is an interesting connection between universal sequences and algorithms for the celebrated k-server problem. Consider a k point metric space $\mathcal{M}$. Let A be a $(k-1)$-competitive algorithm for the $(k-1)$-server problem on $\mathcal{M}$ (see [14]). We use A to obtain a $(k-1, \beta)$-sequence, where β is a constant. By Lemma 14, this gives a $(k-1)$-approximation algorithm for PTSP with a constant additive error.

To define the PTSP sequence, note that the configuration of the $(k-1)$ servers is completely specified by the single point in $\mathcal{M}$ not covered by any server. Define the *state* of A after i requests, s_i, to be this uncovered point. We generate a request sequence $s_0, s_1, s_2, \ldots$, at each step requesting the point left uncovered after the previous step. The universal sequence we derive from A is the request sequence $U = s_0, s_1, \ldots$.

Lemma 15. U *is a* $(k-1, \beta)$*-sequence, where* β *is a constant depending only on* $\mathcal{M}$.

Besides being applicable to PTSP, our universal sequence construction gives a $(\rho - 1)$-approximation (with at most constant additive error) for the Loading Time Scheduling Problem (LTSP), improving the ρ approximation of [2]. This is not an insignificant improvement since ρ, the number of machines, is typically very small (about 4 or 5). The distance function for LTSP is not a metric, since $d(i, j)$ depends only on j. However, $d'(i, j) = (d(i, j) + d(j, i))/2$ defines a metric and has only an additive difference from $d(i, j)$. Thus, the algorithms described above apply to d'.

As the following lemma shows, the bound in Lemma 15 is tight.

Lemma 16. *No universal sequence U can be an (r, β)-sequence for any $r < k - 1$ and constant β.*

Despite Lemma 16, it is conceivable that a universal sequence algorithm might have an approximation ratio better than $k - 1$ due to short-circuiting. For repeating sequence algorithms, however, we can prove that this is not the case.

Theorem 17. *No repeating sequence algorithm for PTSP can have an approximation ratio better than $k - 1 - \epsilon$ for any $\epsilon > 0$.*

4.2 Approximation via TSP

A natural type of repeating sequence algorithm for a k-point metric space repeats sequences of length k; that is, it cycles through $\mathcal{M}$ in a predetermined order. The optimal algorithm of this type uses a TSP cycle. Let T be an optimal TSP cycle, and let T' be its reverse. Let $U_T = T^*$ and $U'_T = T'^*$. Let d_{min} be the minimum distance between two points in $\mathcal{M}$. If the length of the longest chain is ℓ, then cost(U_T) + cost$(U'_T) \leq \ell \cdot$ cost$(T) + \beta$, while the optimal schedule has cost $\geq \ell \cdot d_{min}$. We have the following.

Theorem 18. *The algorithm that schedules instructions using the lower-cost sequence U_T or U'_T is an r-approximation algorithm for PTSP (with at most constant additive error), where $r =$ cost$(T)/2d_{min}$.*

Note that the TSP-based algorithm is not a repeating sequence algorithm, but rather one that takes the best of two repeating sequence algorithms.

If $\mathcal{M}$ is uniform, or if $\mathcal{M}$ is the hypercube, $d_{min} = 1$ while cost$(T) = k$ (for the hypercube, take the points in Grey code order). In each case we get a $k/2$ approximation, beating the lower bound for repeating sequence algorithms.

4.3 Greedy Algorithms

There are several natural heuristic algorithms for PTSP. All of these produce eager schedules. They produce the schedule incrementally, choosing the next point of $\mathcal{M}$ at each stage by looking at the vertices remaining to be scheduled.

Let $p \in \mathcal{M}$ be the last point scheduled. Let $p' \in \mathcal{M}$ be a point that labels some ready vertex, and $n(p')$ be the number of ready vertices that have label $l(p')$. The COST GREEDY heuristic picks the point p' that minimizes $d(p, p')$; this is essentially the heuristic of Su *et al.* [19]. The NODE GREEDY heuristic picks the point p' minimizing $n(p')$. The WEIGHTED GREEDY heuristic picks the p' minimizing $d(p, p')/n(p')$.

A *critical path* in an LDAG is a chain p such that cost$(l(p))$ is maximized over all chains p. The CRITICAL PATH heuristic works as follows: at each step, it selects a ready vertex j which belongs to a critical path; the next point scheduled is the label of j.

As Figure 1 shows, all of these algorithms have unbounded competitive ratios, even when $|\mathcal{M}|$ is as small as 4.

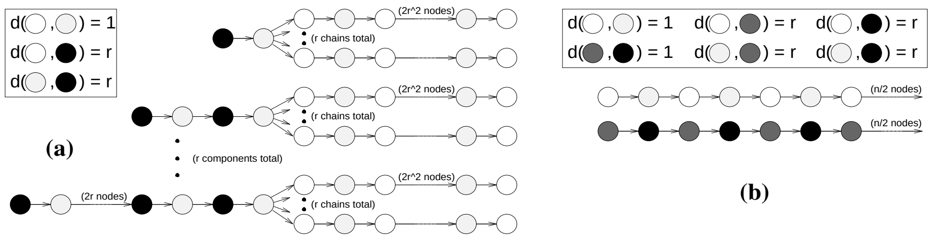

Fig. 1. (a) A lower-bound example for the greedy algorithms. The optimal algorithm schedules the points in the order of the bottom component, incurring cost $2r^2 - r + 2r^2 = 4r^2 - r$. All three greedy algorithms, however, will schedule the components from top to bottom, incurring cost $2r^2 - r + 2r^3$. Since r is arbitrary, this is an unbounded ratio. **(b)** A lower-bound example for CRITICAL PATH. An optimal algorithm incurs cost $n + r$ by scheduling all the points in the first chain and then all the points in the second. CRITICAL PATH will alternate chains, however, incurring cost at least $rn/2+n$. This gives competitive ratio $r/4$ (for $n \geq r$), which is unbounded.

5 Experimental Results

We tested our heuristics on basic blocks generated from the standard SPEC92 benchmark suite [18] that is commonly used in compiler research. We used the experimental setup of Motwani, Palem, Reyen, and Sarkar [15], in that we used the input data generated by them (for an unrelated experimental study) as the input for our empirical testing. They compiled the SPEC92 benchmark programs for the SuperSPARC processor, with full optimization turned on in `gcc`, the Gnu C compiler. They added a small phase to `gcc` that, for each basic block, extracts the corresponding problem instance.

For each of the 19 benchmarks we tested, we broke each function of the benchmark into basic blocks. We threw out basic blocks with less than 10 instructions in `gcc`'s internal representation. To test a benchmark, we ran the scheduling algorithm for each basic block in the benchmark and calculated the cost of following the schedule devised by the algorithm. We took the average cost per basic block to be the cost of the benchmark for that algorithm.

Each instruction is 32 bits long, and the cost to change from one instruction to the other is the number of bits by which the instructions differ. Thus, $\mathcal{M}$ is a 32-dimensional hypercube, and each instruction is a point in $\mathcal{M}$. In reality, not every 32-bit sequence corresponds to a legal instruction, and not all basic blocks use all instructions, so for a given basic block $|\mathcal{M}|$ may be significantly less than 2^{32}. Each vertex of the precedence graph is labeled by an instruction, so a *ready instruction* is one that labels a ready vertex.

We tested the following eager scheduling algorithms.

NGreedy – NODE GREEDY picks the most frequently occurring ready instruction.

CGreedy – COST GREEDY picks the instruction among the ready instructions that is closest (in terms of cost) to the current instruction. This is the algorithm of [19].

WGreedy – WEIGHTED GREEDY combines the other greedy schemes, picking the instruction among the ready instructions that minimizes cost/node count.

SPEC92 Benchmark	Number of basic blocks	Range of bb sizes (# instructions)	SPEC92 Benchmark	Number of basic blocks	Range of bb sizes (# instructions)
008.espresso	489	10 – 74	052.alvinn	11	10 – 17
013.spice2g6	1746	10 – 405	056.ear	113	10 – 74
015.doduc	575	10 – 903	072.sc	201	10 – 39
022.li	96	10 – 201	077.mdljsp2	108	10 – 303
023.eqntott	34	10 – 25	078.swm256	48	10 – 105
026.compress	18	10 – 29	089.su2cor	269	10 – 178
034.mdljdp2	108	10 – 308	090.hydro2d	205	10 – 281
039.wave5	739	10 – 274	093.nasa7	145	10 – 196
047.tomcatv	26	10 – 158	094.fpppp	124	10 – 4182
048.ora	19	10 – 100	**All 19 tests**	**5063**	**10 – 4182**

Table 1. Characteristics of basic blocks used in performance measurements.

Benchmark	NGreedy	CGreedy	WGreedy	TSP	KSrv	KSrv-NoS	CGreedy+KSrv
008.espresso	102%	100%	100%	100%	101%	267%	100%
013.spice2g6	108%	100%	102%	103%	103%	296%	100%
015.doduc	114%	100%	102%	106%	103%	294%	100%
022.li	102%	100%	100%	100%	102%	240%	100%
023.eqntott	101%	100%	100%	99%	100%	248%	100%
026.compress	102%	100%	100%	100%	102%	250%	100%
034.mdljdp2	112%	100%	104%	103%	105%	270%	101%
039.wave5	109%	100%	101%	103%	103%	289%	100%
047.tomcatv	110%	100%	105%	105%	107%	283%	100%
048.ora	108%	100%	100%	103%	102%	299%	100%
052.alvinn	104%	100%	100%	100%	101%	264%	100%
056.ear	104%	100%	100%	101%	101%	265%	100%
072.sc	102%	100%	100%	100%	101%	244%	100%
077.mdljsp2	108%	100%	103%	102%	103%	266%	101%
078.swm256	110%	100%	101%	103%	105%	314%	100%
089.su2cor	108%	100%	102%	103%	105%	306%	100%
090.hydro2d	106%	100%	102%	101%	104%	273%	100%
093.nasa7	109%	100%	102%	104%	105%	329%	100%
094.fpppp	117%	100%	105%	109%	104%	284%	100%

Table 2. The average cost of a basic block, taken as a percent of the average cost for CGREEDY.

Benchmark	NGreedy	CGreedy	WGreedy	TSP	KSrv	KSrv-NoS	CGreedy+KSrv
013.spice2g6	108%	100%	98%	102%	101%	380%	100%
015.doduc	117%	100%	104%	113%	104%	420%	101%
022.li	106%	100%	103%	101%	109%	234%	105%
034.mdljdp2	118%	100%	114%	105%	116%	286%	108%
039.wave5	141%	100%	109%	115%	106%	357%	100%
077.mdljsp2	115%	100%	111%	105%	114%	290%	107%
090.hydro2d	109%	100%	109%	100%	105%	164%	100%
094.fpppp	188%	100%	138%	154%	106%	266%	100%

Table 3. The average cost of large basic blocks (≥ 200 nodes) as a percent of CGREEDY's cost.

TSP – TSP orders the instructions in TSP order. (Actually, we use a 2-approximation algorithm to obtain an approximate TSP ordering.) It picks its next instruction to be the next instruction in TSP order, short-cutting non-ready instructions. It then tries ordering instructions in reverse TSP order and takes the better of the two runs.

KSrv – KSERVER uses the $k-1$-sever algorithm of [14] to obtain a universal sequence. It picks its next instruction to be the next instruction in this universal sequence, short-cutting non-ready instructions.

KSrv-NoS – KSERVER WITHOUT SHORTCUT is the same as KSERVER, but it does not short-cut, and therefore pays for instructions even if they are not used.

CGreedy+KSrv – COST GREEDY PLUS KSERVER combines the two algorithms: it runs COST GREEDY for 100 distinct instructions, adding weights to the $k-1$ servers as if the instructions changes were being made by KSERVER (that is, when changing from node i to node j, it updates server weights as if a server were moving from j to i). After 100 instructions, it switches to KSERVER, using the server weights from COST GREEDY as an initial configuration for the KSERVER algorithm.

The results of these experiments are shown in Figures 2 and 3. Figure 2 indicates that none of the theoretically motivated algorithms — TSP, KSRV, and their variants — outperformed CGREEDY, the best algorithm in our tests, and Figure 3 shows that the difference in performance often grew as the size of the basic block increased. Several of the theoretically motivated algorithms, however, were competitive with CGREEDY and all have superior theoretical properties. It is known that a greedy algorithm performs well on the average for the Shortest Common Supersequence problem [10]. The similarity between that problem and the PTSP could explain why greedy algorithms for the PTSP perform better than predicted by worst case analysis.

Acknowledgements

We are grateful to Sharad Malik for helpful discussions and pointers to the relevant literature. We are deeply indebted to the authors of [15] for sharing their experimental set-up, and in particular to Salem Reyen for his help with the experiments.

References

1. A. Aggarwal, A.K. Chandra and P. Raghavan. Energy Consumption in VLSI Circuits. In *Proceedings of the ACM Symposium on Theory of Computing*, 1988, pp. 205–216.
2. R. Bhatia, S. Khuller, and J. Naor. The Loading Time Scheduling Problem. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, 1995, pp. 72–81.
3. L. Bianco, A. Mingozzi, S. Riccardelli, and M. Spadoni. Exact and Heuristic Procedures for the Traveling Salesman Problem with Precedence Constraints, Based on Dynamic Programming. *INFOR*, 32(1):19–32, 1994.
4. N. Christofides. Vehicle Routing. In *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization* (Ed: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys), John Wiley & Sons (1985), pp. 431–448.
5. C.J. Colbourn and W.R. Pulleyblank. Minimizing Setups in Ordered Sets of Fixed Width. *Order*, 1:225–229 (1985).

6. S. Devadas and S. Malik. A Survey of Optimization Techniques Targeting Low-Power VLSI Circuits. Preprint, 1995.
7. M. Garey and D. Johnson. *Computers and Intractability — a Guide to the Theory of NP-completeness*. W.H. Freeman, San Francisco, 1979.
8. B.L. Golden and A.A. Assad, Eds. *Vehicle Routing: Methods and Studies.* North-Holland, Amsterdam (1988).
9. K. Govil, E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. Preprint, 1995.
10. T. Jiang and M. Li. On the Approximation of Shortest Common Supersequences and Longest Common Subsequences. In *Proceedings of 21st International Colloquium on Automata, Languages and Programming*, 1994, pp. 191–202.
11. G. Kissin. Energy Consumption in VLSI Circuits: a Foundation. *ACM Symposium on Theory of Computing*, 1982.
12. M. Kubo and H. Kagusai. Heuristic Algorithms for the Single-Vehicle Dial-a-ride Problem. *Journal of the Operations Research Society of Japan*, 30(4):354–365, 1990.
13. M. T. Lee, V. Tiwari, S. Malik, and M. Fujita. Power Analysis and Low-Power Scheduling Techniques for Embedded DSP Software. Technical Report FLA-CAD-95-01, Fujitsu Labs of America, March 1996.
14. M. Manasse, L. McGeoch, and D. Sleator. Competitive Algorithms for Server Problems. *Journal of Algorithms*, 11:208–230 (1990).
15. R. Motwani, K. Palem, S. Reyen, and V. Sarkar. Combining Register Allocation and Instruction Scheduling. Submitted for publication, 1997. Preliminary version: *Technical Report STAN-CS-TN-95-22*, Department of Computer Science, Stanford University, 1996.
16. S. Prasad and K. Roy. Circuit Activity Driven Multilevel Logic Optimization for Low Power Reliable Operation. *Proceedings of the European Conference on Design Autoamtion (EDAC)*, 1993, pp. 368–372.
17. K. Roy and S. Prasad. SYSLOP: Synthesis of CMOS Logic for Low-Power Applications. In *Proceedings of the International Conference on Computer Design*, 1992.
18. SPEC Consortium. The SPECint95 and SPECfp95 Benchmarks. World-Wide Web URL `http://www.specbench.org`, 1995.
19. C-L. Su, C-Y. Tsui, and A.M. Despain. Low-Power Architecture Design and Compilation Techniques for High-Performance Processors. In *Proceedings of the IEEE COMPCON*, 1994, pp. 489–498.
20. V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transaction on VLSI Systems*, 2:437–445 (1994).
21. V. Tiwari, S. Malik, and A. Wolfe. Compilation Techniques for Low Energy: An Overview. In *Proceedings of the 1994 Symposium on Low-Power Electronics*, 1994, pp. 38-39.
22. C.Y. Tsui, M. Pedram, and A.M. Despain. Technology Decomposition and Mapping Targeting Low-Power Dissipation. In *Proceedings of the 30th Design Automation Conference*, 1993, pp. 68-73.
23. M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36:74–83 (1993).
24. F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, 1995, pp. 374–382.

This article was processed using the LaTeX macro package with LLNCS style